

FreeRange Computer Design: The RISC-V Otter MCU

Version: v10.10

©Copyright: 2020 James Mealy

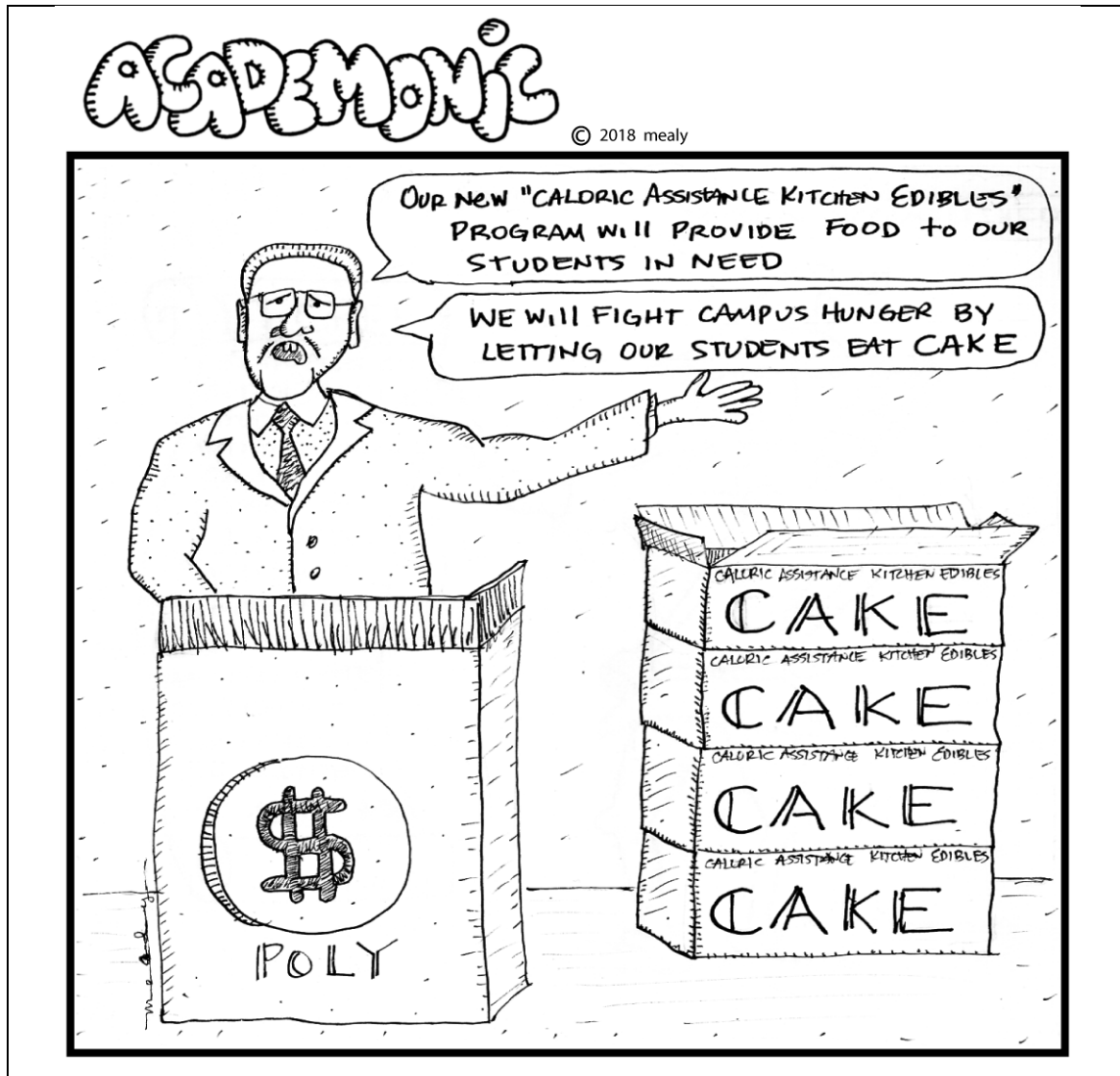


Table of Contents

TABLE OF CONTENTS.....	- 2 -
PRETENTIONS	- 11 -
LEGAL CRAP	- 11 -
ACKNOWLEDGEMENTS	- 12 -
RAMBLING COMMENTARY	- 13 -
OVERVIEW OF CHAPTER OVERVIEWS.....	- 15 -
PART ONE: INTRODUCTION AND REVIEW.....	- 19 -
1 FREERANGE COMPUTER DESIGN OVERVIEW	- 20 -
1.1 INTRODUCTION	- 20 -
1.2 CHAPTER STRUCTURE	- 20 -
1.3 FREERANGE COMPUTER DESIGN BEGINNINGS.....	- 20 -
1.4 ISSUES WITH “MODERN COMPUTER DESIGN”	- 21 -
1.5 THE RAT MICROCONTROLLER/MICROCOMPUTER.....	- 22 -
1.6 THE RISC-V OTTER MCU	- 23 -
1.7 ISSUES WITH THE CPE 233 COURSE	- 24 -
1.7.1 <i>The CPE 233 Approach</i>	- 24 -
1.8 CHAPTER SUMMARY	- 25 -
1.9 CHAPTER EXERCISES	- 26 -
2 DIGITAL DESIGN REVIEW.....	- 27 -
2.1 INTRODUCTION	- 27 -
2.2 THE DESIGN PROCESS.....	- 27 -
2.3 THE NEW DIGITAL PARADIGM: DIGITAL DESIGN FOUNDATION MODELING	- 28 -
2.3.1 <i>DDFM Overview</i>	- 28 -
2.3.2 <i>The Three Approaches to Digital Design</i>	- 30 -
2.3.3 <i>Notes on Modular Design Techniques</i>	- 31 -
2.4 IMPORTANT DIGITAL VOCABULARY	- 32 -
2.5 BASIC GATES	- 34 -
2.6 COMBINATORIAL CIRCUITS	- 36 -
2.6.1 <i>Half Adder</i>	- 36 -
2.6.2 <i>Full Adder</i>	- 37 -
2.6.3 <i>Ripple Carry Adder</i>	- 37 -
2.6.4 <i>Decoders</i>	- 38 -
2.6.4.1 Generic Decoder.....	- 39 -
2.6.4.2 Standard Decoder.....	- 40 -
2.6.5 <i>Multiplexor</i>	- 41 -
2.6.6 <i>Comparator</i>	- 43 -
2.7 SEQUENTIAL CIRCUITS	- 44 -
2.7.1 <i>Simple Registers</i>	- 44 -
2.7.1.1 Special Register Circuits: The Accumulator.....	- 46 -
2.7.2 <i>Counters: Registers with Features</i>	- 46 -
2.7.3 <i>Shift Registers</i>	- 49 -
2.7.4 <i>Registers: The Final Comments</i>	- 52 -
2.8 FINITE STATE MACHINES (FSMs)	- 53 -
2.8.1 <i>High-Level Modeling of Finite State Machines</i>	- 53 -
2.8.2 <i>The FSM: Symbology Overview</i>	- 54 -
2.8.2.1 The State Bubble	- 54 -

2.8.2.2	The State Diagram	- 55 -
2.8.2.3	State Transitions Controlling Conditions	- 56 -
2.8.2.4	FSM External Outputs.....	- 57 -
2.8.2.5	Non-Important FSM Outputs.....	- 58 -
2.8.2.6	Non-Important FSM Inputs.....	- 58 -
2.8.2.7	The Final State Diagram Summary.....	- 59 -
2.9	CHAPTER SUMMARY	- 60 -
2.10	CHAPTER EXERCISES	- 61 -
2.11	CHAPTER DESIGN PROBLEMS	- 62 -
3	ADVANCED REGISTERS.....	- 63 -
3.1	INTRODUCTION	- 63 -
3.2	REGISTERS: THE MOST COMMON DIGITAL CIRCUIT EVER?	- 63 -
3.3	TRI-STATE REGISTERS	- 64 -
3.4	BI-DIRECTIONAL REGISTERS	- 68 -
3.5	SHIFT REGISTERS	- 69 -
3.5.1	<i>Basic Shift Registers</i>	- 69 -
3.5.2	<i>Universal Shift Registers</i>	- 70 -
3.5.3	<i>Barrel Shifters</i>	- 72 -
3.5.4	<i>Other Shift Register-Type Features</i>	- 73 -
3.6	CHAPTER SUMMARY	- 75 -
3.7	CHAPTER EXERCISES	- 76 -
	PART TWO: ADVANCED DIGITAL DESIGN.....	- 84 -
4	CHAPTER: REGISTER TRANSFER NOTATION	- 85 -
4.1	INTRODUCTION	- 85 -
4.2	REGISTER TRANSFER NOTATION SPECIFICS.....	- 85 -
4.3	MICROOPERATIONS AND DATA TRANSFERS.....	- 89 -
4.3.1	<i>Transfer Microoperations</i>	- 89 -
4.3.2	<i>Arithmetic Microoperations</i>	- 90 -
4.3.3	<i>Logic Microoperations</i>	- 90 -
4.3.4	<i>Shift Microoperations</i>	- 91 -
4.4	DATA TRANSFER CIRCUITS	- 93 -
4.4.1	<i>MUX-Based Data Transfers</i>	- 93 -
4.4.2	<i>Bus-Based Data Transfers</i>	- 94 -
4.4.3	<i>Tri-State Bus-Based Transfers</i>	- 95 -
4.5	CHAPTER SUMMARY	- 99 -
4.6	CHAPTER EXERCISES	- 100 -
5	STRUCTURED MEMORY: RAM AND ROM.....	- 112 -
5.1	INTRODUCTION	- 112 -
5.2	MEMORY INTRODUCTION AND OVERVIEW.....	- 112 -
5.2.1	<i>Basic Memory Operations: READ and WRITE</i>	- 113 -
5.2.2	<i>Basic Memory Types: ROM and RAM</i>	- 113 -
5.3	SOFTWARE ARRAYS VS. HARDWARE STRUCTURED MEMORIES.....	- 114 -
5.4	MEMORY OPERATION DETAILS: READING AND WRITING.....	- 114 -
5.5	MEMORY SPECIFICATION AND CAPACITY	- 115 -
5.6	MEMORY INTERFACE DETAILS	- 116 -
5.7	MEMORY PERFORMANCE PARAMETERS	- 117 -
5.8	MEMORY MAPPING.....	- 125 -
5.9	MEMORY ORGANIZATION	- 128 -
5.9.1	<i>Extending Memory Word Length</i>	- 129 -
5.9.2	<i>Extending Memory Address Space</i>	- 130 -

5.10	DIGITAL DESIGN FOUNDATION NOTATION: RAM	- 143 -
5.11	CHAPTER SUMMARY.....	- 145 -
5.12	CHAPTER EXERCISES	- 146 -
5.13	CHAPTER DESIGN PROBLEMS	- 154 -
PART THREE: INTRODUCTION TO COMPUTERS		- 156 -
6	THE BASIC COMPUTER IN HIGH-LEVEL TERMS	- 157 -
6.1	INTRODUCTION	- 157 -
6.2	HIGH-LEVEL VIEW OF LEARNING “DIGITAL STUFF”	- 157 -
6.2.1	<i>Solving Problems with Digital Circuits</i>	- 157 -
6.2.2	<i>Solving Problems with Computers</i>	- 158 -
6.2.3	<i>Final Problem Solving Overview</i>	- 158 -
6.3	WHAT IS A COMPUTER?.....	- 158 -
6.4	YOU AND THE COMPUTER	- 159 -
6.5	COMPUTER ARCHITECTURE: FOR THE HARDWARE PEOPLE	- 160 -
6.6	COMPUTER ARCHITECTURE: FOR THE PROGRAMMER PEOPLE.....	- 161 -
6.6.1	<i>Programmer’s Model</i>	- 162 -
6.6.2	<i>Instruction Set</i>	- 162 -
6.6.3	<i>Computer Instructions</i>	- 162 -
6.7	PROGRAMMING LANGUAGE LEVELS	- 162 -
6.7.1	<i>Machine Code</i>	- 162 -
6.7.2	<i>Assembly Language</i>	- 163 -
6.7.3	<i>Higher Level Languages</i>	- 163 -
6.8	THE DIGITAL DESIGN HIERARCHY	- 164 -
6.9	CHAPTER SUMMARY	- 166 -
6.10	CHAPTER EXERCISES	- 167 -
PART FOUR: RISC-V ASSEMBLY LANGUAGE PROGRAMMING		- 168 -
7	ASSEMBLY LANGUAGE INTRODUCTION	- 169 -
7.1	INTRODUCTION	- 169 -
7.2	BITS TO MNEMONICS AND BACK AGAIN	- 169 -
7.3	PROGRAMMING LANGUAGE LEVELS	- 170 -
7.3.1	<i>Machine Code</i>	- 170 -
7.3.2	<i>Assembly Language</i>	- 171 -
7.3.3	<i>Higher Level Languages</i>	- 171 -
7.4	ASSEMBLY LANGUAGES: THE GOODNESS OF “LOW-LEVEL”	- 171 -
7.5	PROBLEM SOLVING WITH PROGRAMMING	- 173 -
7.6	STRUCTURED PROGRAMMING	- 175 -
7.7	MOTIVATIONAL DISCUSSION OF FLOWCHARTING	- 176 -
7.7.1	<i>The Basics of Flowcharting</i>	- 177 -
7.8	STRUCTURED PROGRAMMING REVISITED.....	- 178 -
7.8.1	<i>The sequence Structure</i>	- 178 -
7.8.2	<i>The if-then-else Structure</i>	- 179 -
7.8.3	<i>The iterative Structure</i>	- 179 -
7.9	THE TRUTH ABOUT SOFTWARE	- 180 -
7.9.1	<i>Software Quality</i>	- 180 -
7.10	WRITING GOOD PROGRAMS	- 180 -
7.11	CHAPTER SUMMARY.....	- 183 -
7.12	CHAPTER EXERCISES	- 184 -
8	INTRODUCTION TO RISC-V ASSEMBLY LANGUAGE PROGRAMMING	- 186 -
8.1	INTRODUCTION	- 186 -

8.2	INSTRUCTION SET ARCHITECTURE DESIGN ISSUES	- 187 -
8.2.1	<i>Instruction Set Design</i>	- 187 -
8.3	ISA DRIVEN COMPUTER HARDWARE DESIGNS	- 187 -
8.4	RISC-V MCU ASSEMBLY LANGUAGE PROGRAM STRUCTURE	- 188 -
8.4.1	<i>The Assembly Language Program</i>	- 188 -
8.4.1.1	Comments	- 188 -
8.4.1.2	Assembler Directives	- 189 -
8.4.1.3	Assembly Code	- 189 -
8.4.1.4	Labels	- 189 -
8.4.2	<i>Important Assembly Language Program Formatting</i>	- 189 -
8.4.3	<i>The Actual Program</i>	- 190 -
8.4.4	<i>Visual Description of Program</i>	- 190 -
8.5	WHAT THE ISA REALLY DOES	- 191 -
8.6	RISC-V MCU ASSEMBLY LANGUAGE BASICS	- 192 -
8.6.1	<i>The Big Picture</i>	- 192 -
8.6.1.1	Program Control	- 193 -
8.6.1.2	Load & Store	- 194 -
8.6.1.3	Operations	- 194 -
8.6.1.4	Auxillary	- 194 -
8.7	INSTRUCTION TYPES	- 194 -
8.7.1	<i>Instruction Formats: High Level</i>	- 194 -
8.7.2	<i>Instruction Operand Addressing</i>	- 195 -
8.8	INSTRUCTION-RELATED TERMINOLOGY	- 195 -
8.8.1	<i>Changing Stored Values</i>	- 195 -
8.8.2	<i>Alternate Register Names</i>	- 196 -
8.8.3	<i>Source and Destination Designations</i>	- 196 -
8.9	EMBEDDED SYSTEMS PROGRAMMING	- 197 -
8.9.1	<i>Software vs. Firmware</i>	- 197 -
8.10	CHAPTER SUMMARY	- 199 -
8.11	CHAPTER EXERCISES	- 200 -
9	ASSEMBLY LANGUAGE PROGRAMMING OPERATIONS	- 201 -
9.1	INTRODUCTION	- 201 -
9.2	BASIC INSTRUCTIONS AND USAGE	- 201 -
9.2.1	<i>The First Data Transfer Instruction</i>	- 201 -
9.2.1.1	The <code>mv</code> Pseudoinstruction	- 202 -
9.2.2	<i>The Second Data Transfer Instruction</i>	- 204 -
9.2.3	<i>The First Data Crunching Instruction</i>	- 205 -
9.2.4	<i>Memory Related Data Transfer Instructions</i>	- 209 -
9.2.4.1	RISC-V Main Memory	- 209 -
9.2.4.2	Accessing Main Memory Data	- 210 -
9.3	INPUT/OUTPUT (I/O)	- 212 -
9.3.1	<i>RISC-V Memory Mapped I/O</i>	- 214 -
9.3.2	<i>RISC-V Input & Output Instructions</i>	- 215 -
9.3.3	<i>Load and Store: The Complete Story</i>	- 218 -
9.3.3.1	Load & Store Instructions Relation to I/O Data Widths	- 220 -
9.4	THE FIRST PROGRAM FLOW CONTROL INSTRUCTION	- 220 -
9.4.1	<i>Introduction to Program Flow Control</i>	- 221 -
9.5	CHAPTER SUMMARY	- 226 -
9.6	CHAPTER EXERCISES	- 227 -
9.7	CHAPTER PROGRAMMING PROBLEMS	- 228 -
10	INSTRUCTIONS, CONSTRUCTS, AND BIT-LEVEL MANIPULATIONS	- 229 -
10.1	INTRODUCTION	- 229 -

10.2	BIT CRUNCHING INSTRUCTIONS	- 229 -
10.2.1	Logic Instructions: AND, OR, & XOR	- 230 -
10.2.2	Arithmetic Instructions: Addition & Subtraction	- 233 -
10.2.3	Shift Instructions.....	- 234 -
10.3	AUXILIARY INSTRUCTIONS	- 236 -
10.3.1	Various Simple Pseudoinstructions Operation: the nop Instruction	- 236 -
10.3.1.1	Pseudoinstruction: nop.....	- 236 -
10.3.1.2	Pseudoinstruction: not.....	- 237 -
10.3.1.3	Pseudoinstruction: neg.....	- 238 -
10.3.2	xxxxSet If Less Than: slt, slti, sltu, sltiu	- 239 -
10.3.3	The Load Address Instruction: la.....	- 240 -
10.3.4	Other Loading-Type Instructions: auipc & lui.....	- 242 -
10.3.4.1	Add Upper Immediate to PC Instruction: auipc.....	- 242 -
10.3.4.2	Load Upper Immediate Instruction: lui	- 242 -
10.3.5	Loading Immediate Values: li	- 243 -
10.4	PROGRAM FLOW CONTROL.....	- 243 -
10.4.1	Labels Revisited	- 243 -
10.4.2	Branch Instructions.....	- 245 -
10.4.2.1	Unconditional Branch Instructions	- 245 -
10.4.2.2	Conditional Branch Instructions	- 248 -
10.5	STANDARD ASSEMBLY LANGUAGE CONSTRUCTS	- 249 -
10.5.1	If-Then-else Construct.....	- 249 -
10.5.1.1	Special if/else Coding Considerations	- 250 -
10.5.2	Iterative Constructs	- 253 -
10.5.2.1	while Loops.....	- 254 -
10.5.2.2	Do-While Loops	- 256 -
10.5.3	Iterative Construct Off-By-One Issues.....	- 257 -
10.6	BIT MANIPULATIONS FOR MCUs.....	- 265 -
10.6.1	Tweaking Bits	- 265 -
10.6.2	Bit Masking.....	- 265 -
10.7	CHAPTER SUMMARY.....	- 274 -
10.8	CHAPTER EXERCISES	- 275 -
10.9	CHAPTER PROGRAMMING EXERCISES	- 277 -
11	WORKING WITH MEMORY	- 279 -
11.1	INTRODUCTION.....	- 279 -
11.2	OVERVIEW.....	- 279 -
11.3	FLEXIBILITY IN INSTRUCTIONS.....	- 279 -
11.3.1	Register Addressing vs. Memory Addressing.....	- 280 -
11.3.1.1	Register Addressing	- 280 -
11.3.1.2	Main Memory Addressing	- 280 -
11.3.2	Assembly Language and Addressing Mode.....	- 281 -
11.4	MEMORY ACCESS: SOLVED PROBLEMS.....	- 281 -
11.5	CHAPTER SUMMARY.....	- 289 -
11.6	CHAPTER EXERCISES	- 290 -
11.7	CHAPTER PROGRAMMING PROBLEMS	- 291 -
12	SUBROUTINES AND SUPPORTING STRUCTURES.....	- 292 -
12.1	INTRODUCTION.....	- 292 -
12.2	SUBROUTINE SUPPORTING STRUCTURES: THE STACK	- 292 -
12.2.1	Pushing and Popping on the RISC-V MCU	- 294 -
12.3	SUBROUTINES OVERVIEW	- 296 -
12.4	SUBROUTINES ON THE RISC-V MCU.....	- 298 -
12.4.1	Calling Subroutines and Returning from Subroutines	- 300 -

12.4.2	<i>Passing Values to Subroutines</i>	- 302 -
12.4.3	<i>Saving Context in Subroutines</i>	- 304 -
12.4.4	<i>RISC-V and Nested Subroutines</i>	- 306 -
12.5	SPECIAL SUBROUTINE ISSUES.....	- 311 -
12.5.1	<i>Recursive Subroutines</i>	- 311 -
12.5.2	<i>Stack Overflow</i>	- 312 -
12.5.2.1	Subroutines and Stack Overflow.....	- 312 -
12.5.2.2	Context Saving and Stack Overflow.....	- 313 -
12.5.3	<i>Subroutine Overhead</i>	- 313 -
12.5.4	<i>Stack Initialization</i>	- 313 -
12.6	INTELLIGENT SUBROUTINE USAGE.....	- 314 -
12.7	CHAPTER SUMMARY.....	- 321 -
12.8	CHAPTER EXERCISES.....	- 322 -
12.9	CHAPTER PROGRAMMING PROBLEMS.....	- 324 -
13	RISC-V MCU INTERRUPT ARCHITECTURE (FIRMWARE)	- 326 -
13.1	INTRODUCTION.....	- 326 -
13.2	INTERRUPT OVERVIEW.....	- 326 -
13.3	THE THEORY OF INTERRUPTS.....	- 327 -
13.3.1	<i>Using Polling for Inputting Data</i>	- 328 -
13.3.2	<i>Moving Towards Real-Time Programming</i>	- 329 -
13.3.2.1	The Advantage of Real-Time Programming.....	- 330 -
13.4	RISC-V INTERRUPT ARCHITECTURE FOR PROGRAMMERS.....	- 330 -
13.4.1	<i>Real-Time Programmer Responsibilities</i>	- 331 -
13.4.1.1	Real-Time Program Structure.....	- 331 -
13.4.1.2	Interrupt Initialization.....	- 332 -
13.4.1.3	The Interrupt Service Routine.....	- 334 -
13.4.1.4	Saving the Context.....	- 334 -
13.4.1.5	Returning From ISRs.....	- 335 -
13.4.2	<i>Basic Interrupt Example Program</i>	- 335 -
13.4.3	<i>Real-Time Programming Considerations</i>	- 337 -
13.5	REAL-TIME PROGRAMMING EXAMPLE PROBLEMS.....	- 338 -
13.6	CHAPTER SUMMARY.....	- 345 -
13.7	CHAPTER EXERCISES.....	- 346 -
13.8	CHAPTER PROGRAMMING PROBLEMS.....	- 347 -
14	IMPORTANT SUPPORTING TOPICS	- 348 -
14.1	INTRODUCTION.....	- 348 -
14.2	MEMORY SEGMENTATION.....	- 348 -
14.2.1	<i>Memory Address Space</i>	- 349 -
14.2.2	<i>Code Segment</i>	- 349 -
14.2.3	<i>Data Segment</i>	- 350 -
14.2.4	<i>Stack</i>	- 350 -
14.2.5	<i>Memory Mapped I/O Segment</i>	- 350 -
14.3	THE RISC-V ASSEMBLERS.....	- 351 -
14.3.1	<i>Assembler Directives</i>	- 351 -
14.3.1.1	Instruction-Related Directives.....	- 352 -
14.3.1.2	Data-Type Directives.....	- 353 -
14.4	PROGRAMMING EFFICIENCY ISSUES.....	- 358 -
14.4.1	<i>Iterative Construct Overhead</i>	- 359 -
14.4.2	<i>Subroutine Overhead Issues</i>	- 361 -
14.4.3	<i>Program Space vs. Bullet-Proof Code Issues</i>	- 363 -
14.5	LOOK-UP TABLES (LUTS).....	- 366 -
14.5.1	<i>LUTs Revisited</i>	- 369 -

14.6	CHAPTER SUMMARY.....	- 370 -
14.7	CHAPTER EXERCISES	- 371 -
14.8	CHAPTER PROGRAMMING EXERCISES.....	- 375 -
15	RISC-V SOLVED PROGRAMMING PROBLEMS	- 376 -
15.1	INTRODUCTION.....	- 376 -
15.2	INTRODUCTORY RISC-V PROGRAMMING PROBLEMS.....	- 376 -
15.3	MORE ADVANCED RISC-V PROGRAMMING PROBLEMS	- 382 -
15.4	C CODE-BASED RISC-V PROGRAMMING PROBLEMS	- 427 -
15.5	CHAPTER SUMMARY.....	- 439 -
15.6	CHAPTER EXERCISES	- 440 -
15.7	CHAPTER PROGRAMMING PROBLEMS.....	- 441 -
	PART FIVE: RISC-V OTTER MCU HARDWARE MATTERS.....	- 443 -
16	RISC-V ARCHITECTURE DETAILS.....	- 444 -
16.1	INTRODUCTION.....	- 444 -
16.2	THE BIG RISC-V MCU OVERVIEW.....	- 444 -
16.3	THE CONTROL UNITS.....	- 445 -
16.3.1	<i>The Control Unit FSM (CU_FSM).....</i>	- 445 -
16.3.1.1	Individual FSM States	- 448 -
16.3.2	<i>The Control Unit Decoder.....</i>	- 450 -
16.4	THE PROGRAM COUNTER (PC) (NO INTERRUPT SUPPORT)	- 452 -
16.4.1	<i>PC Inputs and Outputs.....</i>	- 453 -
16.4.2	<i>PC Functionality.....</i>	- 453 -
16.4.3	<i>jal & jalr Instruction Details</i>	- 454 -
16.4.4	<i>Conditional Branch Instruction Details.....</i>	- 457 -
16.5	MAIN MEMORY	- 458 -
16.5.1	<i>Physical Memory</i>	- 460 -
16.5.1.1	Program Memory	- 460 -
16.5.1.2	Data Memory.....	- 461 -
16.5.2	<i>Input/Output Memory Space</i>	- 463 -
16.5.3	<i>Memory Timing Issues.....</i>	- 464 -
16.5.3.1	Branch Instruction Timing.....	- 466 -
16.5.3.2	Memory Access: Load-Type Instruction.....	- 467 -
16.5.3.3	Inputting Data: Load-Type Instruction.....	- 468 -
16.5.3.4	Memory Access: Store-Type Instructions	- 469 -
16.5.3.5	Outputting Data: Store-Type Instruction.....	- 471 -
16.6	THE IMMEDIATE VALUE GENERATOR (IMMED_GEN)	- 472 -
16.7	THE BRANCH ADDRESS GENERATOR (BRANCH_ADDR_GEN)	- 473 -
16.8	THE REGISTER FILE (REG_FILE).....	- 474 -
16.9	THE ARITHMETIC LOGIC UNIT (ALU)	- 475 -
16.9.1	<i>Addition and Subtraction.....</i>	- 477 -
16.9.2	<i>Shifting Instructions.....</i>	- 478 -
16.9.3	<i>Logic Instructions.....</i>	- 479 -
16.9.4	<i>Set-If-Less-Than Instructions</i>	- 480 -
16.10	THE BRANCH CONDITION GENERATOR (BRANCH_COND_GEN)	- 481 -
16.11	THEN CONTROL AND STATUS REGISTERS (CSR)	- 483 -
16.12	THE RISC-V MCU WRAPPER	- 483 -
16.12.1	<i>Wrapper External Device Addressing</i>	- 484 -
16.12.2	<i>Wrapper Input Circuitry.....</i>	- 484 -
16.12.3	<i>Wrapper Output Circuitry.....</i>	- 485 -
16.13	CHAPTER SUMMARY.....	- 489 -
16.14	CHAPTER EXERCISES	- 491 -

16.15	CHAPTER HDL (VERILOG) EXERCISES	- 495 -
17	RISC-V INSTRUCTION DETAILS	- 496 -
17.1	INTRODUCTION.....	- 496 -
17.2	HARDWARE-BASED STACK IMPLEMENTATIONS	- 496 -
17.3	INSTRUCTION TYPES AND FORMATS	- 497 -
17.3.1	<i>Field Codes and Opcodes</i>	- 497 -
17.4	NOTABLE HANDLING OF SPECIFIC INSTRUCTIONS.....	- 498 -
17.4.1	<i>Add Upper Immediate to PC Instruction: auipc</i>	- 498 -
17.4.2	<i>Load Upper Immediate Instruction: lui</i>	- 499 -
17.4.3	<i>Calling Subroutines: The call Pseudoinstruction</i>	- 500 -
17.4.3.1	Subroutine Call Timing.....	- 501 -
17.4.4	<i>Returning from Subroutines: The ret Pseudoinstruction</i>	- 503 -
17.4.4.1	Subroutine Return Timing	- 503 -
17.4.5	<i>Loading Immediate Value: li</i>	- 504 -
17.4.6	<i>Load Address Pseudoinstruction: la</i>	- 505 -
17.4.6.1	Assembler Handling of Labels.....	- 506 -
17.4.7	<i>Special Operations: the slt-Type Instructions</i>	- 507 -
17.5	CHAPTER SUMMARY.....	- 509 -
17.6	CHAPTER EXERCISES	- 510 -
18	RISC-V MCU INTERRUPT ARCHITECTURE (HARDWARE)	- 512 -
18.1	INTRODUCTION.....	- 512 -
18.2	RISC-V MCU INTERRUPT OVERVIEW	- 512 -
18.2.1	<i>The RISC-V Interrupt Input</i>	- 513 -
18.2.2	<i>The Interrupt Cycle</i>	- 513 -
18.3	INTERRUPT SUPPORT HARDWARE.....	- 514 -
18.3.1	<i>The Interrupt Masking Circuitry</i>	- 514 -
18.3.2	<i>The Control and Status Registers (CSRs)</i>	- 515 -
18.3.2.1	The mie Register	- 516 -
18.3.2.2	The mtvec Register	- 517 -
18.3.2.3	The mepc Register	- 517 -
18.4	INTERRUPTS AND PROGRAM FLOW CONTROL	- 517 -
18.4.1	<i>Interrupt Initialization</i>	- 517 -
18.4.2	<i>Acting on Interrupts</i>	- 518 -
18.4.2.1	Interrupt Cycle Timing	- 518 -
18.4.3	<i>Returning from Interrupt Processing</i>	- 519 -
18.4.3.1	Return From Interrupt Timing	- 520 -
18.5	OTHER RISC-V INTERRUPT-RELATED HARDWARE MODIFICATIONS	- 524 -
18.5.1	<i>Program Counter (PC) Support</i>	- 524 -
18.5.2	<i>Control Unit Support: FSM & DCDR</i>	- 525 -
18.6	INTERRUPT SIGNAL-RELATED TIMING ISSUES	- 526 -
18.6.1	<i>Interrupt Signal Noise</i>	- 527 -
18.6.2	<i>Interrupt Signal Duration</i>	- 527 -
18.7	INTERRUPT ARCHITECTURE SUMMARY	- 528 -
18.8	CHAPTER SUMMARY.....	- 529 -
18.9	CHAPTER EXERCISES	- 530 -
19	MISCELLANEOUS RISC-V MCU AND OTHER ARCHITECTURE DETAILS	- 531 -
19.1	INTRODUCTION.....	- 531 -
19.2	RISC VS. CISC ARCHITECTURE TYPES.....	- 531 -
19.3	STANDARD COMPUTER ARCHITECTURES	- 532 -
19.4	LEVELS OF MEMORY.....	- 533 -

19.5	SWITCH BOUNCE	- 534 -
19.6	MONOSTABLE MULTIVIBRATORS (ONE-SHOTS)	- 536 -
19.7	SEVEN-SEGMENT DISPLAY MULTIPLEXING	- 537 -
19.7.1	<i>Undesirable 7-Segment Display Effects</i>	- 539 -
19.7.2	<i>Lead-Zero Blanking</i>	- 539 -
19.8	CHAPTER SUMMARY	- 541 -
19.9	CHAPTER EXERCISES	- 542 -
20	RISC-V MCU TIMING ISSUES	- 544 -
20.1	INTRODUCTION	- 544 -
20.2	RISC-V OTTER MCU TIMING PROBLEMS	- 544 -
20.2.1	<i>Modeling Instructions Using Timing Diagrams</i>	- 545 -
20.3	CHAPTER SUMMARY	- 557 -
20.4	CHAPTER EXERCISES	- 558 -
21	RISC-V ARCHITECTURAL MODIFICATIONS	- 559 -
21.1	INTRODUCTION	- 559 -
21.2	RISC-V ARCHITECTURAL MODIFICATIONS AND EXTENSIONS	- 559 -
21.3	CHAPTER SUMMARY	- 575 -
21.4	CHAPTER EXERCISES	- 576 -
21.5	xxxxCHAPTER DESIGN PROBLEMS	- 577 -
APPENDIX	- 578 -
	FOUNDATION MODELING CHEATSHEET	- 579 -
	RISC-V OTTER MCU ARCHITECTURE DIAGRAM (NO INTERRUPTS)	- 580 -
	RISC-V OTTER MCU ARCHITECTURE DIAGRAM (WITH INTERRUPTS)	- 581 -
	FINITE STATE MACHINE MODELING USING VERILOG BEHAVIORAL MODELS	- 582 -
	RISC-V MCU WRAPPER SOURCE CODE	- 583 -
	VERILOG STYLE FILE	- 585 -
	RISC-V MCU ASSEMBLY LANGUAGE STYLE FILE	- 588 -
	GLOSSARY OF COMPUTER DESIGN TERMS	- 590 -
	INDEX	- 607 -

Pretentions

(Bryan Mealy 2016 ©)



Legal Crap

FreeRange Computer Design: The RISC-V MCU

Copyright © 2020 Bryan Mealy, aka James Mealy

Date: Jan 1, 2016

You can download a free electronic version of this book from:

my calpoly teacher website

The author has taken great care in the preparation of this book, but makes no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or models contained in this book.

This book is licensed under the Creative Commons Attribution-ShareAlike Un-ported License, which permits unrestricted use, distribution, adaptation, and re-production in any medium, provided the original work is properly cited. If you build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. To view a copy of this license, visit:

<http://creativecommons.org/licenses/by-sa/3.0/>

We are more than happy to consider your contribution in improving, extending, or correcting any part of this book. For any communication or feedback that you might have regarding the content of this book, feel free to contact the author at the following address:

bmealy@calpoly.edu



Acknowledgements

First, I'd want to thank Adobe for making their software available at no cost to academic types. I created easily over 99% of the images in this text using various versions of Adobe Illustrator. I find almost all software I use crashes more often than I like; Adobe software crashes about once every five years. Thanks Adobe; I don't know where I'd be without you. The entire Adobe suite of software is amazing beyond description; if you need to do something on the computer, be sure to check out Adobe products.

Second, I'd want to thank Lucidchart (www.lucidchart.com) for their amazing software. I created some of the flowcharts in this text using Adobe Illustrator, but that was like driving a Ferrari to the grocery store. The Lucidchart software made flowchart creation fast, easy, and with great results. The Lucidchart software does many amazing things, so be sure to take a look if you need some type of visual modeling of your task. A special thanks to Lucidcharts for providing extra storage to academic types at no cost.

Thanks to Larry Schwartz, who showed me that some people have the cards stacked against them from the get go.

Thanks to my good friends John and Mike; the view from your sets of eyes is both insightful and inspirational, and is something I always look forward to.

Thanks to Bruce for giving a stranger like me a chance, and for calling me everyday when my light was at its dimmest.

Thanks to Gary for demonstrating to me how to be brave, creative, honest, loyal, and humble.

Thanks to Richard, for your honesty, insight, and inspiration. Things really do come out in the WASH.

Thanks to Gilbert, who believed in me when though I could never believe in myself, and for not giving up on me.

Thanks to James for always being there for me when I'm down with no hope of getting back up.

Thanks to Duncan Applegarth for pointing out the many errors in this text; there are many less now.

Hey Dickson... Someday we'll work together on all the things we've yet completed. I'm looking forward to that day.

Rambling Commentary

My inspiration for writing free textbooks came from my own personal notion that knowledge, particularly technical knowledge, should not be held ransom by publishing companies, bookstores, book authors, and academic administrators. Students seeking knowledge are sitting ducks when it comes to the notion of structured learning situations such as colleges and universities. Being that students are the lowest hanging fruit, they always are the first to have their wallets lightened by various well-connected entities. I hope this book serves as an alternative to shelling out money for overpriced textbooks.

This book is going to have errors. Please accept my sincerest apologies for the errors you will come across. I did my best to remove errors, but there are a few reasons why some errors remain.

1. Writing and proofreading is very timing consuming.
2. Unlike several of my colleagues, I do not bribe students into proofreading my writing. There has been more than one instance of an instructor at the institution where I teach giving “extra credit” to anyone who reported errors in their writing. I do happily accept suggestions and corrections from students, but it is out of the student’s own desire to help on the project.
3. I generated every digital design problem in this book. Once again, unlike many of my colleagues, I did not “assign” students to generate problems for me. I believe instructors who force students to create problems as graded assignments are unethical and are taking advantage of their positions as instructors. It’s called an abuse of power; I simply won’t do that.

I could spend the remainder of my life tweaking this text, but I need to move onto other things. Feel free to contact me with corrections and comments.



There were two primary negative comments I received when I mentioned I was writing a textbook and was planning to give it away at no cost.

- “If you don’t charge something, people will not value it”. I don’t understand this statement. The things I value most in my life were given to me. Maybe I’m missing something here.
- “You need to have experts in your field review your text”. As a college teacher, I constantly receive requests from book companies to “review” one of their texts. They always sweeten the deal with an offer of cash. I know of no one who is going to dedicate any significant amount of their time to reading a text they care nothing about, but I know of people who pretend to review books, write down some drivel, and receive their cash. Wow! Great review! A book is a mechanism to transfer knowledge; it’s not a popularity contest, as are most things in academia.



As you read this book, you may get the impression that I don’t like academic administrators. The truth is that I do not like academic administrators. In addition to cleaning out the wallets of students, they seem to want to rip the hearts out of students and teachers alike. They lack ethics and morals (an understatement). They reward those who support their agendas and crap on everyone else. They make sure they are first at the feeding trough and leave little for anyone else. They run the school as a business and not as institutions of learning. They base all their decisions on economics; quality of education and the basic needs of students are never one of their considerations. Academic administrators seem to believe that students and teachers should be serving them; I believe we should all be serving students. Schools exist to help students learn; we should base all decision making on supporting student learning. Academic administrators have clearly lost sight of the basic tenets of education.

Finally, this text is what it is. The quality and coverage is the best I can do given the various constraints I face. I made the decision to embark on this project knowing that it was more than likely a career killer in the context of Cal Poly SLO. Well, no need to wonder anymore; it’s definitely a career killer. I don’t regret ensuring that I was a person who supported student learning rather than simply supporting my own personal

career aspirations. The academic environment in general encourages faculty to adopt characteristics of NPD (narcissistic personality disorder) in order for the admin to bestow the “successful” label. I continue to choose to not compromise my ethics and to never lose sight of my mission as an instructor.



“If you judge safety to be the paramount consideration in life you should never, under any circumstances, go on long hikes alone. Don’t take short hikes alone, either – or, for that matter, go anywhere alone. And avoid at all costs such foolhardy activities as driving, falling in love, or inhaling air that is almost certainly riddled with deadly germs.....And never, of course, explore the guts of an idea that seems as if it might threaten one of your more cherished beliefs. In your wisdom, you will probably live to be a ripe old age. But you may discover, just before you die, that you have been dead for a long, long time.” (Collin Fletcher, Complete Walker 3).



Overview of Chapter Overviews

This text presents introductory computer design and assembly language programming concepts. This book focuses on a single generic computer design: The RISC-V OTTER Microcontroller. There are definitely topics in computer architecture and assembly language programming that this book does not cover. The general idea behind this textbook is to support a college course that attempts to teach computer design and assembly language programming in a ten-week course. While teaching this amount of information in single 10-week course barely seems possible, this textbook and various supporting materials do their best to make it possible.

PART ONE: Introduction and Review

Part One of FreeRange Computer Design introduces the various aspects of computer design. This introduction includes a high-level overview of computers using terms and concepts that you would find in a typical digital design course. In addition, if you can't remember those terms and concepts, this part also provides a fast overview of the important digital concepts.

Chapter 1: This chapter includes an overview of the course, a brief history of the course, and history of the MCUs used in the course. *This chapter is important because it provides a context for this text by describing some the issues regarding computer design courses as well as other pertinent information.*

Chapter 2: This chapter provides a review of the basic building blocks (modules) of digital design. These building blocks include combinatorial and sequential circuits, as well as Finite State Machines (FSMs). *This chapter is important because it describes most of the important concepts from a typical beginning digital design course. In particular, this chapter provides a fast overview of the topics presented in FreeRange Digital Design Foundation Modeling.*

Chapter 3: This chapter introduces and a relatively complete coverage of two of the most common and useful type of registers: shift registers and counter. This chapter covers the many types of shift registers including barrel shifters, universal shift registers, etc. *This chapter is important because registers and their simple variations are extremely useful and thus often found in just about all meaningful digital designs.*

PART TWO: Advanced Digital Design

Part Two of FreeRange Computer Design introduces the various hardware aspects of computer design that typical digital design courses often do not present. We present these topics because they are critical to understanding the low-level aspects of subsystems typically found in computers.

Chapter 4: This chapter introduces Register Transfer Notation, which is highly useful in both designing and describing circuits. This chapter uses RTN to describe and design various classifications of data transfer circuitry commonly used in digital design. *This chapter is important because shift registers and counters are extremely useful in many areas of digital design, particularly in applications requiring fast arithmetic operations. These devices are simple registers with extended features.*

Chapter 5: This chapter provides an introduction many of the more common and important aspects of structured memory. This chapter primarily involves the high-level characteristics of structured memory and describes them primarily in general terms. This memory introduction includes items such as standard memory vernacular and basic performance characteristics. *This chapter is important because it provides a basic overview of digital memory and the operation of memory in a digital system.*

PART THREE: Introduction to Computers

Part Three of FreeRange Computer Design represents our first description of computers, which we do from a relatively high level. Even though computers are nothing more than complex digital circuits, they come with their own “processes” and vernacular. The intent of this section is to show the connection between digital circuits and the particular digital circuit we call a computer.

Chapter 6: This chapter introduces the notion computers in general and how humans use computers. This chapter places the notion of computer design in a digital design context, leveraging what you already know about digital design. This chapter also places the notion of digital design into the context of computer design using terms associated with a beginning digital design course. *This chapter is important because it provides a high-level overview of the computer design by placing computer design into a familiar context.*

PART FOUR: RISC-V Assembly Language Programming

Part Four of FreeRange Computer Design provides the knowledge you'll need to become a RISC-V assembly language programmer. We wrote this section of the book in such a way as to not include hardware details in the various programming topics in the chapter in an attempt to make this section of the text as useful as possible to people only interested in programming (and not hardware). The underlying hardware details appear in Part Five of the text. While separating the programming and hardware concepts of the RISC-V results in some repetition to those interested in both programming and hardware, we make all efforts to keep the repetition to a minimum.

Chapter 7: This chapter introduces programming using assembly language and writing assembly language programs. This chapter also provides an overview of structured programming concepts and an overview of basic flowcharting techniques as they apply to structured programming. *This chapter is important because it introduces assembly languages and associated concepts as well as basic program structure concepts.*

Chapter 8: This chapter provides a high-level introduction to assemble language programming including instruction set design and assembly language program structure, appearance, design and documentation. This chapter introduces important RISC-V assembly language program vernacular as well as a basic classification of instructions in the RISC-V instruction set. This chapter also provides a description of embedded systems as they relate to assembly language programming. *This chapter is important because it describes the basic structure of assembly language programs and provides several well-commented assembly language example programs.*

Chapter 9: This chapter introduces a basic set of RISC-V instructions with detailed explanations to enable the reader to write basic RISC-V programs. This chapter describes three types of instructions including I/O, data transfer, and number crunching instructions. A later chapter introduces a more complete set of RISC-V instructions. *This chapter is important because it represents an introduction to the RISC-V instruction set in such a way as to be able to write basic RISC-V programs.*

Chapter 10: This chapter introduces most of the remaining instructions in the RISC-V instruction set including logic-type, arithmetic-type, shift-type instructions, and a set of “auxiliary” instructions that have various purposes. This chapter also introduces the complete set of program flow control instructions, which we initially introduced in a previous chapter. This chapter also introduces bit manipulation techniques that support the notion of bit masking. *This chapter is important because it describes some of the basic programming concepts and approaches beyond simple description of individual instructions.*

Chapter 11: This chapter provides highlights into accessing various RISC-V memory modules in the context of the RISC-V instruction set. The instruction set provides efficient and generic access to main memory, which allows great flexibility for programmers using the RISC-V to solve problems. *This chapter is important because it shows the full flexibility and functionality of RISC-V memory-type instructions.*

Chapter 12: This chapter describes everything there is to know about writing and using subroutines in assembly language programs. Proper subroutine usage in assembly language programs is the foundation of good programming forms such as modular programs. *This chapter is important because it describes the details involved in the design and implementation of subroutines in assembly languages.*

Chapter 13: This chapter describes all the information that programmers should be aware regarding interrupts on the RISC-V MCU. We complete the hardware description of this interrupt architecture in a later chapter. *This chapter is important because it describes the RISC-V interrupt architecture from the standpoint of an assembly language programmer.*

Chapter 14: This chapter introduces supporting topics such as memory segmentation, the various RISC-V assemblers, and programming efficiency issues. *This chapter is important because it describes many important support topics associated with programming the RISC-V MCU.*

Chapter 15: This chapter provides many solved problems ranging in scope from introductory to quite challenging. These problems and their detailed explanation show every trick in the assembly language coding book. *This chapter is important because it shows how to solve a wide set of problems by writing RISC-V assembly language programs.*

PART FIVE: RISC-V MCU Architectural Details

Chapter 16: This chapter describes the various submodules of the RISC-V OTTER MCU and their relation to instruction execution. This chapter also describes interfacing the RISC-V OTTER MCU to external world items such as development boards. *This chapter is important because it describes the low-level architecture details of the RISC-V MCU and its interfacing to the outside world with particular attention to instruction execution.*

Chapter 17: This chapter is the first hardware-based chapter, which introduces the hardware details associated with instruction implementation that are beyond the skillset of pure programmers. *This chapter is important because it describes some of the low-level details regarding RISC-V instructions and instruction execution.*

Chapter 18: This chapter describes the hardware details associated the RISC-V OTTER MCU interrupt architecture. This description includes modifications to existing modules and low-level details of instruction implementation. *This chapter is important because it describes the low-level architecture details of the RISC-V MCU interrupt architecture.*

Chapter 19: This chapter describes several topics typically included with computer architecture topics but not really part of the RISC-V MCU description. These topics include RISC vs. CISC, standard computer architectures, levels of memory, and seven-segment display multiplexing. *This chapter is important because it describes some the non-architectural but still important details involving the RISC-V MCU.*

Chapter 20: This chapter introduces the timing characteristics associated with RISC-V MCU instruction execution. *This chapter is important because it provides important insights into RISC-V MCU instruction execution by the use of timing diagrams.*

Chapter 21: This chapter presents problems that involve modifying the existing RISC-V MCU and support tools such that they support new and/or extended operations and/or instructions. *This chapter is important because it advances your knowledge of the RISC-V MCU by outlining possible hardware architecture changes in response to stated design goals.*

The Appendix

The Appendix provides some useful and handy RISC-V MCU information as well as fast overviews of Verilog. These items include:

- Digital Design Foundation Modeling Cheatsheet
- RISC-V OTTER MCU Architectural Diagrams
- Finite State Machine Modeling using Verilog Behavioral Models
- Wrapper Model for Basys3 Development Board Interfacing
- Verilog Style File
- RISC-V MCU Assembly Language Style File

Glossary of Computer Design and Assembly Language Programming Terms

This item provides a list of important computer design terminology and their relatively brief definitions.

Index

This item provides fast locator for the more important terms and acronyms used throughout the text.

PART ONE: Introduction and Review

1 FreeRange Computer Design Overview

1.1 Introduction

The main purpose of this chapter is to put FreeRange Computer Design into a meaningful context. I'm hoping to give you this context in several ways: 1) by describing the outline of the various chapters in this text, 2) by describing some of the issues involved with "computer design" textbooks, 3) by providing you with a general overview of the course, and finally, 4) by providing a quick history of the course. While all of this is not killer useful information, having the proper context for your endeavors facilitates learning, which is never a bad thing.

Main Chapter Topics

- **DESCRIPTION OF CHAPTER FORMATS:** Each chapter has a similar format; this chapter describes the chapter format for FreeRange Computer Design.
- **OVERVIEW OF TEACHING COMPUTER DESIGN:** Computer Design means different things to different people; this chapter describes the relatively unique approach for FreeRange Computer Design.
- **TEXT AND COURSE HISTORY:** This text and course have had a relatively long history. This chapter mentions some of the finer points and acknowledges the people who did the work to make this course text happen and continually improve.

Why This Chapter is Important

This chapter is important because it provides a context for this text by describing some of the issues regarding computer design courses as well as other pertinent information.

1.2 Chapter Structure

Each chapter has useful features in order to help the reader organize and digest the material in the chapter. Each chapter generally has the following features, though some chapters have special formats of their own.

- **Introduction:** Quick motivating prose overview including a list of the main topics and the chapter and why that chapter is important in digital design
- **The Body of the Chapter:** In case you want the whole story (with example problems)
- **Chapter Summary:** The quick overview of chapter
- **Practice Problems:** Including both exercises and/or design problems for the reader's entertainment

1.3 FreeRange Computer Design Beginnings

This text presents a course in Computer Design and Assembly Language Programming. The original label for this course was CPE 229 (the lecture portion of the course) and CPE 269 (the laboratory portion of this course). We later changed the course delivery to a studio format, which also entailed a label switch to CPE 233. Cal Poly first taught the original CPE 229/269 course-set Fall 2003 quarter. The previous version of CPE 233 has developed considerably in the years it was taught (the RAT MCU), but it had several issues that made it rife for replacement. The new version of CPE 233 uses the OTTER MCU, or the OTR MCU, which is a

modern and better supported version of the course. Like all good courses, CPE 233 is under constant development and is well on its way to becoming a great course.

In the late 1990's, a bunch of old guys¹ sitting around a table dreamed up a change in the CSC, CPE, and EE curriculum that was perceived to improve the quality of education in the respective departments. The idea was to do-away with EE 319 (hardware-based finite state machines and advanced digital design) and CSC 215 (software-based 68000 assembly language programming) and compress those topics into a single course. In reality, both of these courses went relatively deep into their respective topics. As if this was not enough, they also decided to add an element of computer architecture to the course. The initial result was a highly specialized course that covered finite state machine design, basic computer architecture, and assembly language programming. We later removed the notion of finite state machine design and placed it into the beginning digital design course (CPE 133).

This change in curriculum created several problems, some of which we are still dealing with today. Here are the gory details and status of these problems:

1. The first problem is there is no existing book that is appropriate for the entire course. Unfortunately, this has resulted in a compromised learning experience for the students taking the class. The instructor that spearheaded the development of CPE 229/269 originally promised to provide teaching materials for the instructors teaching the course, but the materials provided were not only worthless, but an on-going joke² amongst the students taking the course. In truth, this instructor's primary focus was to use CPE 229/269 as a vehicle to write another useless textbook and subsequently force CPE 229/269 students to purchase the text. This instructor finally retired. He did finish the book, however; no surprise that no Cal Poly instructor ever used his book after his retirement, which stands as a testament to the overall quality of this instructor's product and professionalism.
2. The second problem that this curriculum changed caused was an overlap in topics and concepts taught in this course for CPE and CSC majors. This is an ongoing problem, but the form of the problem has mutated. The issue is that CPE students are required to take CPE 315, an architecture course offered by the CSC department. Many professors in the CPE program have complained that the overlap is bad for CPE students. The truth is that CSC department chose to no longer require their students to take CPE 133 and CPE 233. As a result, CSC students have little or no experience with actual digital hardware or hardware design in general. That being the case, it's a mystery to me how you can teach a junior-level computer architecture course (CPE 315) without having a clue about basic digital hardware. The reality is that CPE 233 exists in large part to support EE students and we've been able to fight off the notion of changing CPE 233 in order to support the shortcomings of another department's curriculum. Additionally, the different skill levels in CPE and EE students resulted in the creation of CPE 333, which represents a continuation of the CPE 133 and CPE 233 courses.
3. CPE students taking CPE 233 most likely have more programming experience than EE students based on the notion that CPE students take CPE 123-101-202-203, while EE students only take CPE 101. Although this is an issue, programming only comprises about 40% of this course. Moreover, the programming is low-level (assembly language) and is the first time in either the CPE or EE curriculum that students see the material.

1.4 Issues with “Modern Computer Design”

If you ask a hundred people to define the notion of a computer, you will surely receive a hundred different replies. As you know (or as you'll soon find out), if you search for a common and usable definition of a computer, you may only find a description at such a high level, that the definition is almost worthless.

¹ I got in a lot of trouble for writing this. It's true; the truth only hurts liars; there are a lot of hurt people in academia.

² As reported to me by countless students in this instructor's class. The joke continues due to 1) the politics in the EE Department and CPE program, and 2) the strong resistance to any type of change exhibited by a vocal minority of EE and CPE faculty. The only thing that allowed change to occur in this area was the creation of CPE 233; the studio-version of this course allows instructors to operate independently of each other, thus somewhat protecting individuals from the politics.

Moreover, if you pick up a book on computer design (and there are hundreds of them out there), each book typically takes a different approach to describing what a computer is and what a computer does (and this of course does not include the different programming languages these computer books describe). There was a time (maybe in the 1950's) where there was a definition of a computer that described actual computers more completely, but the ever-expanding field of computer science and computer technology quickly muddied that definition.

I've spent a significant amount of time trying to figure out how to arrange a book on computer design such that it makes the topic both relatively easy to grasp and somewhat interesting to work with along the way. The main problem is that it is hard to describe something until you know what it does, and you generally won't get a good feel for what it does until you do it, but you can't do it until you know what you're doing, but you don't really learn things until you do them... I'm thinking you get the picture. The result of this dilemma is what you are reading now. My basic solution to this dilemma is to divide FRCD into five sections.

In a perfect world, where authors write perfect books, each chapter in the book would lead nicely into the next chapter and no chapter would assume knowledge contained on a page after the current page you're reading. I've divided FRCD into five parts: 1) an introduction and basic digital design review, 2) advanced digital design topics, 3) introduction to computer design, 4) The RISC-V from a programmers's perspective, and, 5) the RISC-V from a hardware perspective. The intention of dividing the text in this manner was to ensure the text was useful to people who were only interested in programming the RISC-V using assembly language, which is why the programmer's portion of the book bypasses all hardware notions.

1.5 The RAT Microcontroller/Microcomputer

The first meaningful version of CPE 233 involved developing and programming the RAT MCU. Microcontroller (MCU) or RAT Microcomputer. There are many ways to introduce the concept of computer design; the RAT MCU version of this textbook took the approach of having you design the RAT MCU using VHDL models and synthesizing those modules onto a FPGA-based development board. There is nothing overly special about the RAT MCU, but there is some worthwhile history associated with it:

- The RAT MCU started out as a concept for a computer design course. The Cal Poly Electrical Engineering department was suddenly required to teach a computer design/assembly language course in a ten-week period, but there were no materials out there to support such a knowledge-impacted course. The first attempt at such a processor was the ESX MCU, which I designed in the summer of 2004. The ESX MCU was a great learning experience, but it never went anywhere due to some oversights with the design. The main problem with the ESX MCU was that I designed it to be a subset of the Atmel AVR line of MCUs, something that felt like a good idea at the time but turned out to be rather pointless and stupid.
- A really cool student (Kianoosh Salami) and I “designed” the RAT MCU in the summer of 2009. The design simply comprised of a minimal set of instructions (we'll discuss exactly what that means later) that the RAT MCU would support. Our main goal was to make the instruction set as small as possible but result in a meaningful, synthesizable, and useful computer. Note that we never designed any actual hardware: the design started out as simply an instruction set. Our inspiration for the RAT MCU design was partially driven by our experience using the PicoBlaze2 and PicoBlaze3 MCUs in an older version of the course. The PicoBlaze designs represent a working MCU defined with VHDL models. The PicoBlaze3 was an improvement over the PicoBlaze2 design; naturally, the RAT MCU is an improvement over the PicoBlaze3 design³
- In the Fall of 2010, I pitched the RAT MCU concept for a course to Jeff Gerfen. He apparently liked the concept enough because he agreed to use it for his CPE 233 course he would teach in the Winter 2011 quarter. At that point, I agreed to support his efforts by generating an assembler for the course. I wrote the assembler for the course in the Fall 2009 quarter based on the instruction set Kianoosh and I had previously designed. I worked with Jeff to refine the

³ The notion of improvement needs defining here. The PicoBlaze designs were highly optimized to create a fast MCU that synthesized into a small footprint. The PicoBlaze design did not use much VHDL behavioral modeling. The resulting model was great, but the models did not support actually understanding how a computer works.

assembler and add some features during the Fall 2010 and Winter 2011 quarters. Included with this was the “RAT Assembler Manual”, which describes the instruction set and the various features contained in the assembler. In reality, Jeff made the course happen. He started from on the course based on an instruction set and the promise of an assembler and assembler manual. He thus did all the major initial development work for the course, which is a significant feat that you cannot overstate⁴. When the course was first taught in Winter 2011, two students created the RATSim, which serves as a simulator and debugger for the RAT MCU.

- Other instructors including Bridget Benson, Kari Haworth, Jeff Gerfen, and myself further developed the CPE 233 course. These changes underscore the good things that can be done when instructors work together and share their work, something that was unheard of in the days when old-fart wankers ruled the digital area of the EE and CPE departments with an iron-fist; dark days indeed.⁵

1.6 The RISC-V OTTER MCU

The RAT MCU started out as a great idea, but became tired over time. The main issue as I see it is that it was great at first, but then students became more smart and savvy. So as students progressed with their knowledge and ability to learn, the limitations in the RAT MCU became painfully obvious. The result is that students taking CPE 233 were no longer getting an optimal experience.

Joseph Callenes-Sloan was a recent addition to the Computer Engineering part of the Electrical Engineering department. It was with his wisdom, knowledge and basic hard work that he developed the RISC-V OTTER MCU. The OTTER MCU solved some of known issues of the RAT MCU; we list a few of those issues below.

- The RAT MCU and associated tools was a homegrown product, which meant any modifications required a significant amount of time and effort. We had to delay many meaningful modifications, sometime indefinitely, based on time constraints.
- The RAT MCU programs were limited to 1024 instructions. This was initially not an issue, but later became a big issue when student skills increased and many students found this program size limiting.
- The RAT MCU had no associated C compiler. As most of us know and are willing to admit, C is the programming language of hardware. The best approach to becoming a great hardware programmer is to be able to create programs at both the assembly language and higher-level language levels.

The OTTER MCU is not without issues. The main issue is the lack of documentation aimed at the students who know nothing or very little about computer architecture. Here’s the best analogy I can think of... The RAT is like an old Volkswagen bug: simple and dependable, but severely limited. It gets you where you need to go, but you won’t be travelling in style. The OTTER is like a Porsche: it also gets you where you need to go, but it gets you there much faster and in a more comfortable manner. If you know how to drive, you can drive either; but with the OTTER MCU, you’ll be able to effortlessly transition to other modern processors.

One of the really good things about the RISC-V is that it is a modern and known computer. When you list the RISC-V on your resume, people will know what it is and have some idea of your skills and abilities. When students listed the RAT MCU on their resume, prospective employers probably thought you had a rodent problem.

⁴ For those people who don’t know what is involved in developing a course, here are some of the issues. In a perfect world, the administration would provide instructors with as much time as the required in order to present a “good” course. In reality, the administration provides no time at all for instructors to develop courses. Instructors are not required to develop courses, but good instructors don’t feel comfortable presenting bad courses. A bad course in this instance is one that does not present the course materials in an coherent and modern manner, which is an ongoing problem in fast-moving technologies such as digital electronics.

⁵ And the worst part of it all was that all of the buttheads who were trying to control the course had never taught the course and had no intention of ever teaching the course. Why is it that people become corrupt when they find themselves in a position of power?

1.7 Issues with the CPE 233 Course

The CPE 233 course has one main issue: it's nearly impossible (if not impossible) to present a meaningful amount of material in a manner that support mastery of the topics in the time provided. The title of the CPE 233 course is "Computer Design and Assembly Language Programming". The main problem is that either of two subjects could easily be a course on their own. The problem is that we try to stuff all this material into one ten-week course, which turns out to be a questionable approach. It's a mystery why this problem has never been adequately addresses.

1.7.1 The CPE 233 Approach

The approach we take in CPE 233 is somewhat unique and a little bit questionable. Because of the time constraints, we provide students with the architecture, meaning they don't actually design a computer. We thus replace designing a computer with "*here's a computer; you must understand all aspects of this computer*" with the hopes that if you understand the provided design, you'll later be able to design your own computer from scratch. This is not an optimal approach, but it's much better than nothing.

Someday, someone will fix this course. I envision this course as being true computer design in that you must design a computer to solve a specific problem. Instead of "here's a computer", the course will be: "Here's a problem; design a computer to solve this problem". I'm confident this will happen; it's only a matter of time.

1.8 Chapter Summary

- **Course History:** This textbook was originally designed to support a new course in the Electrical Engineering Department. This new course started out as CPE 229/CPE 269, but later morphed into CPE 233. This course replaced a course in advanced digital design and assembly language programming. This course and textbook is under constant development.
 - **Original Course Conception:** This course and subsequent support materials was originally conceived of by Kianoosh Salami and Bryan Mealy. The aim of the design was to have a computer with a small instruction set have it be large enough to be both useful, versatile and facilitate the understanding of low-level computer design and basic assembly language programming concepts.
 - **The RISC-V OTTER MCU:** The MCU initially developed/implemented by Joseph Callenes-Sloan that replaced the RAT MCU for CPE 233 courses.
 - **Course Progress:** This course is a result of several instructors working together and sharing their work. This sort of collaboration is not typically found in academic environments due to the administration's underlying approach of judging instructor's performance on something other than an absolute standard, which results in instructor's having being reluctant to share their work. In this scenario, students always lose.
-

1.9 Chapter Exercises

- 1) Briefly explain why is there no great definition of a computer that satisfies everyone who may be asking such a question.
 - 2) Briefly describe why it is hard to define the notion of a computer.
 - 3) Briefly explain why there is no good off-the-shelf textbook for this course material.
 - 4) Briefly describe the five parts of this textbook.
 - 5) Briefly explain why this text divided into five parts.
 - 6) Briefly comment on where the name RAT came from.
 - 7) Briefly describe the main problem with the ESX MCU.
 - 8) Briefly describe how the RAT MCU started out.
 - 9) Briefly describe how the RISC-V OTTER MCU was created.
 - 10) Briefly describe some of the less than good issues associated with the CPE 233 course.
 - 11) Briefly describe some of the less than good issues associated with the RISC-V OTTER.
-

2 Digital Design Review

2.1 Introduction

The first course in digital design typically entails learning a standard set of combinatorial and sequential circuits, which we refer to as Foundation Modules. Despite the fact that this set of circuits is relatively small, you can design any possible digital circuit using them. We keep referring to these basic circuits as being in our “digital bag of tricks”, which means we know what these do, how they do it, and easily use them as the building blocks in any digital circuit. Moreover, we generally understand these basic digital building blocks at a high-level; we know how they operate at a low-level so we avoid designing at that low level and opt to abstract upwards and design at the modular level.

This chapter provides a quick overview of digital design including combinatorial circuits, sequential circuits, and Finite State Machine design. For a complete overview of these topics, please consult an appropriate digital design text such as FreeRange Digital Design Foundation Modeling; this text is available at: www.unconditionalllearning.com.

Main Chapter Topics

- **OVERVIEW OF IMPORTANT DIGITAL VERNACULAR:** This chapter lists and defines some of the more important terms and concepts related to introductory digital design.
- **COMBINATORIAL CIRCUIT REVIEW:** This chapter describes the basic combinatorial circuits that everyone should be familiar with including basic gates, half adders, full adders, ripple carry adders, multiplexors, decoders, comparators and parity circuits.
- **SEQUENTIAL CIRCUIT OVERVIEW:** This chapter describes the basic sequential circuits everyone should be familiar with including flip-flops and all the major aspects of Finite State Machines (FSMs).
- **APPROACHES TO DIGITAL DESIGN:** This chapter describes the three basic approaches to digital design: 1) brute force design, 2) iterative modular design, and 3) modular design.
- **DIGITAL DESIGN HIERARCHY:** This chapter provides a reminder of the path you’ve taken to arrive at the point of designing a computer.

Why This Chapter is Important

This chapter is important because it describes most of the important concepts from a typical beginning digital design course. In particular, this chapter provides a fast overview of the topics presented in FreeRange Digital Design Foundation Modeling.

2.2 The Design Process

If doing digital design was like following a recipe, everyone would be doing it and, employers would not be paying you the big bucks for you to do it. There is quite a bit of literature out there detailing the design process from many different angles; not all of it is exciting reading. The truth is that “design” is a process, which is sort of like a journey without needs to go anywhere. You know where you need to go, you know the tools at hand to get you there, and you dive in and get going. The good news is that the starting point in digital design is not

always at the beginning, which means you're absolutely expected to borrow from existing designs as part of the design process. The other good news is that the more design you do, the better you get at it.

Black box modeling is the mainstay of digital design. Accordingly, two of the most basic and important digital design principles (modular design and hierarchical design) deal directly with black box modeling. Here are the first four laws of digital design as they appear in Digital Design Foundation Modeling.

Mealy's First Law of Digital Design: If in doubt, draw some black box diagrams.

Mealy's Second Law of Digital Design: If your digital design is running into weird obstacles that require kludgy solutions, toss out the design and start over from square one.

Mealy's Third Law of Digital Design: Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.

Mealy's Fourth Law of Digital Design: The digital design process is circular, not linear. If you think you're going to generate the correct solution with the first pass, you're bound for disappointment. The digital design process is circuit; always make going backwards a few steps to fix issues part of the design process. Don't try to make your design perfect from the get-go, make it simple to understand so that you can fix issues as they arise.

Digital design is not a process where you can simply find the correct formula and plug in the correct values¹. The path to a solution (notice I did not say "the" solution) is rarely clear from the start, but the path becomes more familiar as you work on the problem. Working on the problem entails understanding all levels of the problem. The point here is that by the time you finish your design, you'll understand all levels of the problem. When you first start the problem, the path to the solution may not be (is rarely) clear. You will make mistakes along the way to solving your problem: we expect this in all designs. The key here is to make a mistake, learn from that mistake such that the path to the final solution becomes clearer. The design process is thus a constant learning process, it's not a grunt work thing you can do by rote. If you fall, pick yourself up and keep going. Good digital designers are people who know they are going to make mistakes, but have the wherewithal to quickly correct their issues.

A significant portion of this book is about assembly language program, including program design and understanding how the computer hardware implements instructions. The design process is similar for programs. First, you'll probably never get your program correct the first time you write it. Second, you learn to use the available tools (simulators, debuggers, etc.) to help you in the design process. You'll for sure develop your own style with the goal of writing good programs in a timely and efficient manner.

Lastly, I found this quote somewhere that I feel is perfect for digital design, programming, and any other non-trivial task you may embark on.

"One day when I was studying with Schoenberg, he pointed out the eraser on his pencil and said, 'This end is more important than the other.'"

-- John Cage, *Silence*

2.3 The New Digital Paradigm: Digital Design Foundation Modeling

We base our digital design knowledge on the *Digital Design Foundation Modeling* approach, or *DDFM*. This approach builds upon both *modular design* and *hierarchical design*, which are the main tenets of modern digital design. DDFM focuses on presenting digital design topics in the context of actual digital designs. The underlying goals of DDFM are to simplify the presentation of introductory digital design, and to provide a simple circuit model that describes all levels of digital design.

2.3.1 DDFM Overview

¹ This would be a good description of analog design

This section provides the high-level details about DDFM. The focus of DDFM is to present digital design in a simple and organized manner, which expedites learning the subject matter. These are the main tenets of DDFM:

- The main purpose of digital design is to solve problems using digital circuits
- We can best describe digital circuits in a modular and hierarchical manner
- Digital circuits are a set of digital modules that exchange information under the control of some entity
- We perform digital circuit design in a *structured*² manner, meaning that we can model *any* digital circuit using a relatively small subset of digital modules, which we refer to as the *digital design foundation modules*. Each foundation module performs a relatively small set of simple operations.
- We present the digital design foundation modules at a high-level by modeling the modules in terms of their data, control, and status signals, which allows us to use the modules in designs, while not requiring us to initially understand underlying implementation details.
- We classify the digital design foundation modules as either “controlled” or “controller” circuits
- We consider there to be four approaches to controlling a digital circuit:
 - 1) **NO CONTROL** (no flexibility in circuit behavior)
 - 2) **INTERNAL CONTROL** (controlling circuits using internal signals)
 - 3) **EXTERNAL CONTROL** (controlling circuits with devices such as buttons, switches, etc.)
 - 4) **CIRCUIT CONTROL** (controlling circuits using FSM or computer)
- We categorize digital design approaches into three categories:
 - 1) **BRUTE FORCE DESIGN (BFD)**
 - 2) **ITERATIVE MODULAR DESIGN (IMD)**
 - 3) **MODULAR DESIGN (MD)**

Figure 2.1 shows a high level generalization of a digital circuit. This figure provides a visual representation of a the digital circuit model we work with in this text. Figure 2.1 shows a circuit with inputs and outputs; the interior of the circuit contains combinatorial modules (cloud-shaped items) and sequential modules (square-shaped modules). The inputs to the circuit can be either data or status signal; the outputs of the circuit can be either data or control signals. The interior modules of the circuit communicate with each other using data, status, and/or control signals.

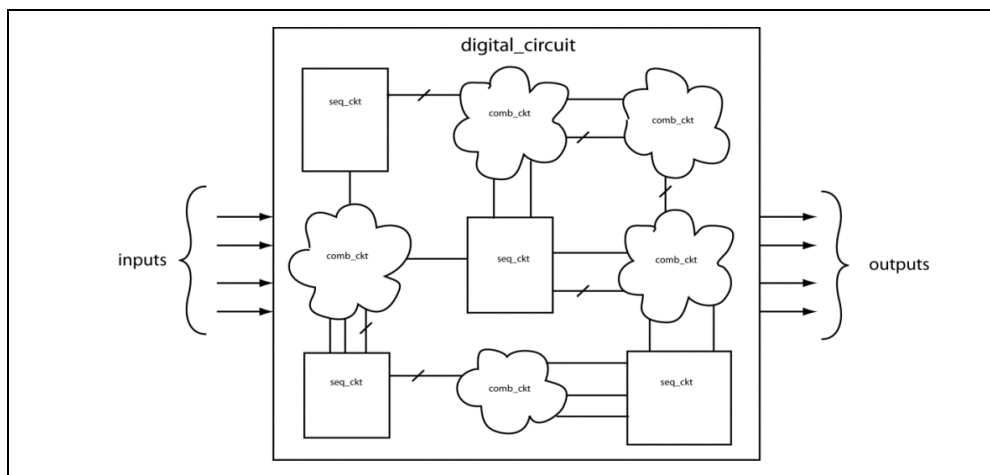


Figure 2.1: A generic digital circuit containing a set of digital modules.

² This is an analogy to structured computer program design

Figure 2.2(a) shows the standard approach to modeling digital circuits, where we classify all digital circuit signals as either inputs or outputs. Figure 2.2(b) and Figure 2.2(c) shows how DDFM further classifies inputs and outputs by first separating digital modules into “controlled circuits” and “controller circuits”. Figure 2.2(b) shows that we further classify the inputs to controlled circuits as either “data” or “control” and classify the outputs of controlled circuits as either “data” or “status”. This means the various circuit elements in Figure 2.2(b) are able to 1) pass data from their data inputs to their data outputs under the direction of the “control” inputs, and, 2) describe characteristics of the data transfers using the status outputs. Similarly, the status outputs of the controlled circuit form the status inputs of the controller circuit. The controller circuit of Figure 2.2(c) inputs the status signals of controlled circuits and manages the controlled circuits by outputting the appropriate control signals to control the controlled circuits³.

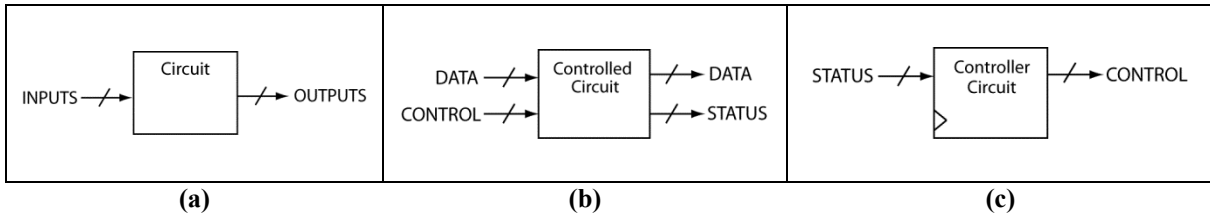


Figure 2.2: The old digital circuit model (a); models for controlled (b) and controller circuits (c).

The DDFM paradigm allows us to model all digital circuits as a controller that controls a set of modules. We then consider the solution to any digital design problem as a matter of using a controller to properly control the dataflow through a set of controllable modules. Figure 2.3 shows an example of many circuit modules controlled by a controller circuit; the controller circuit is either a finite state machine (FSM) or some type of computer control, such as a microcontroller. Figure 2.3 includes three different module shapes showing that controllable modules can either be combinatorial or sequential circuits, as well as off-the-shelf computer peripherals.

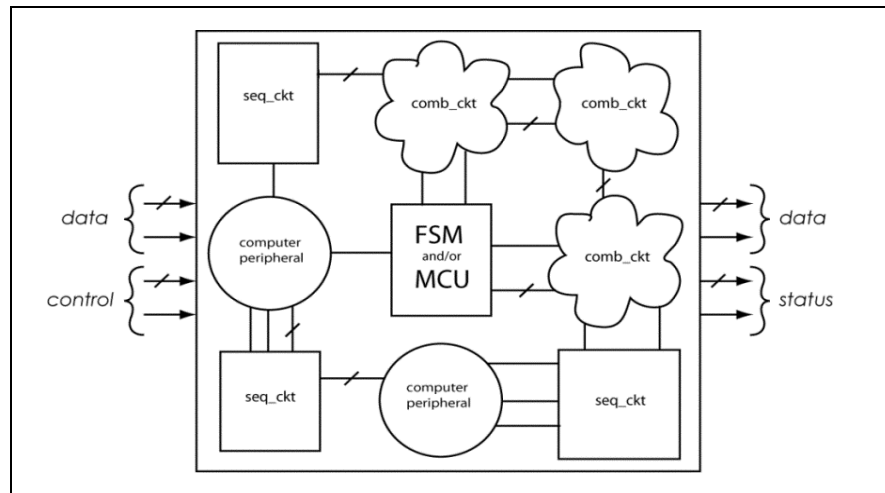


Figure 2.3: Our unifying digital circuit model.

2.3.2 The Three Approaches to Digital Design

Part of DDFM includes categorizing digital design into three different approaches. With some combination of these three approaches, you can create any digital circuit. Table 2.1 below shows the pros and cons of these three approaches.

BRUTE FORCE DESIGN (BFD): Our first approach to digital design. Although simple, its simplicity limits its practicality in non-trivial designs.

³ We purposely omitted data signal from the controller circuit. Controller circuits can have data inputs, but we generally try limit controller circuit inputs and outputs to only status inputs and control outputs.

ITERATIVE MODULAR DESIGN (IMD): Our second approach to digital design. Although IMD removes some of the limitations of BFD, it is only applicable to a few of circuits.

MODULAR DESIGN (MD): Our final and most powerful approach to digital design, and is thus where this text expends most of its effort.

Design Approach	Pros	Cons
Brute Force Design (BFD)	Really straight forward	Limited by truth table size
Iterative Modular Design (IMD)	Straight forward	Not applicable to all designs
Modular Design (MD)	Massively powerful	Requires a working brain

Table 2.1: Matrix explaining why Modular Design can save the world.

2.3.3 Notes on Modular Design Techniques

The general approach to becoming an efficient digital designer is to design on as high of level as possible. In terms of the three design techniques, that means you should always aim for the modular design approach, which necessarily incorporates all of your previously designed digital modules.

One of the underlying themes in digital design is the use of modularity, thus modern digital design consists primarily of Modular Design. To put this statement in other terms, you can subdivide even the most complex digital circuit into a set of the relatively few standard digital circuits. You do modular design by plopping down black boxes and connecting them (thus forming BBDs) in intelligent ways that solve your given problem. The black box diagrams are of course a form of modeling, which convey various levels of information regarding the digital circuit. Here are a few rules you need to follow when doing modular design.

- **Be clear and concise:** A messy dark box model or circuit diagram is a tragedy that hinders the transfer of information. Strongly consider using a ruler if you're modeling by hand.
- **Label everything:** Make sure the reader of your model does not need to make any assumptions about anything.
- **Provide a definition for all black boxes:** Black box modeling facilitates the notion of modern digital design. Every box you use in your model should either be clearly defined somewhere (such as at another level) or be a standard digital "box". There are many standard digital "boxes" out there. If you call out one of these boxes in your black box models, everyone knows what you're talking about and there is no need to define it at a lower level. The catch here is that you must use these boxes in the exact way there were defined; if you don't, people will not know what you're trying to model. Table 2.2 shows a few examples of proper black box usage.

Model	Comment
	This sort of looks like a 3-input OR gate, but having two outputs makes it non-standard. Being non-standard, it's a mystery how the circuit assigns the outputs. This is a bad model. To make it valid would require that it be defined somewhere so we all know what it is.
	This is a true digital box. Since we know what an RCA is, and the inputs and outputs of the box labeled RCA match what we know about RCAs, we know exactly how it works. This is a valid model and there is no need to define it further.
	This is also a true digital box. If you replace the HA in a RCA with a FA, you'll have the extra carry-in input as is listed in this model. Having this input is handy and often useful. This is a valid model.
	This circuit has the RCA label, but since we know RCAs to have multiple inputs (bundles) for the addend and augend, we're left scratching our heads. You could assume it's a RCA but you could be wrong. This is an invalid model.
	This has all the correct inputs for an RCA, but since it has the ADDER label, we can't assume we know exactly what this box is doing. This is an invalid model. You could make this model valid by providing a definition for the ADDER somewhere in your design.

Table 2.2: Some good and bad example of standard digital dark boxes.

2.4 Important Digital Vocabulary

If you only remember a few things from introductory digital design, you should remember the items in this section. These items probably won't help you pass any specific course, but they may help you pass in interview because even a substandard HR person can gauge whether you know these items or whether you're a sack of dead chi. As for vocabulary, there are 25 pages of vocabulary in the glossary of the FreeRange Digital Design textbook, consider browsing that stuff if none of these terms make sense.

Functionally Complete: This refers to the fact that some logic gates have the ability to implement all basic logic functions while others do not. NAND & NOR gates are functionally complete for example, because a NAND gate can be used to implement an AND, OR, or an inversion function. This is not true for AND & OR gates so they are not considered functionally complete.

Combinatorial vs. Sequential Circuits: The rough explanation is that sequential circuits contain memory while combinatorial circuits do not. In other words, a sequential circuit has the ability to "remember" at least one bit while combinatorial circuits do not. The better and longer explanation is that outputs of combinatorial circuits are a strict function of the circuit's inputs while in a sequential circuit, the outputs are a function of the sequence of the circuit's inputs. We derive the notion of a finite state machine (FSM) from this previous definition. Sequential circuits have at least one feedback path in them, which is the characteristic that gives them the notion of memory.

Mealy vs. Moore FSMs: In a Moore-type FSM, the circuit outputs are only a function of the state variables. In a Mealy-type FSM, the outputs are a function of both the state variables and the external inputs to the circuit.

Set-up and Hold-times: Generally speaking, in edge-sensitive devices, the non-clocking inputs to a device must be stable (non-changing) for a given period of time both before and after the active clock edge. The setup time refers to the time the inputs must be stable before the active clock edge while the

hold-time refers to the time the inputs must be stable after the active clock edge. If you violate setup and/or hold times, the circuit will probably not work because the circuit will be “metastable”. Metastability generally refers to the characteristic of the devices output as being neither high nor low and... stuck in the netherworld.

Latches vs. Flip-flops: Both latches and flip-flops are 1-bit storage elements. The difference is that flip-flops are “edge sensitive” latches, meaning that the flip-flops outputs can only change on an active clock edge. The latch is level-sensitive device, meaning roughly that the outputs can change anytime.

The first task at hand in your introductory digital design course was to learn the basics of digital design. This included the basic logic functions such as AND & OR, but was more specifically designed towards the gates that implemented these functions. The circuits you initially designed were primarily gate-level, which you abstracted up from the transistor level. The next part of the course used those logic gates to build the digital design foundation modules such as multiplexors, decoders, RCAs, etc. These are all considered combinatorial circuits. The next part of the course introduced sequential circuits with the introduction of memory elements such as latches and flip-flops. The main use of sequential circuits in the introductory course was register, which of course included two types of “registers with features”: counters and shift registers. Another way of looking at what we did was that we kept abstracting our circuit models upwards, which allowed us to model circuits at the modular level.

Table 2.3 uses the term modeling in a context that you’re somewhat used to hearing. This table provides an overview to the circuits you used in your introductory digital design course, as well as a reference as to how you implemented them. In this course, we’ll continue our abstraction of circuits upwards, which requires that we use new techniques to represent those circuits as they necessarily become more complex. Our new tool is register transfer language (RTL), which we mention in Table 2.3 but define in a later chapter.

The focus of this course is to develop a relatively complex digital circuit commonly referred to as a computer. There are many approaches to designing computers; this text describes one way in relatively great detail. With the knowledge you gain implementing this computer, you’ll be able to quickly understand the operation of other computer architectures, as they are nothing other than complex digital circuits.

Course	Design Focus	Circuit Models	Circuit Implementations
Intro Digital Design	basic logic: gates, circuit minimization	BBDs	FPGA
	combinatorial circuits: decoders, MUXes, adders, parity generators	BBDs, HDL	FPGA
	sequential circuits: latches, flip-flops, registers, counters, shift registers, FSMs	BBDs, HDL	FPGA
Computer Design	FSMs, counters, registers	BBDs, HDL	FPGA
	computer architecture	BBDs, HDL, RTL level	FPGA
	assembly language programming, microcontroller	BBDs, (programmers model), RTL level, HDL	FPGA

Table 2.3: Models and circuit implementation for CPE 133 and CPE 233.

In that we all aspire to be great digital designers, we want to be able to generate digital designs as efficiently as possible. While we could implement all of our designs at the gate-level, this would not be efficient. A better approach would be to implement designs at the “block level” or “object-level”, or what we refer to as “modular design”. The general theme of this design approach is to use “black-box” models of known circuit elements (such those listed in Table 2.3 or Figure 2.4) to model digital circuits at a relatively high level. This type of design is extremely efficient because so nicely supports two important concepts in digital design-land: 1) the concept of hierarchical design, and, 2) the ease at which we can use an HDL to implement modular designs.

The presence of large design libraries full of digital devices waiting for use by crafty digital designers fully supports the notion of modular design using HDLs. As you know, the reality in digital design land is that there are only a relatively few number of core digital devices out there (digital design foundation modules); you can use these modules to implement any digital circuit as a set of these core digital devices. This decomposition is a reversing of the hierarchical design process. If you are able to understand the operation of the digital design foundation modules, you'll also be able to understand any digital device, regardless of its complexity.

Figure 2.4 shows a quick overview of digital design as it relates to introductory digital design. What you should see from Figure 2.4 is that there aren't that many standard digital devices (or modules) out there and the ones that are out there, are relatively simple devices. Digital circuits become complicated only after you toss down a bunch of these modules into a design; hierarchical design mitigates this complexity. Note that most of the modules referenced in Figure 2.4 are Foundation Modules.

In summary, here's all I know about digital design:

- 1) Digital design is based on a relatively small set of digital devices
- 2) Digital design relies heavily on various modeling approaches, particularly modular-level design
- 3) Digital design modeling relies heavily on hierarchical modeling

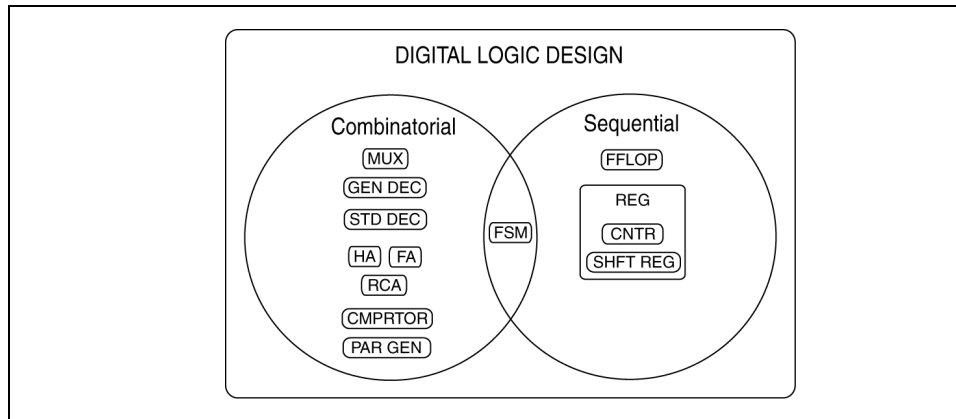


Figure 2.4: The quick digital design overview (most of which was covered in CPE 133).

2.5 Basic Gates

A gate is a hardware device that implements basic logic functions. We use transistors to implement gates, but transistors are too low level of abstraction for the needs of this text. Though we try to implement our circuits at the highest level possible (or reasonable for a given problem), we sometimes need to drop down to the gate-level in order to implement our designs. An inverter is not really a gate so we do not list it here. Figure 2.5 shows the list of basic gates and their various forms.

Gate	Description	Func Comp	Comments
AND	Output = '1' when all inputs are '1'; otherwise output = '0'	no	<ul style="list-style-type: none"> • has two or more inputs • has AND and OR forms • aka: logic multiplication
OR	Output = '0' when all inputs are '0'; otherwise output = '1'	no	<ul style="list-style-type: none"> • has two or more inputs • has OR and AND forms • aka: logical addition
NAND	Output = '0' when all inputs are '1'; otherwise output = '1'	yes	<ul style="list-style-type: none"> • has two or more inputs • has AND and OR forms
NOR	Output = '1' when all inputs are '0'; otherwise output = '1'	yes	<ul style="list-style-type: none"> • has two or more inputs • has OR and AND forms
XOR	Output = '1' when all inputs are not equal; otherwise output = '0'	no	<ul style="list-style-type: none"> • has two inputs only
XNOR	Output = '1' when all inputs are equal; otherwise output = '0'	no	<ul style="list-style-type: none"> • has two inputs only • aka: <i>equivalence gate</i>

Table 2.4: Summary of digital design gates.

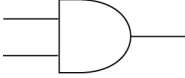
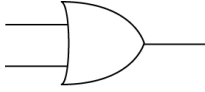
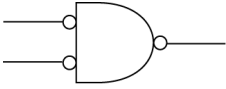
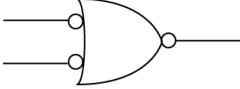
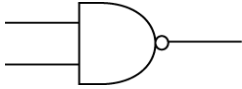
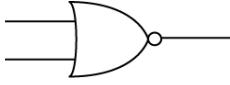
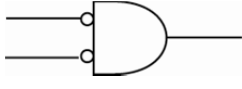
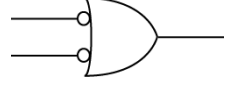
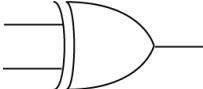
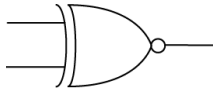
Standard Gates	
	
AND form of AND gate	OR form of OR gate
	
OR form of AND gate	AND form of OR gate
	
AND form of NAND gate	OR form of NOR gate
	
AND form of NOR gate	OR form of NAND gate
	
XOR gate	XNOR gate

Figure 2.5: The giant summary of logic gates.

2.6 Combinatorial Circuits

Combinatorial circuits are one of the two types of circuits in digital design. The outputs of combinatorial circuits are a function of the circuit's inputs. The following sections list the well-known digital circuits along with a brief description; we list most of the modules as digital design foundation modules. Please consult the appropriate text for full explanations of these circuits.

2.6.1 Half Adder

The Half Adder (HA) is generally the first somewhat meaningful in digital design. The HA is a one-bit adder (adds two one-bit values) and outputs a one-bit *sum* and a *carry-out*. We generally design the HA using a truth table (brute force design). Figure 2.6(a) shows the equations describing the HA's two outputs while Figure 2.6(b) show the associated BDD. The HA is somewhat useful for all those occasions where you want to add two one-bit values.

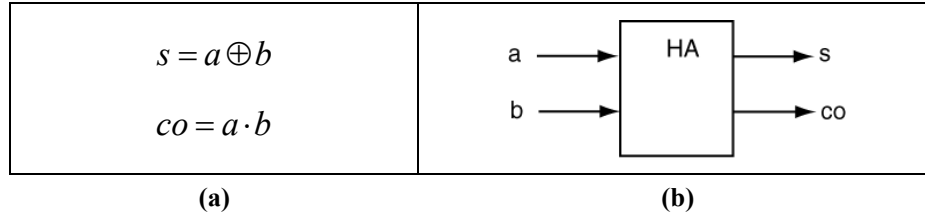


Figure 2.6: Boolean equations describing the outputs of the HA (a), and the associated BBD (b).

2.6.2 Full Adder

Once you figure out that a HA is not too useful, you move onto designing a Full Adder (FA). The FA is almost the same as the HA but the FA has an extra input which is considered the carry-in (meaning it's the carry in from a carry-out output of some other FA or HA). Figure 2.7(a) shows the equations describing the FA while Figure 2.7(b) shows the associated dark box model.

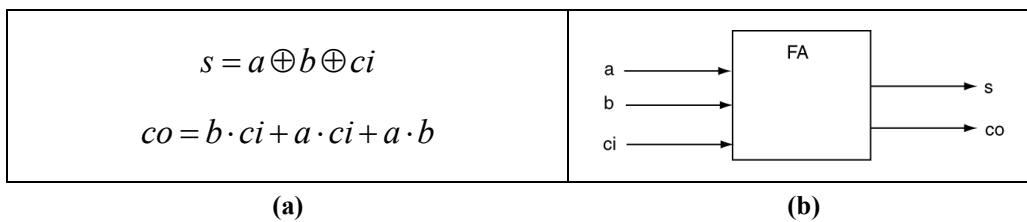


Figure 2.7: (a) Boolean equations describing the outputs of the FA, (b) the associated BBD.

2.6.3 Ripple Carry Adder

Once you realize that there is not too much opportunity out there for Half and Full adders, you generally move onto the ripple carry adder (RCA). The RCA is generally the first circuit you design using iterative modular design (IMD) noting that the 4-bit adder in Figure 2.8(a) would have required a truth table with 256 rows had it been designed using iterative design techniques. The IMD technique easily extends the 1-bit adding elements (HAs and FAs) to create multi-bit adders. Note that we can often times substitute the HA in the LSB position of the RCA with a FA, which gives up the ability to make larger RCAs (wider, or more bits) by connecting the RCAs in a cascade formation. Figure 2.8 (b) shows the BBD of a RCA while Figure 2.8(a) shows the RCA one level below Figure 2.8(b). The RCA in Figure 2.8(a) can include a carry-in input if we replace the HA in the LSB position with a FA.

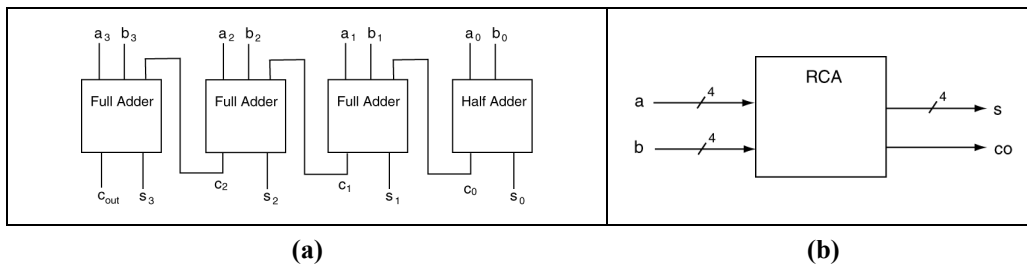


Figure 2.8: The guts of a 4-bit RCA (a), and the associated block diagram a 4-bit RCA (b).

We consider the RCA to be a Digital Design Foundation module. The RCA is a controlled circuit; Figure 2.9 shows the RCA in appropriate digital design foundation notation. As you would expect from an adder-type circuit, the RCA adds the two input operands (A & B) and the carry to generate the SUM output. Note the RCA has no control inputs, which means the device always performs the same operation on the three data inputs. The RCA's CO output provides status for the RCA's addition operation. Table 2.5 provides a description of all the inputs and outputs to the RCA.

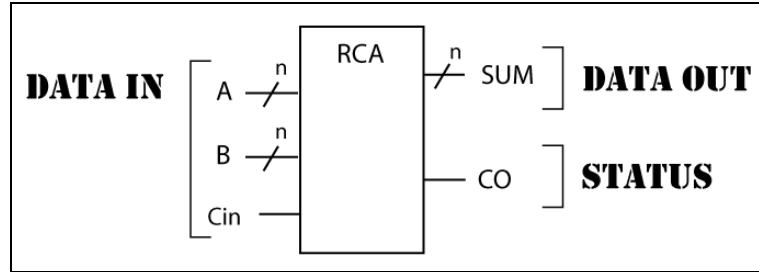


Figure 2.9: Data, control and status signals for a RCA.

	Signal Name	Description
INPUT DATA	A	One of two multi-bit addends (or operands). The data width of the two addends is equivalent.
	B	One of two multi-bit operands. The data width of the two addends is equivalent.
	Cin	A “carry in” input.
OUTPUT DATA	SUM	The result of summing the three inputs: two addends and the Cin input.
CONTROL	n/a	-
STATUS	Co	A “carry-out” signal; this signal shows when the summation operation has generated a carry. The carry is effectively the “n+1” bit of an n-bit RCA.

Table 2.5: The foundation matrix for a RCA.

2.6.4 Decoders

We use the word *generic decoder*, or just *decoder*, to refer to the digital device where the values of the decoder’s input always produce the same values on the decoder’s output. This is a generic definition of a decoder, thus we refer to most decoders as “generic” if we can model them in tabular format (a truth table). The basis of all things digital are basic gates, which we defined using tables; we can thus consider basic logic gates as decoders because of their tabular definitions.

In addition to the generic decoder, there is a *standard decoder*. The terms “generic” and “standard” decoders are terms that you won’t find in other digital design texts; we created these names to simplify the digital design paradigm. The standard decoder is a special type of a generic decoder and has a special relationship between the inputs and outputs. Figure 2.10 shows that, a standard decoder is a subset of a generic decoder. Standard decoders have specific uses while generic decoder usage is open-ended.

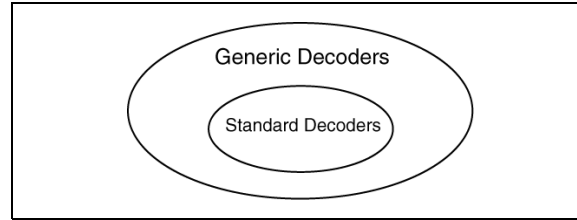


Figure 2.10: Venn diagram showing the hierarchy of decoders.

Modeling digital circuits using tables is powerful because we can easily translate the tables to a hardware description language (HDL) models. You may have a notion of the “power of tables” from your programming career in that using “look-up-tables” or “LUTs”; the same usefulness of LUTs applies to hardware modeling. The approach in modern digital design is to allow the development tools to do the work for you. Thus, modeling circuits using decoders (LUTs) hands a significant portion of the circuit implementation effort to the tools. If you need some “logic” using an HDL, the best approach is often to model the function in tabular format.

2.6.4.1 Generic Decoder

The “Generic Decoder” is the name given to any combinatorial circuit that implements a combinatorial circuit that you can’t label as some other standard digital circuit. Often times in digital design land, you’ll need to implement a circuit with a combinatorial “input/output relationship”. Any time you need to implement such a functional relationship, attempt to represent it in a tabular format, because you can then have defined a generic decoder. Figure 2.11 shows a BBD of a generic decoder. There can be any non-zero number of inputs and outputs; the number of inputs and outputs don’t need to match.

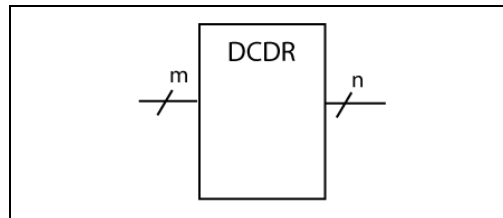


Figure 2.11: A black box diagram of a generic decoder.

You can define two general types of tables: 1) complete tables, and, 2) incomplete tables. Both tables are equally straightforward to model using an HDL. We define a complete table as a table that has a row for every unique combination of the circuit’s inputs; a non-complete table is any table that is not a complete table. We make this distinction so you realize that you don’t need to completely specify every possible input combination for generic decoders. Additionally, HDLs have solid support for modeling incomplete tables.

Figure 2.12 shows completely and incompletely specified tables. The table in Figure 2.12(a) has three inputs; because there are eight rows in Figure 2.12(a), we consider this table completely specified. The table in Figure 2.12(b) has three inputs, but only five of those three inputs combinations have outputs. Not declaring outputs indicates that for the missing input combinations, the designer for some reason does not care about the outputs. Another approach to non-complete tables is to list the missing inputs and state the outputs as don’t cares, which we do in Figure 2.12(c).

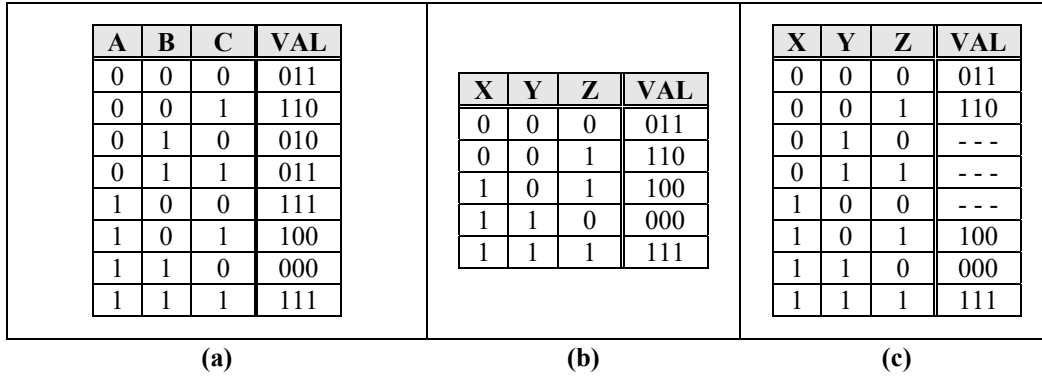


Figure 2.12: A completely specified table (a), and an incompletely specified table (b) & (c).

The generic decoder is one of our Digital Design Foundation circuits. We consider the generic decoder to be a controlled circuit; Figure 2.13 shows the generic decoder in appropriate foundation notation. The generic decoder models a table, so the **DATA_IN** inputs act as the independent variables and the **DATA_OUT** signals are the dependent variables. The generic decoder does not have either control inputs or status outputs. Table 2.6 provides a description of the inputs and outputs to the generic decoder.

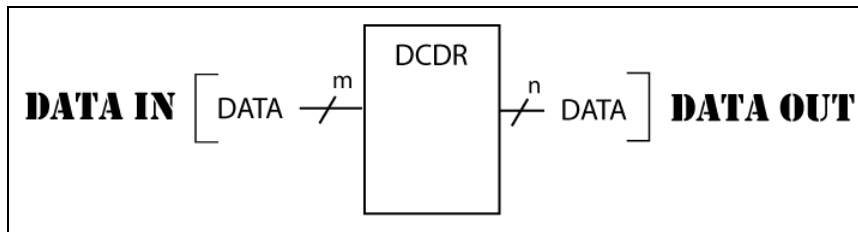


Figure 2.13: Data signals for a generic decoder.

	Signal Name	Description
INPUT DATA	DATA	The independent variable of the look-up-table
OUTPUT DATA	DATA	The dependent variable of the look-up-table
CONTROL	n/a	-
STATUS	n/a	-

Table 2.6: The foundation matrix for a generic decoder.

2.6.4.2 Standard Decoder

The Standard Decoder has some specific uses in digital design; we'll see some of those designs later in this text. We often label different flavors of standard decoders as DMUXes, but we'll avoid using such terminology here. Figure 2.14 shows a diagram gate-level diagram of a standard 2:4 decoder. There is a binary relationship

between the circuit's inputs (which are select inputs) and the circuit's outputs. If the standard decoder has one input, there are two (2^1) outputs; if the standard decoder has two inputs, there are four (2^2) outputs, and so on. Note that the output of the standard decoders form either one-hot or one-cold codes (the circuit in Figure 2.15 shows a one-hot code output).

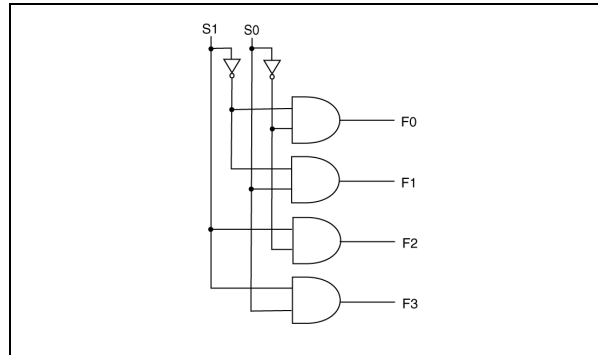


Figure 2.14: The important underlying details of a standard decoder.

The standard decoder is a Digital Design Foundation Module. The standard decoder is a controlled circuit; Figure 2.13 shows the standard decoder in appropriate foundation notation. The standard decoder has no data inputs; the only inputs are the **SEL** inputs, which decide the exact format of the **DATA_OUT** signals. By definition, the **DATA_OUT** signals form a one-hot code. Table 2.7 provides a description of all the inputs and outputs to the standard decoder.

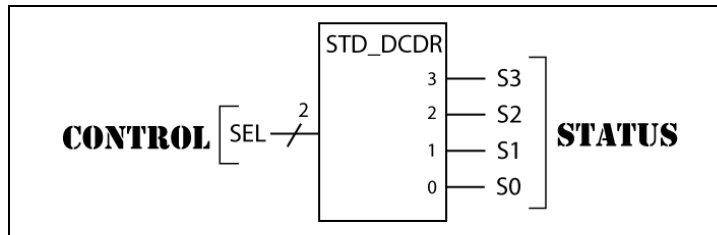


Figure 2.15: Control and status signals for a 2:4 standard decoder.

	Signal Name	Description
INPUT DATA	n/a	-
OUTPUT DATA	n/a	-
CONTROL	SEL	The inputs that select the desired form of the output.
STATUS	S(3:0)	The output signals chosen by the SEL input.

Table 2.7: The foundation matrix for a standard decoder.

2.6.5 Multiplexor

The multiplexor, or MUX, is an element that “selects” or “chooses” one of many data elements on the input to be passed to the output. The inputs to a MUX are the data (the things “being chosen”) and control lines (does the actual choosing). The control lines have the typical binary relationship to the circuit inputs in that one control line can choose between two (2^1) items to appear on the MUX outputs, two control lines can choose between four (2^2) items to appear on the circuit outputs and so on. The MUX’s data inputs can be single signals or bundles. Figure 2.16 shows the inner workings of a 4:1 MUX with single-bit data inputs.

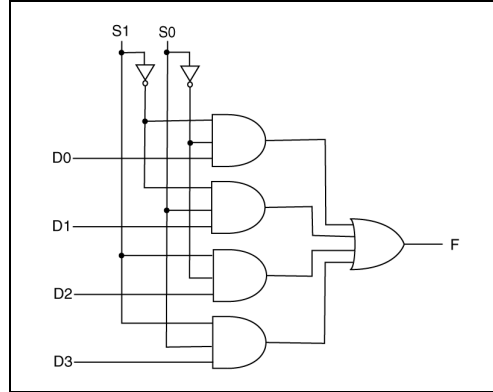


Figure 2.16: The well-known guts of a basic 4:1 MUX.

The MUX is a Digital Design Foundation Modules. The MUX is a controlled circuit; Figure 2.17 shows the MUX in appropriate foundation notation. The SEL signal is a control input and decides which DATA_IN signal becomes the DATA_OUT signal. The MUX thus has a control input but has no status outputs. Table 2.8 provides a description of the MUX’s inputs and outputs.

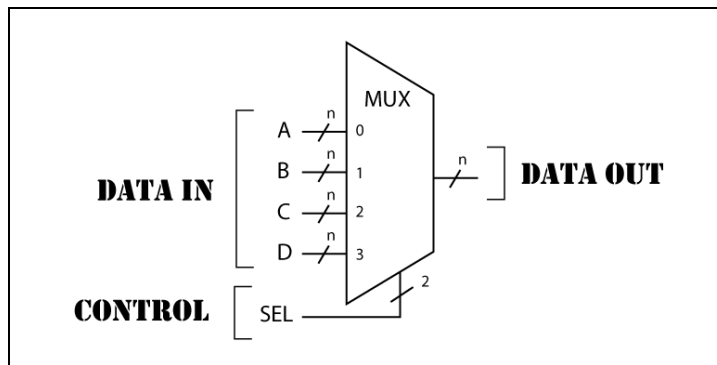


Figure 2.17: Data and control signals for a 4:1 MUX.

	Signal Name	Description
INPUT DATA	A, B, C, D	Data inputs to the MUX; MUXes can have any number of data inputs. One of these data inputs becomes the single data output.
OUTPUT DATA	F	A single output, which is one of the inputs as selected by the SEL signal.
CONTROL	SEL	Selects which data input appears on F . The width of the SEL signal is such that $2^{\text{SEL}} \geq$ to the number of data inputs.
STATUS	n/a	-

Table 2.8: The foundation matrix for a MUX.

2.6.6 Comparator

The comparator is another common digital circuit. While comparators in general can come in many different forms, Figure 2.18 shows the general form. It is referred to as a general form because there is only one output (indicating whether the two inputs are equal or not). Other less general comparator forms include outputs such as “great than or equal”, “greater than”, etc. The classic things to remember about comparators are 1) they involve EXOR-type functions, and, 2) we generally design them using iterative modular design (IMD). Figure 2.18(a) and Figure 2.18(b) show black box models and circuit implementation of a 2-bit comparator, respectively.

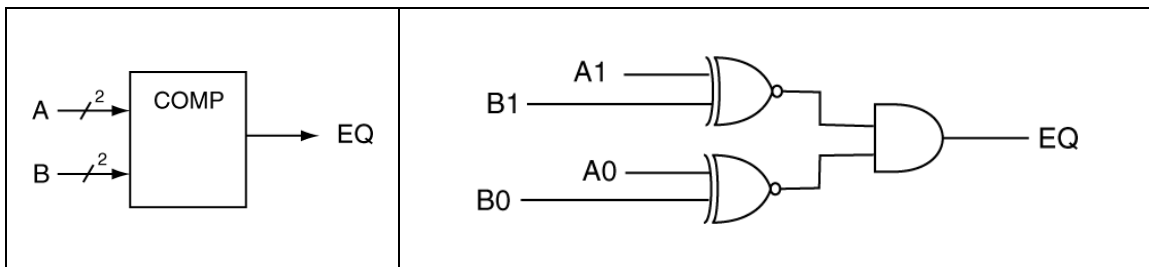


Figure 2.18: A black box model and a circuit diagram for a standard 2-bit comparator.

The comparator is a Digital Design Foundation module. The comparator is a controlled circuit. Figure 2.19 shows the appropriate digital design foundation notation for the comparator. Comparators always have two inputs, but we can choose between which comparator outputs we want to include in our design (so our comparator module has at least one, but not greater than three outputs). The **LT** output indicates when the **A** input is less than **B** ($A < B$), while the **GT** output indicates when $A > B$. The **EQ** output indicates that $A = B$.

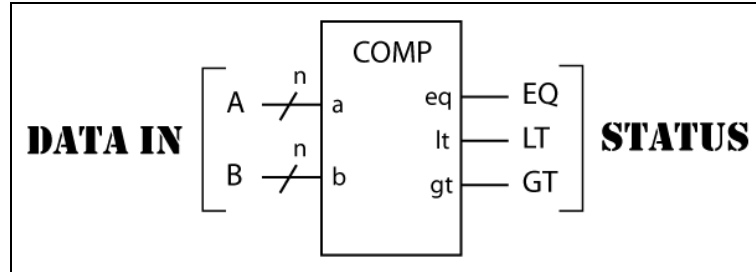


Figure 2.19: Typical data, and status signals for a comparator.

	Signal Name	Description
INPUT DATA	A, B	Two values to be compared; these values have equivalent data widths.
OUTPUT DATA	n/a	-
CONTROL	n/a	-
STATUS	EQ, LT, GT	Signals that indicate a relation between the two inputs A & B. EQ is asserted when A=B, LT is asserted when A<B, GT is asserted when A>B.

Table 2.9: The foundation matrix for a comparator.

2.7 Sequential Circuits

Sequential circuits are circuits that have the ability to “remember” at least one bit. The official definition of a sequential circuit is that the circuit’s outputs are dependent upon the sequence of inputs. The notion of remembering bits give sequential circuits the notion of having “state”. And thus, the notion of finite state machines (FSMs) is born.

The simplest 1-bit storage element in digital design land was the “latch” which was based on cross-coupled NOR and cross-coupled NAND cells. We consider latches to be “level sensitive” devices. Because we generally need more control over devices, we usually use another 1-bit storage element, which we refer to as a flip-flop. There are several types of flip-flops out there, but D flip-flops are the most common flip-flop in digital design based on their simplicity. Recall that the D in “D flip-flop” refers to “data”. The D flip-flop generally has a clock input; changes in state of a D flip-flop are synchronized to an active clock edge (either a rising or falling clock edge, but not both).

2.7.1 Simple Registers

Registers are multi-bit storage elements modeled as a parallel configuration of D flip-flops that share a common clock signal. When we refer to “registers”, we refer to simple registers; we refer to other common register types by their names: counters and shift registers. Figure 2.20 shows four D flip-flops assembled to act as a simple multi-bit register. In particular, Figure 2.20(a) shows the block diagram for a 4-bit register and Figure 2.20(b) shows the underlying circuit. The block diagram in Figure 2.20 (a) shows that this register is rising-edge triggered and that every flip-flop shares a common clock signal.

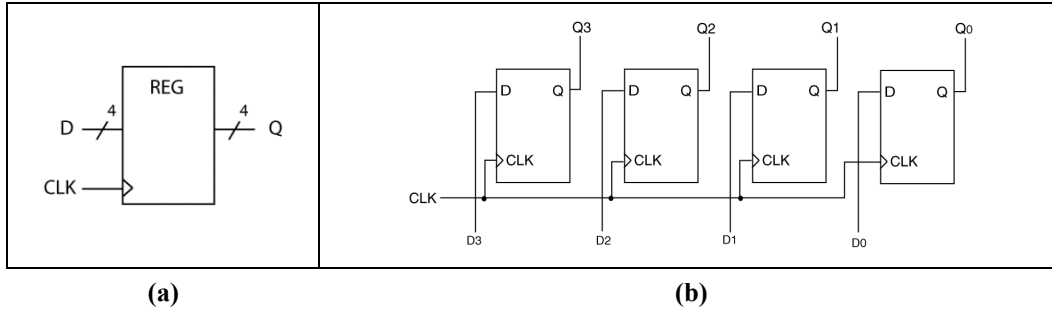


Figure 2.20: A block diagram for a 4-bit register (a), and the associated lower-level model (b).

The register is a controlled circuit and is one of our Digital Design Foundation Modules. Figure 2.21 shows the appropriate digital design foundation notation for the register with a basic set of control features. Registers typically have both data inputs and data outputs. The typical set of controls for a register includes synchronous load signals (LD) and an asynchronous clear input. Table 2.10 show a complete description of the registers input and output signals.

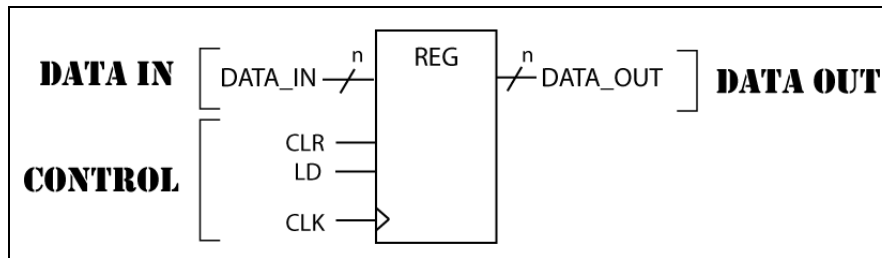


Figure 2.21: Typical data and control signals for a register.

	Signal Name	Description
INPUT DATA	DATA_IN	The data that can be latched into the register’s storage elements.
OUTPUT DATA	DATA_OUT	The DATA_OUT signal is the data currently being stored in the counter’s storage elements.
CONTROL	CLK	Registers are synchronous circuits, in that the loading of data to the register happens on the clock edge.
	LD	Allows the latching (loading) of the DATA_IN signal to the counters storage elements. This signal is always synchronous.
	CLR	Latches 0’s into each of the register’s storage elements; can be synchronous or asynchronous.
STATUS	n/a	-

Table 2.10: The foundation description for a simple register.

2.7.1.1 Special Register Circuits: The Accumulator

The accumulator is a useful and common circuit in digital design. The accumulator does what its name implies: it accumulates. In digital design is that we can only add two numbers at a time, but often we need to add more than two numbers. In this case, we still can only add two numbers at a time, but we add the successive values to a “running total”. The resulting circuit is relatively simple: we need a device to store the running total (a register) and a device to do the adding (an RCA). Since we have flexibility in the features we add to the register, when we design an accumulator, we need to make sure of the following items:

- We need to ensure we can clear the register, as anytime we’re accumulating something; we typically start accumulating with a register value of zero.
- We need to ensure the width of the register is wide enough to hold the maximum possible value based on the width of the values we’re adding and the maximum quantity of values we need to add. For the sake of simplicity, the width of the accompanying RCA generally has the same data widths as the register, which requires bit-stuffing of the input RCA’s data-widths.

Figure 2.22 shows a diagram of a generic accumulator. Note that some other entity needs to issues control signals to the counter (**CLR**, **LD**, & **CLK**). For this example, we’re not connecting these signals, but we do in later examples that use finite state machines (FSMs). Here are some important details.

- The register has a **CLR** control input so that we can clear the value stored in the circuit before we commence accumulating. The circuit also has a **LD** control input, which some other entity provides
- We list the output data width as “n” bits and the input data width as “m” bits. The notion here is that we’ll be adding a bunch of numbers of width “m”. In doing this we need to do two things:
 1. Ensure the output data width “n” is wide enough to handle the maximum possible value of the accumulation
 2. Bit-stuff the “m” width input data to match the “n” width of the output. We do this because we expect both inputs of the RCA to have the same data width. Figure 2.22 indicates this bit-stuffing with the square containing the “+”. For this diagram, we are stuffing (n-m) bits to the DATA input.

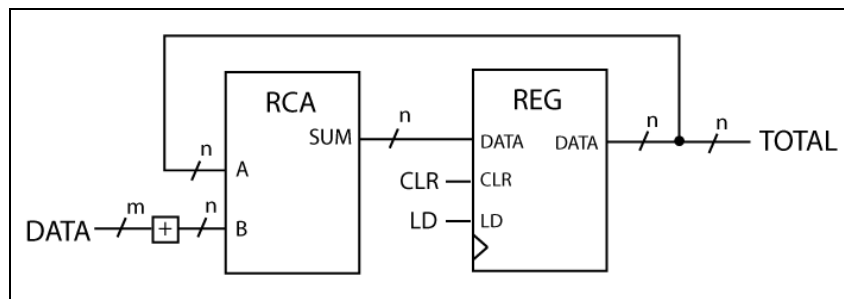


Figure 2.22: Generic circuit for an n-bit accumulator.

2.7.2 Counters: Registers with Features

A counter is a type of register, so it inherits all the attributes of a register. The main new “feature” of a counter is that it outputs a given sequence of code words, which is the “count” sequence. Counters typically synchronize their stepping through the count sequence to an active clock edge input to the counter. Counters can have one or more typical operational features, which we control with the counter’s “control” inputs. Counters can also have status outputs that provide external circuits information about the counter.

Our approach is to define and describe every word and/or term you typically hear in the context of counters, and then do a few example problems. When you say the word counter, it has a few standard connotations that you

can assume are true unless told otherwise. The following list describes even more assumptions made when dealing with counters.

- Because counters are registers, they are sequential circuits
- An active clock edge synchronizes a counter's traversing of the count sequence; there is one count value, or code-word, from the count sequence at each clock cycle.
- A counter's output represents a repeatable sequence of a given number of bits. The sequence the counter "counts" in does not change; the bit-width of the counter won't change either.
- When a counter completes a traversal through its count sequence (either in the up or down direction), the counter automatically starts counting over (and is thus "circular").

There is a set of vernacular associated with counters. Digital designer must be fluent with all the new terms associated with counters so they can converse with their peers and understand important things such as datasheets. Here are the common terms associated with counters:

- **n-bit Counter:** A counter that uses n-bits to represent each of the values (or code words) in its count sequence.
- **Up Counter:** A counter that counts up (increasing count values in count sequence).
- **Down Counter:** A counter that counts only down (decreasing count values in count sequence).
- **Up/Down Counter:** A counter that can counter either up or down according to a control input on the device.
- **Increment:** An operation associated with counters where '1' is added to the current value of counter.
- **Decrement:** An operation associated with counters where '1' is subtracted from the current value of counter.
- **Counter Overflow:** The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its largest representable value to its smallest value.
- **Counter Underflow:** The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its smallest representable value to its largest value.
- **Cascadeable:** A characteristic of many digital devices such as counters and shift registers that allow you to effectively increase the overall bit-width of devices providing inputs and outputs such that you can easily interface the devices. One such output is the "ripple carry out".
- **Count Enable:** A signal on counters that enables the counting operation of the counter when asserted and disables the counting when not asserted.
- **Ripple Carry Out (RCO):** A signal typically found on counters that indicate when the counter has reached its maximum count value (for an up counter) or minimum count (for a down counter). Counters often use the term RCO to indicate when the counter has reached its terminal count value.
- **Parallel Load:** A characteristic of a counter or shift register indicating that all the storage elements in the device can simultaneously latch external values.
- **Circular:** When counters overflow their maximum or minimum counts, we consider them to "overflow". Counters are typically circular meaning that when the counter reaches the

maximum value, it automatically continues counting in the same direction starting at the minimum value⁴.

The counter is a controlled circuit and one of our Digital Design Foundation modules. Figure 2.23 shows the appropriate digital design foundation notation for the counter. This foundation module is more flexible (resulting in more control inputs) and thus harder to define than other foundation modules. For example, the only required signal for a counter is a clock, as we consider the counter a synchronous device; the only required information we need to know about counters is the bit-width of their internal storage elements. Because counters are straightforward to design and/or model in with an HDL, we typically only include (or connect) counter inputs and outputs as we need them.

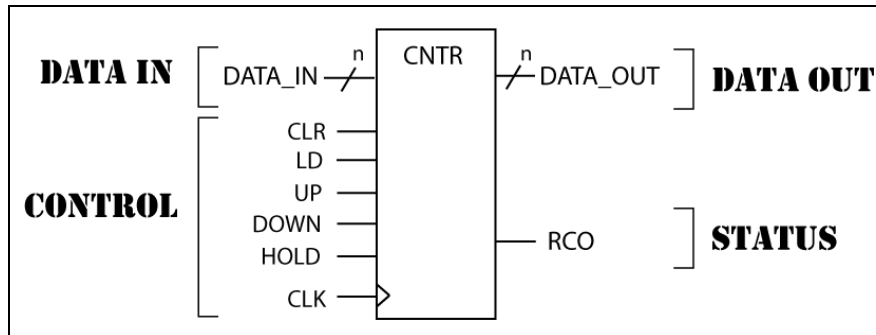


Figure 2.23: Typical data, control and status signals for a counter.

Table 2.11 shows all the inputs and outputs that we can typically associate with a counter. Table 2.11 essentially lists a set of features that we can apply to a counter. The two things to note about this list is 1) that not every counter has every listed feature, and 2) actual counter implementations typically combine many of the control features as required into less signals than listed.

⁴ This characteristic is for an up counter; the same idea is true for a down counter.

	Signal Name	Description
INPUT DATA	DATA_IN	A counter is a register, so it can typically load data in to the counter's storage elements. The DATA_IN input is the data that is loaded to the counter.
OUTPUT DATA	DATA_OUT	A counter is a register, so the DATA_OUT signal is the data currently being stored in the counter's storage elements. The DATA_OUT signal is necessarily a given value in the counter's count sequence.
CONTROL	CLK	Counters are typically synchronous circuits, in that many counter operations are synchronized with the active edge of the clock signal.
	LD	As with registers, this signal controls the latching (loading) of the DATA_IN signal to the counters storage elements. This signal is always synchronous.
	CLR	Latches 0's into each of the counter's storage elements. Can be synchronous or asynchronous.
	HOLD, EN	Prevents the output from changing (HOLD) or enables the output to change (EN) based on other control signals (sort of the same idea)
	UP	Directs counter to count "forward" in the sequence; the an asserted up signal counts forward while a non-asserted count signal counts backwards
	DOWN	Directs the counter to count "backward" in the sequence.
STATUS	RCO	This signal indicates when the counter has reached the terminal value in the associated count sequence. For counters counting up, the terminal value is the max count value (all internal storage elements set); for counters counting down, the terminal value is the min counter value (all internal storage elements cleared).

Table 2.11: The foundation description for a full-featured counter.

2.7.3 Shift Registers

A shift register is another type of register. Shift registers, and their various flavors, are useful devices because of their ability to quickly perform a small but useful subset of mathematical operations.

We can decompose a shift register down to its most basic component, which we refer to as a shift register cell. This cell is a storage element, which we model as a D flip-flop. Figure 2.24 shows a schematic diagram of a generic shift register. Upon further inspection, you should discern the following:

- We can model the n-bit shift register as a set of "n" specially connected D flip-flops. The D flip-flops in the shift register share the same clock signal.
- The difference between simple registers and shift registers is in the way that the individual storage elements connect to each other. While simple registers have D flip-flops that receive data from the inputs, the shift register's storage elements receive data from interconnections between individual storage elements. Figure 2.24 shows that the output of one flip-flop becomes the input to the adjacent flip-flop in the shift register, which allows the device to "shift".
- The number of bit storage elements in a shift register defines shift registers. The shift register in Figure 2.24 represents a generic model of a shift register including the magic ellipsis in strategic locations. Common descriptions of shift registers include "a 4-bit shift register" or "an 8-bit shift register", etc. Figure 2.24 shows a generic "n-bit shift register".

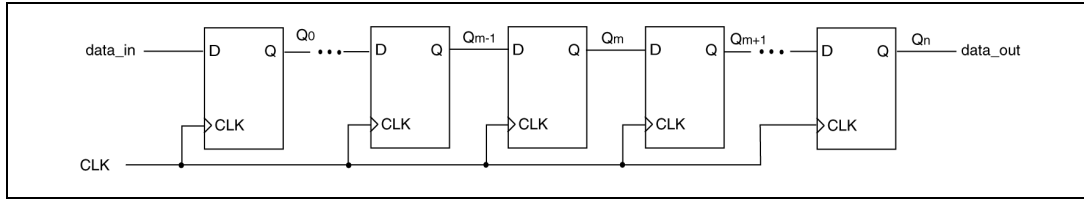


Figure 2.24: A typical n element shift register.

Figure 2.25(a) shows a schematic diagram of a 4-bit shift register while Figure 2.25(b) shows a model of the underlying circuitry. Figure 2.26 shows an example timing diagram for a 4-bit shift register in Figure 2.25 (b). Figure 2.26 contains annotations to help with the following description.

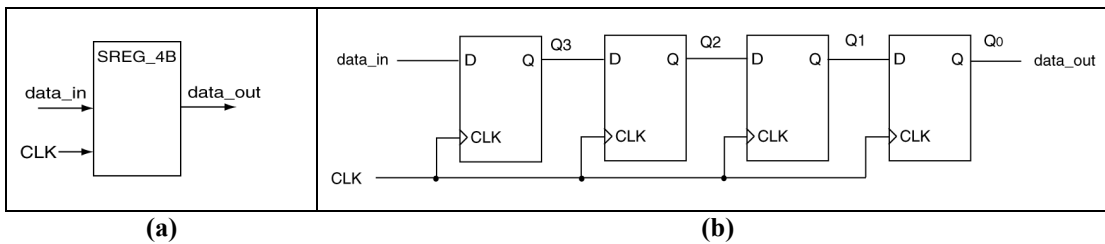


Figure 2.25: A block diagram for a 4-bit simple register (a) and a model of the underlying circuitry of a 4-bit shift register (b).

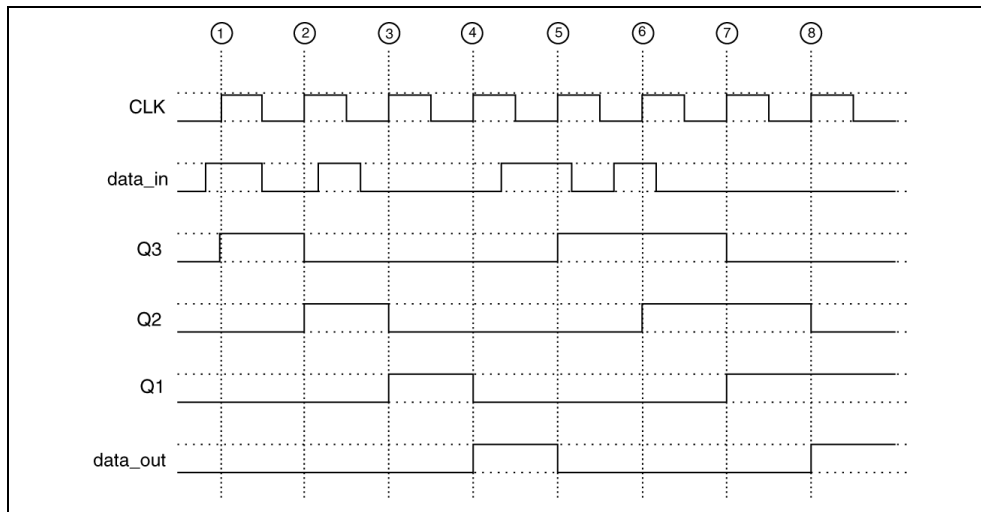


Figure 2.26: An arbitrary timing diagram associated with the shift register of Figure 2.25(b).

- The schematic in Figure 2.25(b) labels each of the internal shift register signals to help describe the operation of the basic shift register in Figure 2.26. The “**Q_x**” notation indicates the bit positions of the storage elements in the shift register. We consider **Q₃** the higher order bit while **Q₀** (or **data_out**) is the lowest order bit⁵. Note that **data_out** and **Q₀** are the same signal.
- We consider shift registers to “shift” in either direction; that is, they shift to the left (“shift left”) or shift to the right (“shift right”). Figure 2.25 (b) shows a right-shifting shift register.
- The notion of this circuit shifting is primarily a term of convenience and not altogether accurate. The “thing” being shifted in Figure 2.25 (b) is the “data”. Another way to view this is that the

⁵ We often use shift registers for mathematical operations; numbers generally have weights associated with the bit positions.

circuit inputs 1's and 0's from the left side of the circuit and passing them through to the right side.

- Since this is a sequential circuit, the storage elements have a state associated with them. For the timing diagram of Figure 2.26, the initial state of each storage element is '0', which is arbitrary.
- On the clock edge labeled '1', all of the flip-flops transfer the value on their inputs to their outputs. On the active clock edge, the left-most flip-flop latches "data_in"; **Q3** latches into the second to the left-most flip-flop, etc.

If you stand back a few paces, you can see the so-called shifting action of the shift register. The individual signals are shifted versions of each other; specifically, Q3 is a shifted version of "data_in", Q2 is a shifted version of Q3, etc. Another way to view this is that the "data_out" signal is a delayed version of the "data_in" signal. In this case, Q0 is a delayed version of Q3; the delay is three clock cycles because the pulse appearing on Q0 is the same pulse that appeared on Q3 three clock cycles earlier. The right-shift operation (one shift in the right direction) is the same thing as a divide-by-two operation with truncation⁶.

The shift register is a controlled circuit and one of our Digital Design Foundation Modules. We generally consider all shift register operations synchronous, except for the **CLR** input, which is sometimes asynchronous. Because shift registers are straightforward to model in with an HDL, we typically only include (or connect) inputs and outputs as we need them. The width of the **SEL** input sufficient to support the shift register's operations. Figure 2.27 shows the foundation module for a shift register.

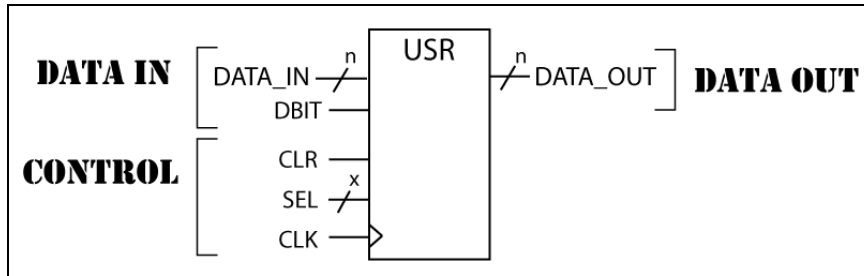


Figure 2.27: Typical data, control and status signals for a universal shift register.

⁶ Truncation means the lowest order bit is lost; a similar operation is "round-up" where the value of the lowest order bit is "taken into account" and your weeds are killed at the same time.

	Signal Name	Description
INPUT DATA	DATA_IN	A counter is a register, so it can typically load data in to the counter's storage elements. The DATA_IN input is the data that is loaded to the counter.
	DBIT	The bit that becomes the left-most bit for a right shift operation or the right-most bit for a left-shift operation
OUTPUT DATA	DATA_OUT	The DATA_OUT signal is the data currently being stored in the counter's storage elements.
CONTROL	CLK	Registers are synchronous circuits; most operations are synchronized with the active edge of the clock signal.
	CLR	Latches 0's into the register's storage elements; can be synchronous or asynchronous.
	DBIT	The bit that shifts into the register on shift operations, which is the new left-most bit or the new right-most bit for shift right and shift left operations, respectively.
	SEL	These bits select the operation the shift register performs. These operations could include: shift left, shift right, hold, load, rotate left and/or right, barrel shifts, etc. The width of this input depends on the number of possible operations.
STATUS	n/a	-

Table 2.12: The foundation description for a universal shift register.

2.7.4 Registers: The Final Comments

A register is nothing more than a set of bit storage elements that share a single clock signal. In other words, registers are a parallel configuration of signal bit storage elements; what makes them parallel is the fact changes in register state are generally synchronized to some event (usually a clock edge). Registers (simple, counters, and shift registers) are quite common in digital design. All registers are sequential circuits, but some registers have more “features” than others. The Venn diagram in Figure 2.28 shows how the various flavors of registers relate to each other.

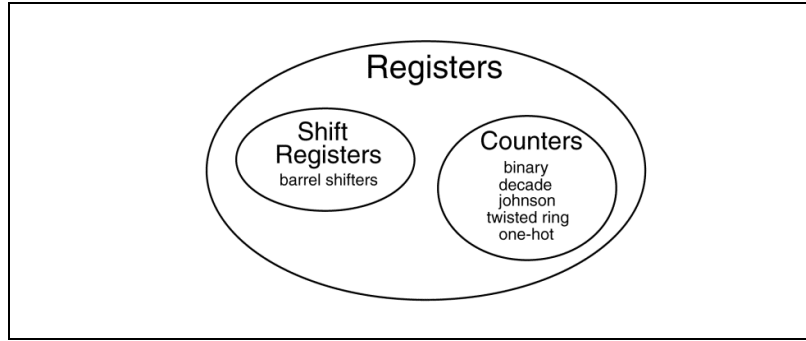


Figure 2.28: Venn diagram for the register family.

The main difference between the many types of register is their feature set. In an attempt to show all the possibilities in one spot, Table 2.13 shows a possible breakdown of the register types and their relation to each other. Keep in mind that many of the features listed in Table 2.13 can be either synchronous or asynchronous.

Register Type	Sub-Types	Features
plain register		parallel load, preset, clear, load enable, cascadeability
shift register	Universal Shift Register (USR), Barrel Shifter	parallel load, preset, clear, load enable, shift left/right, arithmetic shift left/right, hold, rotate left/right, cascadeability
counter	Up/Down Counters, Decade Counters	parallel load, hold, preset, clear, increment, decrement, cascadeability

Table 2.13: The feature progression of the register device.

2.8 Finite State Machines (FSMs)

The term “Finite State Machine” has many official meanings and definitions in digital-land. As you saw previously, any circuit that has the ability to remember something (namely bits), can be regarded as having a “state”. A circuit-oriented definition of a FSM is this: *a circuit whose behavior can be modeled using the concept of “state” and the transitions between the various states in that circuit.*

We generally use FSMs for two purposes: 1) designing counters with special count sequences, and, 2) as controller circuits, or *a circuit that control other circuits.* People use FSMs in one form or another in many different technical disciplines and each discipline seems to have its own particular flavor of representing FSMs. Despite these many flavors to modeling FSM, always keep in mind that the best approach is to be clear in a way that expedites the transfer of information. Always remember that the state diagram is a model that visually describes the behavior of the FSM.

2.8.1 High-Level Modeling of Finite State Machines

Digital design typically classifies FSMs as one of different two types: Moore-type or Mealy-type. We opt to simplify this definition as follows: there is only one type of FSM, but FSMs can have one of two types of outputs: Moore-types and/or Mealy-type outputs. All FSMs share the same properties: the only difference is the two types of FSM outputs.

Figure 2.29 shows a basic model of an FSM. We can abstract the FSM’s internal circuitry into three separate blocks: 1) Next State Decoder, 2) the State Registers, and 3) the Output Decoder. The output decoder can have two types of outputs, which we refer to as Moore and Mealy-type outputs; Moore-type outputs are a function of the present state of the FSM while Mealy-type outputs are a function of both the FSM’s present state and the external inputs. Table 2.14 provides a detailed description of the FSM’s individual modules.

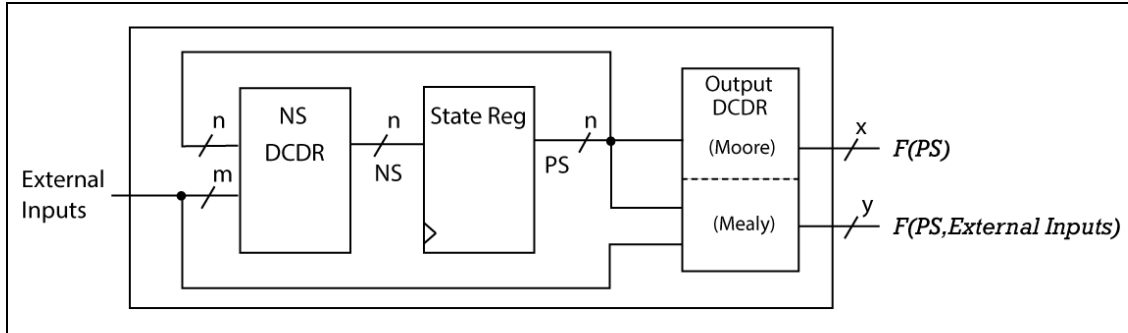


Figure 2.29: The lower-level BBD for a generic FSM.

Module	Description and Comments
State Registers	The State Registers represent the memory elements in the FSM. The term <i>register</i> implies the circuit is a synchronous storage element. The state register is the only sequential module in an FSM; the other two modules are both combinatorial circuits. The state registers store the <i>state variables</i> of the FSM; the value stored in the state registers is the state of the FSM.
Next State Decoder	The Next State Decoder is a combinatorial circuit that provides excitation input logic to the state register module. The next state logic generally has two types of inputs, which provide the <i>excitation inputs</i> to the state registers: 1) the current value of the state variables (the present state, and, 2) the inputs from the external world. Excitation inputs to the state registers determine the <i>next state</i> of the state register. On the next active clock edge, the data inputs to the state registers becomes the next state of the FSM, which is why we refer to next state decoder as the <i>next state logic</i> . The external inputs to the next-state decoder function as status signals from the world outside of the FSM.
Output Decoder	The Output Decoder is a combinatorial circuit that generates the external outputs of the FSM. The output decoder is responsible for generating the two types of FSM outputs: Moore-type outputs and Mealy-type outputs. Moore-type outputs are a function of the FSM's state only, while Mealy-type outputs are a function of both the FSM's state and the external inputs to the FSM. The outputs from the output decoder generally serve as control signals to the device(s) controlled by the FSM.

Table 2.14: A detailed description of the three main FSM functional blocks.

2.8.2 The FSM: Symbology Overview

Probably the hardest thing about FSMs is understanding the state diagram symbology. The good news is that it's relatively simple once you work with it.

2.8.2.1 The State Bubble

FSMs use the state bubble to represent a particular *state* in an FSM. Figure 2.30(a) shows a typical state bubble. The following verbage lists some of the key features regarding the state bubble:

- A state needs some way to visually delineate it from other states, which is why the state bubble contains identifying information. State bubbles provide the state with a symbolic name that identifies the purpose of that state to the human reader.
- Timing diagrams represent the states by the time slots representing the possible states. Figure 2.30(b) shows that the boundaries of these time slots delineated the associated active edges of

the FSM's clock input, which is the clock input to the state registers.. Figure 2.30(b) show that the state registers are rising-edge triggered (RET) because the rising clock edge defines the state boundaries.

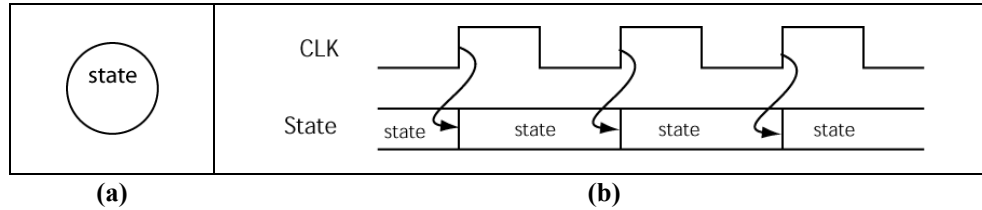


Figure 2.30: The State Bubble and associated timing diagram.

2.8.2.2 The State Diagram

The state diagram is one of many methods we use to model FSMs. The main purpose of the state diagram is to convey meaning and understanding to the human viewer. State diagrams provide four main forms of information: 1) the states in the FSM, 2) the state transitions the FSM makes, 3) the input conditions controlling the state transitions, and, 4) the output values associated with the FSM. Figure 2.31(a) shows a fragment of a state diagram. The following verbage describes some of the key features of this state diagram.

- We refer to the terminology describing how a FSM goes from one state to another as a *state transition* or just *transition*. State diagrams use singly directed “arrows”, directed from the source state to the destination state to represent state transitions.
- There are only two possible state transitions in a state diagram from a given state. On the active clock edge, a transition can occur from, 1) one state to another state (indicated by the “state change” label in Figure 2.31(a)), or, 2) the FSM can remain in the same state (indicated by the “no state change” label in Figure 2.31(a)). We refer to the “no state change” arrow as a “*self-loop*”.
- The state diagram contains no explicit clock signal; the clock signal is implied rather specifically listed. The only part of the clock signal we’re interested in is the active clock edge; the state transition arrows represent what action occurs on the active clock edge associated the FSM.
- The two states in Figure 2.31(a) have unique names. In real life, you would want to give these more meaningful names such as something to indicate why the state exists.
- The state names in Figure 2.31(a) give no indication how we would represent the states if we were to implement the FSM. In other words, the state diagram provides no commitment to the actual state variable assignment that disambiguates the states on a hardware level.
- The relation between the timing diagram in Figure 2.31(b) and the state diagram in Figure 2.31(a) is the key to understanding state diagrams in general. When we talk of state, we’re talking about all the time in-between the active edges of the clock. The state bubble essentially represents all the time between any two active edges of the system clock. The state transition arrow represents what happens on each of the FSM’s active clock edges. On each clock edge, one of two things must necessarily occur: the FSM transitions either to another state or the FSM remains in the same state. A state transition occurs on every active clock edge, but sometimes it transitions back to the same state.
- The concept of Present State (PS) and Next State (NS) is somewhat hard to define in a timing diagram such as the one in Figure 2.31(b). The problem is that the present state (and hence the next state) is constantly changing as you travel from left to right on the time axis. If you declare one state as the present state, then you can declare the following state as the next state relative to the present state. This definition changes as you traverse the timing diagram. PS/NS tables do a better job of presenting present and next-state information.

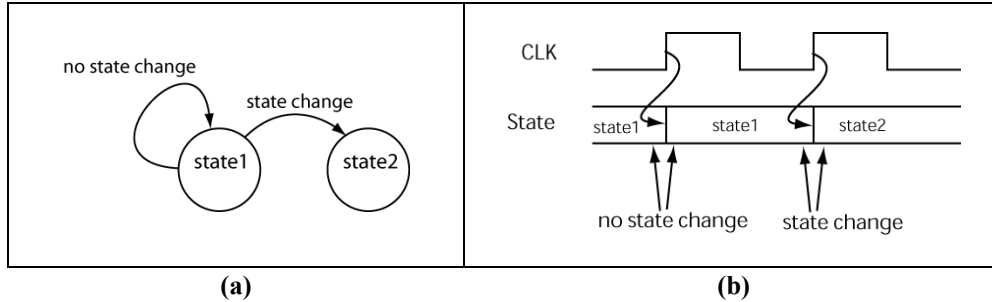


Figure 2.31: A state diagram (a) and the associated timing diagram (b) with interesting details.

2.8.2.3 State Transitions Controlling Conditions

As you would guess from examining the state diagram of Figure 2.31(a), there must be some mechanism that decides which transition will occur from a given state on the next active clock edge. In Figure 2.31(a), state1 has two arrows leaving the state, which mean there are conditions associated with those arrows that decide on which transition occurs.

There are two forms of information that determine the transition a FSM takes: 1) at least one of the external inputs to the FSM, and, 2) the present state of the FSM⁷. The external inputs to a FSM are generally status signals from the circuit the FSM is controlling. Each state has its own set of conditions that govern transitions, so we're concerned on a state-by-state basis what external input conditions determine the state transitions from a given state. Figure 2.32 shows that we indicate the conditions governing transitions by placing the conditions next to the state transition arrows. On this note, there are three important things to keep in mind:

- 1) The conditions associated with the state transition arrows leaving a given state must be mutually exclusive. This means that there can never be the same input conditions associated with two different transitions arrows leaving the same state.
- 2) The set of conditions associated with a particular state must be complete, meaning it must provide a transition arrow for every possible meaningful combination of input conditions. If there is a set of conditions in given state not covered by the associated state transition arrows, the FSM won't know what to do⁸. State diagrams should leave no room for guessing, if they do, their behavior will not be deterministic (which is an impressive way of saying your FSM won't always work as you intend).
- 3) If the transition is unconditional, then the state diagram indicates this by listing a "don't care" symbol by that transition.

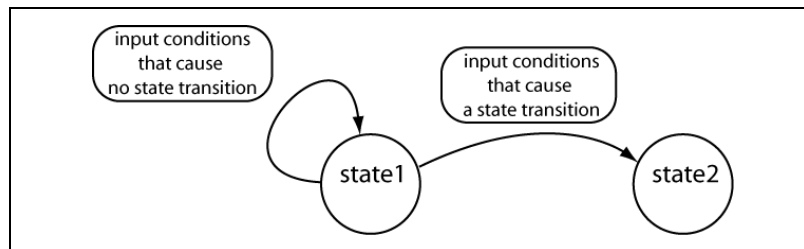


Figure 2.32: How state diagrams indicate the conditions associated with state transitions.

⁷ Recall that the PS and the external inputs are the inputs to the next-state decoder.

⁸ In cases such as these, the tools you're working with will generally not tell you about such conditions and will arbitrarily decide what it wants to do. In general, software design tools are generally make the assumption you know what you're doing and that you always do the right thing. With that assumption, the tools gladly fill in any details that you have unintentionally forgotten.

2.8.2.4 FSM External Outputs

The external outputs from a FSM are generally “control signals” that are controlling other circuits. The state diagram has different states and thus the control signals output from one state are generally not the same as control signals output from other states, so the FSM is performing different control functions based on the different states.

There are two different types of outputs in a FSM: Mealy-type outputs and Moore-type outputs. Although these outputs are similar in their controlling functions, they have one major difference. The outputs Moore-type outputs are a function of the state variables only while the Mealy-type outputs are a function of both the state variables and the current external inputs. Since Moore-type outputs are a function of the state variables only, we represent them by placing their values inside the state bubble. Figure 2.33 shows a state diagram that uses this approach. There can be any number of outputs represented inside the bubble.

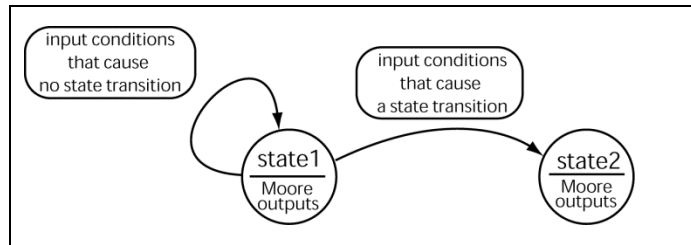


Figure 2.33: The State Bubble with associated Moore outputs.

We can't represent Mealy-type outputs inside the state bubble because they are a function of the external inputs as well as the state variables. To account for these characteristics in a state diagram, we list the Mealy-type outputs next to the external inputs associated with the individual state transition arrows. We separate external inputs and outputs with a forward slash. Figure 2.34 shows an example of this approach; we comma-separate multiple Mealy-type outputs.

Figure 2.34 lists two sets of Mealy-type outputs because there are two transitions from state1. The arrows are associated with the state transitions, which are based upon the current external inputs; the Mealy-type outputs are also a function of those same inputs. Since the Mealy-type outputs are a function of the external inputs, we represent them by placing them next to the external inputs. *We always associated Mealy-type outputs with the state the arrow is leaving (and not the state the arrow is entering).* Additionally, *the Mealy-type output is associated with the external input, not the transition arrow as the diagram seems to show.* To say the Mealy-type output is associated with the transition arrow indicates you should rethink the issue.

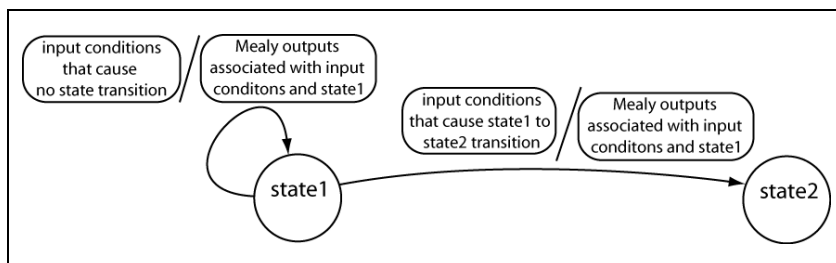


Figure 2.34: Representing Mealy-type outputs in a state diagram.

In addition, we can represent both Mealy and Moore-type outputs in the same state diagram. Figure 2.35 shows an example of a state diagram that contains both Mealy and Moore-type outputs.

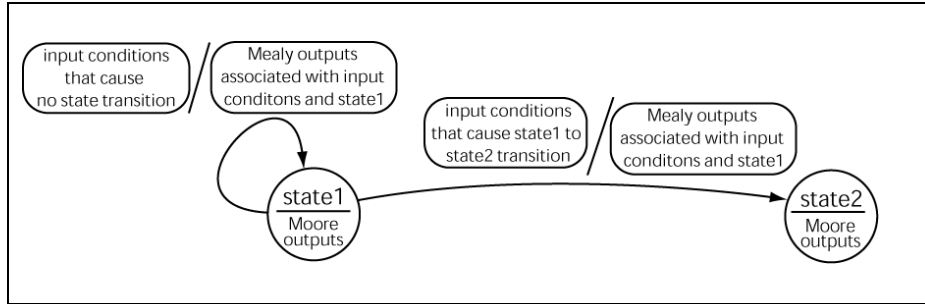


Figure 2.35: A state diagram that has both Mealy and Moore-type outputs.

2.8.2.5 Non-Important FSM Outputs

While there are times when you may need to generate a “complete” state diagram, you must remember that the state diagram is primarily meant for a human viewer. Combine this notion with the fact that even a modest sized FSM can have enough external inputs and outputs to quickly compromise the readability of the state diagram.

There are generally many outputs from a FSM, but the state diagram does not necessarily need to assign a value for every output in every state. If in any state a given output is not assigned, it is assumed to be a “don’t care” in the context of that state, which means that output does not affect the external operations associated with that state. You can thus omit outputs from a given state if those outputs don’t matter for that state. It is not necessarily bad practice to list all external outputs for each state, but your state diagram becomes harder to understand.

2.8.2.6 Non-Important FSM Inputs

The external input conditions control the state transitions of the FSM; these conditions must be mutually exclusive. This seems like we require a complete set of inputs for each transition and for every state, but this is not the case. In real FSMs, you’ll find that not all external inputs matter in every state. In those cases, we don’t need to include the inputs that don’t matter next to the state transition arrow. If we include the inputs that don’t matter, we make our state diagrams less readable.

The example state diagrams we’ve work with so far seem to indicate the FSM states are somehow limited in the number or transition arrows that can leave (or enter) the state. There is no limit, though we do need to ensure the conditions governing the transitions are mutually exclusive. There are a few key issues to be aware of regarding the transition arrows exiting a given state.

- Your state diagram must account for every possible set of external input conditions for every state. For example, if your FSM has “n” external inputs, every state must necessarily account for 2^n possible combinations of those inputs in order to completely specify the FSM. In reality, the 2^n is the worst-case scenario; you often find that not all inputs matter for all states.
- You must make sure that all conditions associated with the arrows leaving a given state are mutually exclusive, which means that no two arrows can have the same conditions. If two states had the same set of conditions, the FSM would know the correct transition.
- You can’t assume that an FSM stays in the same state if you don’t explicitly and completely specify all transition arrows leaving the state. This means that if there is a condition where the FSM does not transition to another state, it must indicate this condition with a self-loop, which explicitly states the associated conditions.

FSM are neither magical nor intelligent. FSMs do exactly what you design them to do. This means you must never allow the FSM to “make a decision” on its own. It’s quite easy to not completely specify a FSM and get a good feeling that the FSM is working properly in all of your testing. Inevitably, if you don’t properly specify the FSM, it will fail, and probably fail during a demo of your product to a potential buyer or investor.

2.8.2.7 The Final State Diagram Summary

Figure 2.36 provides a quick overview of the relation between the FSM black box and the example state diagrams we've been working with in this section. What you should be gathering from this diagram is that properly designed state diagrams have a particular structure and use a particular symbology.

- Singly directed arrows represent state transitions
- The FSM has external inputs that govern the state transitions from a given state
- Each transition arrow lists the external inputs that control its transition
- The state bubbles list the Moore outputs since they are only a function of state
- We list Mealy-type outputs with the external inputs (and hence the state transitions) since they are a function of both the present state and the external inputs.

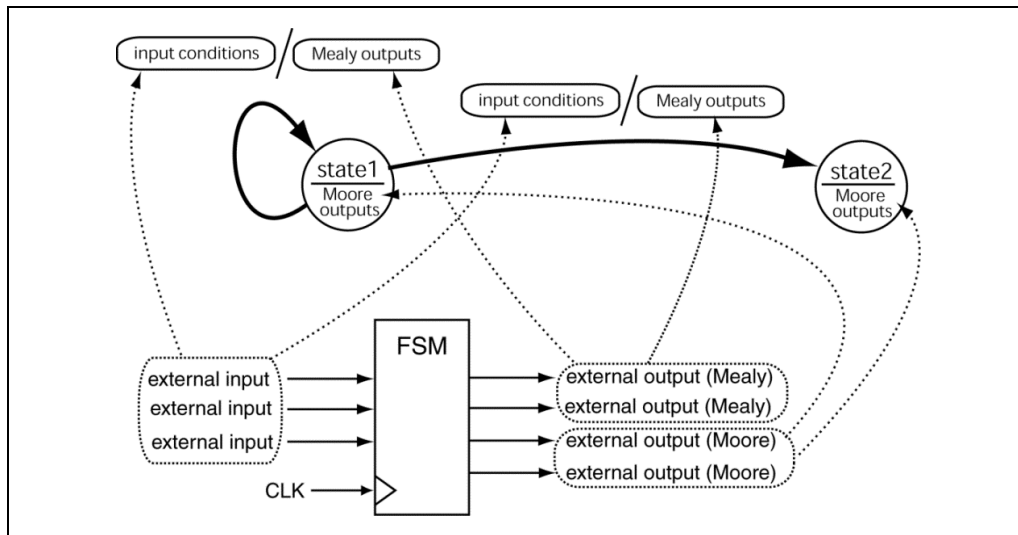


Figure 2.36: The relation between the state diagram and the high-level FSM.

The good news is that once you understand FSMs, and traverse the associated learning curve, you'll agree that there is not much to them. Here is everything in a nutshell.

- The heart of the FSM is the state registers; the heartbeat of the FSM is the clock signal that controls the state-to-state transitions of the FSM.
- On each active clock edge, the state of the FSM can transition to the present state (self-loop) or transition to a different state.
- The next state is a function of the present state of the FSM and the external inputs, which form the inputs to the next-state decoder.
- The outputs of the next-state decoder are the inputs to the state registers and thus determine the next state of the FSM.
- The FSM's external inputs are generally status signals from the outside world.
- The FSM sends the control signals to the outside world via the output decoder.
- The external outputs from the FSM are a function of the state variables (Moore-type) or a function of both the state variables and the external inputs (Mealy-type).

2.9 Chapter Summary

- All digital circuits can be categorized as being either a combinatorial or a sequential circuit. Combinatorial circuits do not have memory and their outputs are a simple function of their inputs. Sequential circuits have the ability to store bit, thus making their outputs a function of the sequence of inputs.
 - All digital circuits, including the most complex digital circuits, comprise of a set of basic digital design modules. These modules include both combinatorial and sequential circuits.
 - The main combinatorial digital building block circuits are built from simple logic gates. These circuits include the following:
 - Half and full adders: circuits capable of adding two 1-bit values
 - Ripple Carry Adders: circuits comprised of half and full adders chained together to form “n-bit” adders.
 - Multiplexors: circuits used as signal selection circuits
 - Decoder: circuits used to establish a given relationship between the circuit inputs and output.
 - Comparators: circuits that compare two values and provides information regarding the relationship between the two input values; well known to be made with EXOR-type gates.
 - Flip-flops: one-bit synchronous storage elements
 - Registers: n-bit synchronous storage elements
 - Counters: n-bit register “with features” that output a count “sequence”
 - Shift Registers: n-bit register “with features” that do fast division or multiplication by two.
 - Finite State Machines (FSMs): circuits that have sequential and combinatorial elements typically used as controllers for other digital circuits for counters with special count sequences. The general model of an FSM as a circuit controller is that inputs to the FSM provide status information from modules external to the FSM while FSM outputs represent control signals that are used to control modules external to the FSM.
 - State Diagrams are used to visually model the operation of FSMs. State diagram use their own special symbology to describe FSMs.
-

2.10 Chapter Exercises

- 1) In your own words, briefly describe what we mean by the term “digital bag of tricks”.
 - 2) Briefly describe the main differences between a combinatorial and sequential circuit.
 - 3) In your own words, describe the relation between memory of a sequential circuit and the notion that the outputs are a function of the “sequence” of inputs to the circuit.
 - 4) Briefly describe what characteristic gives a circuit the ability to store bits.
 - 5) Briefly describe the difference between a half adder and a full adder.
 - 6) Briefly explain whether it would be possible to construct a ripple carry adder using only half adders.
 - 7) Briefly describe how a “ripple carry adder” was given such a name.
 - 8) Briefly explain why is the “ripple carry adder” considered a slow adder?
 - 9) We consider the carryout output of an RCA to be a status output; Briefly describe how you could use the carryout as a data output.
 - 10) At any given time, how many AND gates in a multiplexor are not dead? Briefly explain your answer.
 - 11) In your own words, briefly describe the difference between a generic decoder and a standard decoder.
 - 12) Briefly describe the relationship between LUTs and generic decoders.
 - 13) What is the primary purpose of a parity generator?
 - 14) What basic digital component do parity generators, parity checker, and comparators all share.
 - 15) Briefly describe the difference between a flip-flop and a latch.
 - 16) Briefly describe the differences between a Mealy and Moore-type FSM.
 - 17) Briefly describe the purpose of a state diagram.
 - 18) Briefly describe the relationship between the number of states in a state diagram and the minimum number of bits in the associated FSM’s state registers.
 - 19) Briefly describe why the conditions associated with transitions leaving a state bubble must be mutually exclusive.
-

2.11 Chapter Design Problems

1) Design a circuit that continually outputs the following sequence:

{...0, 2, 4, 6, 8, 10, 12, 14, 0, 1, 2, 3, 4, 5, 6, 7, 0, 2, 4, 6, 8, 10, 12, 14, 0, 1, 2, 3...}

- Use a counter controlled by an FSM in your solution
- Provide a state diagram describing the FSM controlling the circuit
- Minimize hardware and the number of states in your FSM

2) Design a circuit that, upon pressing a button, continually outputs the following sequence:

{0,1,2,3,3,4,5,6,7,7,8,9,10,11,11,12,13,14,15,15,0,1,2,3,3,...}

- Use the up counter shown below in your circuit as well as an FSM, but don't add any other hardware. In other words, your circuit should contain only two components: a FSM and the counter shown below.
- Provide a state diagram describing the FSM controlling the circuit.
- Don't connect the button directly to the counter.
- The button asynchronously clears the counter

3) Design a circuit that, upon pressing a button, continually outputs the following sequence:

{0,0,1,2,3,4,4,5,6,7,8,8,9,10,11,12,12,13,14,15,0,0,1,2,3,...}

- Use the up counter shown below in your circuit as well as an FSM, but don't add any other hardware. In other words, your circuit should contain only two components: a FSM and the counter shown below.
 - Provide a state diagram describing the FSM controlling the circuit
 - Don't connect the button directly to the counter
 - The button asynchronously clears the counter
-

3 Advanced Registers

3.1 Introduction

The most commonly used circuit in digital design is the “register”. We’ve already used the term quite often in this text, particularly regarding finite state machines (FSMs). Recall that a main component of FSM was the storage associated with the state variables. This chapter describes registers with extra feature and some of their many various flavors and incarnations. Most of the description appearing in this chapter is at a higher-level as the low-level details are somewhat cumbersome and not overly useful.

Main Chapter Topics

- **SIMPLE REGISTERS AND REGISTERS “WITH FEATURES”:** This chapter defines and describes basic including registers with extended features that make them more useful in digital circuits.
- **TRI-STATE DEVICES AND TRI-STATE REGISTERS:** This chapter describes tri-state devices and their use in tri-state registers and associated circuitry.
- **BI-DIRECTIONAL REGISTERS:** This chapter briefly describes the notion of bi-directional registers and their relation to tri-state registers.
- **SHIFT REGISTERS:** This chapter describes various flavors of shift registers and their basic implementations as well as their common extensions and associated operations.

Why This Chapter is Important

This chapter is important because registers and their simple variations are extremely useful and thus often found in just about all meaningful digital designs.

3.2 Registers: The Most Common Digital Circuit Ever?

Stated as simply as possible, a register is nothing more than a multi-bit flip-flop. Flip-flops are single bit storage elements while registers multi-bit storage elements modeled as a given number of flip-flops sharing the same clock signal. When we say, “register”, we typically mean “simple register”; this works well as the more specialized registers have their own names. A later section introduces more advanced registers.

Figure 3.1 shows four D flip-flops assembled such that they act as a register; Figure 3.1(a) shows the block diagram for a 4-bit register and Figure 3.1(b) shows the underlying circuit. Here are a few things to note about Figure 3.1:

- The block diagram in Figure 3.1(a) shows a clock signal but also assumes other characteristics. Since we model the register with D flip-flops, there must be an active clock edge not shown in Figure 3.1(a). Unless otherwise stated, registers are generally active on the rising-edge of the clock, which is what Figure 3.1(b) shows.
- Figure 3.1(b) shows that each flip-flop in the register shares the same clock. The result is that all the flip-flops latch their data simultaneously.

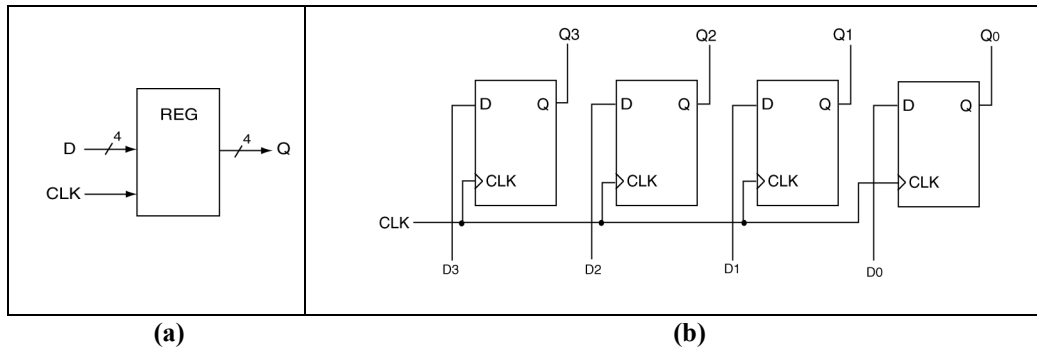


Figure 3.1: A block diagram for a 4-bit register (a), and the lower-level implementation details of a 4-bit register (b).

3.3 Tri-State Registers

Although the underlying theme of digital design is the notion of binary signals, there is one other common and useful “state” in digital-land¹. Certain digital devices have the ability to have a third output in addition to the standard ‘1’ and ‘0’. We refer to these devices as “tri-state” or “three-state” devices², because these devices have a third output known as the “high-impedance” state. The best way to refer to think about these devices is not to consider these devices as having a third state, but rather to think about these devices as having a magic switch that either allows the device to operate normally or kills the device altogether.

The notion of high-impedance is common in both analog and digital design. There are many ways out there to model high impedance devices, but I prefer to model them using Ohm’s Law: $V=IR$, with V representing voltage, I representing current, and R representing resistance. For this discussion, we can consider impedance the same thing as resistance. If we rearrange Ohm’s Law, we obtain the $R=V/I$, which states that the resistance is directly proportional to the voltage and inversely proportional to the current. In digital circuits, the voltage is generally constant so we’ll only consider R and I . For the R value to be large implies that the I value to be small. When I is small means that there is little current flowing in a circuit. Digital circuits require current in order to operate, so a circuit with high-impedance means the circuit has low current, which implies the circuit is dead. Yet another way to model high-impedance is as a switch that turns off the current to a circuit; an open switch is the same as an open circuit or broken circuit, which implies the circuit is dead.

There are many great reasons out there for you to kill your digital circuit. The two major reasons in digital design are to 1) save power, and 2) give your circuit the ability to share resources. The notion of sharing resources is important and useful.

Although there are many tri-state-type devices out there, we can best explain them with a simple buffer. Figure 4.7(a) shows a tri-state buffer; this circuit is simply a buffer with a control input. The control input in Figure 4.7(a) is the “EN” input; this input controls whether the output of the device is in a high-impedance state or not. Another way to think of the EN input is as a switch that either turns on or turns off the circuit. Note that because of the way we drew the circuit in Figure 4.7(a) that the control input is active high; had the “EN” input included a bubble, the control input would be active low.

Figure 4.7(b) shows a truth table that describes the operation of the tri-state device. Note that Figure 4.7(b) uses the term “Z” to represent high-impedance³. Figure 4.7(b) states that the buffer output is in a high-impedance state when the “EN” input is not asserted ($EN=0$) or the circuit is operating normally (outputs of 1’s and 0’s) when the “EN” input is asserted. Figure 4.7(c) shows a compressed truth table describing the circuit. Figure

¹ It’s not really a state though...

² The difference between “tri-state” and “three-state” is that some company trademarked one of these terms. Be careful how you use the terms; lawyers are waiting in the wings.

³ The term “Z” is how both digital and analog electronics represent high-impedance.

4.7(b) and Figure 4.7(c) indicate that the EN (enable) input essentially enables the input to appear on the output of the device as.

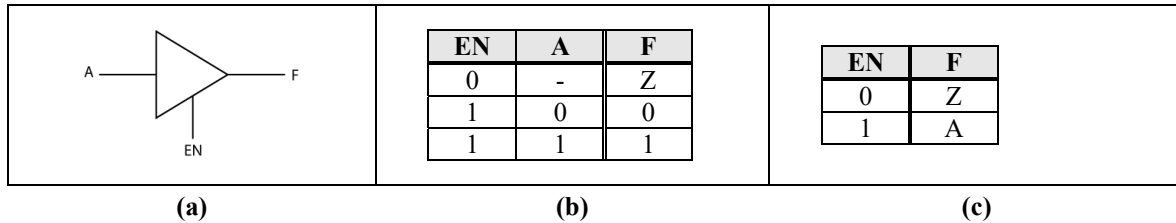
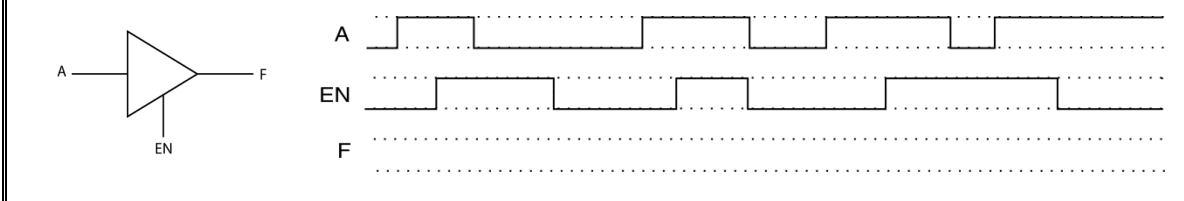


Figure 3.2: A tri-state buffer (a) and associated truth tables in full and compressed form (b) and (c).

Example 3.1: Tri-State Buffer Timing Diagram

Use the following tri-state buffer diagram to complete the following timing diagram.



Solution: Figure 3.3 shows two different forms of the solutions to Example 3.1. In reality, there are many different ways to represent high-impedance. What you'll find out in digital-land is that every datasheet and every simulator represents high-impedance in different ways; the two approaches in Figure 3.3 are two of the more popular approaches.

The upper F timing in Figure 3.3 uses bundle-related notation for showing when the signal is high-impedance. Note that when the EN signal is not asserted, the F output is in the high-impedance state; when the EN signal is asserted, the A input appears on the F output. The lower F timing shows the same characteristics as the upper one, but the timing diagrams shows the high-impedance output with a signal that is neither high nor low. For single signals (as opposed to bundles), the lower version of the F timing is more common. Even better, devices such as simulators that display these types of outputs typically use colors to represent the signal values such that the high-Z output is a different color than the normal digital signal.

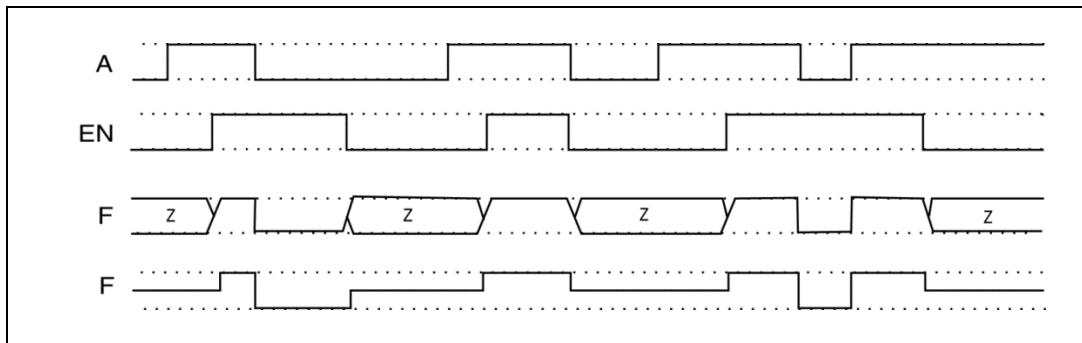


Figure 3.3: Two equivalent solutions to Example 3.1.

The notion of tri-stating applies to many digital devices. The notion of “tri-stating” is a feature of a device and thus does not come free. When your particular circuit requires a tri-state device, then you use one; otherwise, you avoid using a device with the tri-state feature to save costs. The tri-stating needs of a circuit are most often

associated with circuits that share resources in an effort to reduce overall circuit size and/or costs. One particularly common tri-state device out in digital-land is the tri-state register.

The term tri-state register refers to the notion that you can place each of a register's outputs into a high-impedance state. The tri-state control input associated with a tri-state register always controls the register's output in a parallel manner. In other words, the tri-state control places either all of the circuit's output in high-impedance state when the control is asserted, or all the registers output are in a digital state when the control input is not asserted. Figure 3.4 shows a circuit diagram for a typical tri-state register. We know this device is a tri-state register because of the triangle adjacent to the OUT signal. We also know that since this is a tri-state device, the EN signal is what controls whether the output is hi-Z or a normal digital output.

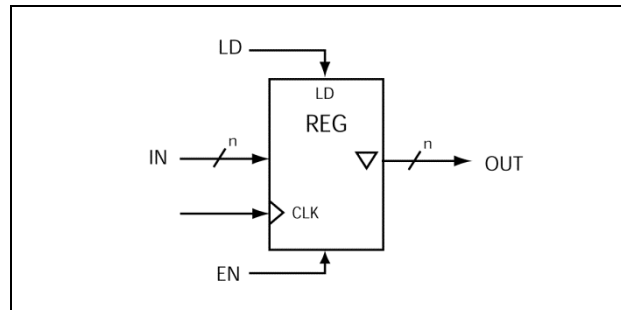
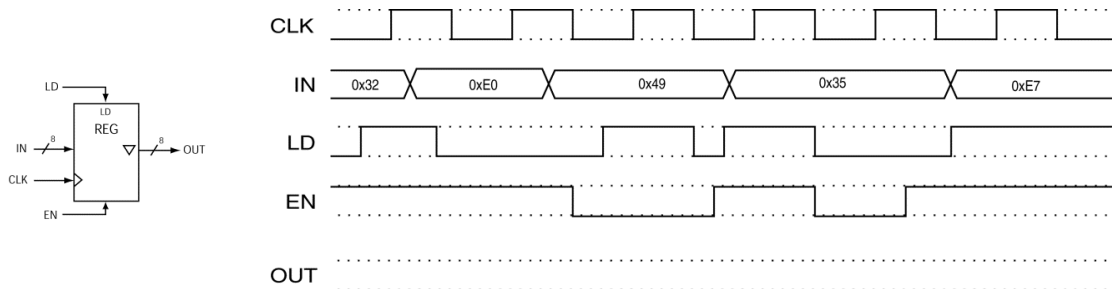


Figure 3.4: A schematic diagram of a basic tri-state register.

Example 3.2: Tri-State Register Timing Diagram

Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.



Solution: Figure 3.5 shows the solution to Example 3.2. There are several particularly important things to note about the solution in Figure 3.5.

- Anytime the EN input is not asserted, the OUT signal is in its high-Z state. We arbitrarily represented the high-Z state with “ZZZ”, which you should not equate with the fact that this problem is boring.
- The LD signal is effectively independent from the output. In this way, the register still loads the IN signal into the register regardless of whether the EN signal is asserted or not. This event occurs during the third rising-edge of the clock in Figure 3.5.

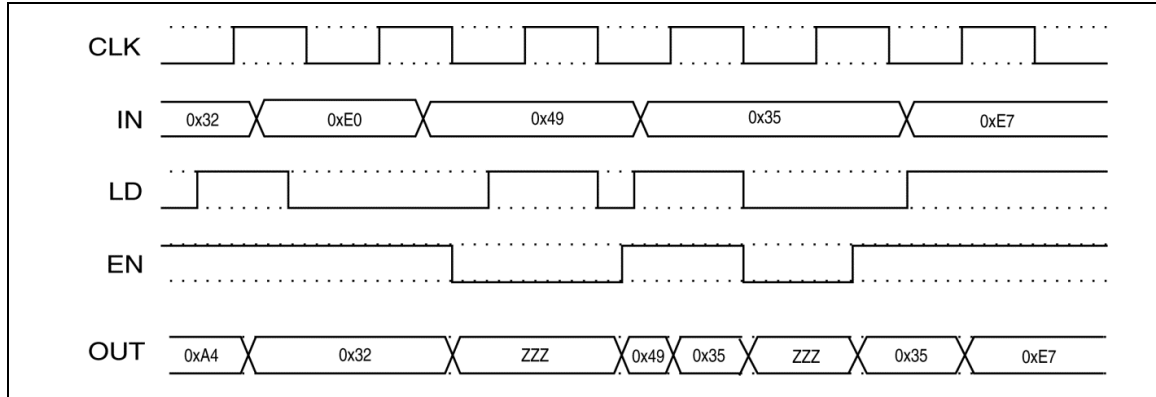


Figure 3.5: The solution to Example 3.2.

One of the reason tri-state registers exist is to save resources. This is a topic we generally save for advanced digital design, but we'll mention it here in case you never advance digitally. Aside from that lame attempt at humor, the notion of using tri-state registers for resource sharing brings up a massively important point which every digital designer needs to know.

As an example of resource sharing, Figure 3.6 shows two tri-state registers in same circuit. Note in Figure 3.6 that there is a connection between the outputs of the two registers. Because these two registers are sharing the same routing resources, and because both of these devices have the ability to “drive the bus”, a potential problem exists. Enabling both registers are simultaneously creates a situation we refer to as “bus contention”. Bus contention occurs when two more output devices (registers in this case) simultaneously drive their data onto the same lines bus line. Bus contention results in indeterminate circuit behavior and is thus something you should avoid. For example, if one output device drives the bus with all 1's and another device drives the bus with all 0's, what would some input device see on these lines? Who knows!⁴

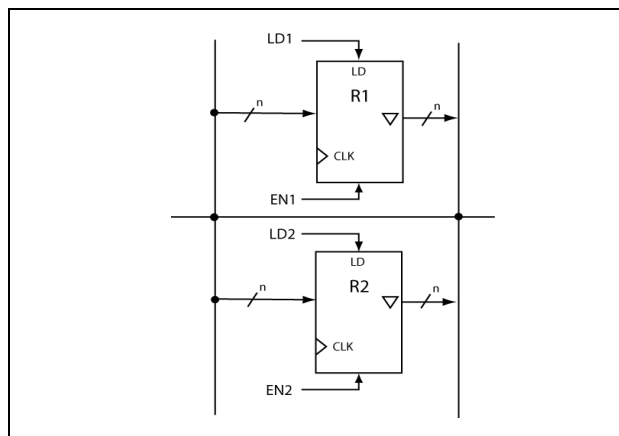


Figure 3.6: A schematic diagram of a basic tri-state register.

Working with circuits that share resources in this way certainly creates a new aspect to digital design. But all is not lost; the way to avoid bus contention is to make sure that no more than one output device is driving the bus lines at any given time. The way to “drive the bus” is to assert the enable input on the given device. Recall that when the tri-state outputs are not asserted, the device is essentially removed from the circuit as the devices outputs are providing no current to the circuit.

3.4 Bi-Directional Registers

A discussion on registers would not be complete without a description of bi-directional registers. In a continued effort to save resources, some registers use a single bundle (or bus) to route the data into and out of a register. These registers retain the required control signals including load control, tri-state control, and a clock, but they share the input and output lines. Figure 3.7(a) shows a schematic diagram of a typical bi-directional register; Figure 3.7(b) shows the same register drawn on a lower level to show some of the pertinent device implementation details. There are a few items in Figure 3.7(b) worth noting.

- Figure 3.7(a) represents the bi-directionality of the device with the doubly directed arrow for the Q bundle. In this way, the Q bundle can be either an input or an output depending on the EN control signal.
- The diagram uses the standard “tri-state” upside-down triangle in conjunction with the double directed arrow to officially represent the bi-directionality of the device.
- Figure 3.7(b) does not include the tri-state symbol. Figure 3.7(b) shows that you can model a bi-directional register as a standard register with a tri-state buffer on the output. The notion with this circuit is that the enable signal (EN) effectively prevents the register from driving its data to the outside world when the EN signal is not asserted. However, despite the EN signal being unasserted, the register can still latch any data that some other circuit is driving onto the data lines.

Bi-directional registers are similar to tri-state registers, but they do have a subtle difference. In tri-state registers, the enable input either drives the output or places the output into high-Z mode. In bi-directional registers, the device’s enable either drives its data onto the shared resource when the enable signal is asserted, or the device is “listening” to the shared resource when the enable signal is not asserted.

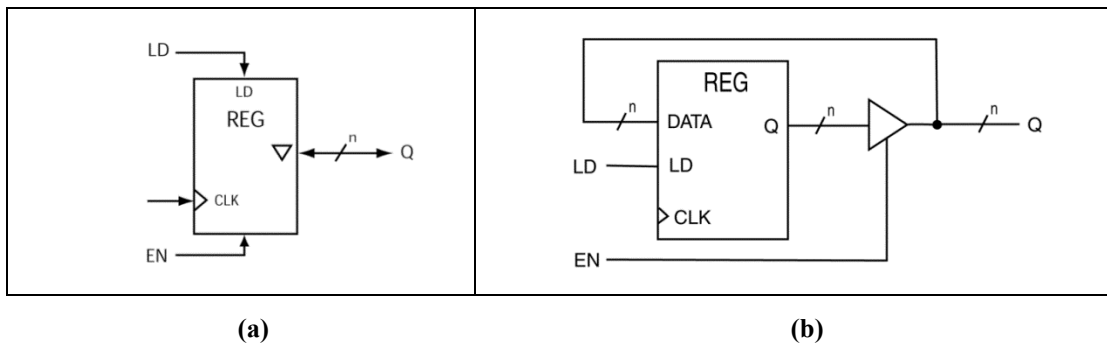


Figure 3.7: A circuit diagram for a bi-directional register (a), and the same bi-directional register drawn on a lower level (b).

Once again, this is a slightly advanced subject so we won’t provide much more than a mention of some of the bi-directional device’s functionality. We’ll leave this subject with one final diagram. The notion of tri-stating and bi-directionality saves routing resources in a circuit, sometime at the cost of losing some flexibility in the circuit. Figure 3.8 shows two functionally equivalent circuits that advertise this resource sharing. Figure 3.8(a) shows a circuit with two register with tri-state outputs while Figure 3.8(b) shows a circuit with two registers with bi-directional outputs.

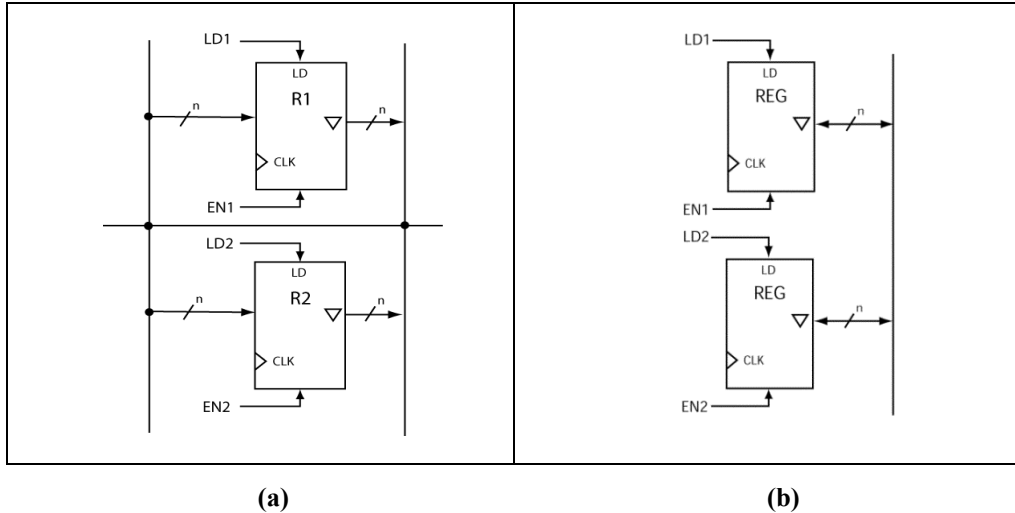


Figure 3.8: Two functionally equivalent circuit models: (a) is the tri-state version of the circuit while (b) is the bi-directional version of the circuit.

3.5 Shift Registers

Basic shift registers are typically a circuit introduced in an introductory design course. They are a commonly used digital circuit because of their ability to do integer math simply and quickly. More specifically, a single left or right shift in a shift register performs a multiply or divide by two, respectively; these operations are done one per clock cycle. The notion here is that multiplication and division in digital circuits often requires large and complex circuitry; shift registers perform multiplication and division quickly, but at the cost of only being able to multiply or divide by two.

We can extrapolate the operations of shift registers by noting that they can multiply or divide by powers of two. In simple shift register, division by integral powers greater than one require extra clock cycles as shift registers only perform one shift per clock cycle. Also worthy of noting here is that division by two (right shifts) cause a truncation of the original data as one bit of the original shift register contents is lost per right shift. In summary, shift registers don't do a lot, but what they do, they do really well.

3.5.1 Basic Shift Registers

Figure 3.9 shows a comparison of block diagrams for a simple 4-bit register and a basic 4-bit shift register⁵. The important thing to notice from these diagrams is that the simple 4-bit register generally deals with “parallel” data while the basic shift register generally deals with “serial” data. What you'll find later in this chapter is that the definition of these devices starts to overlap as we add more features to the devices.

⁵ Keep in mind that the block diagrams show only the very basic devices for comparison purposes, which hopefully is somewhat instructive.

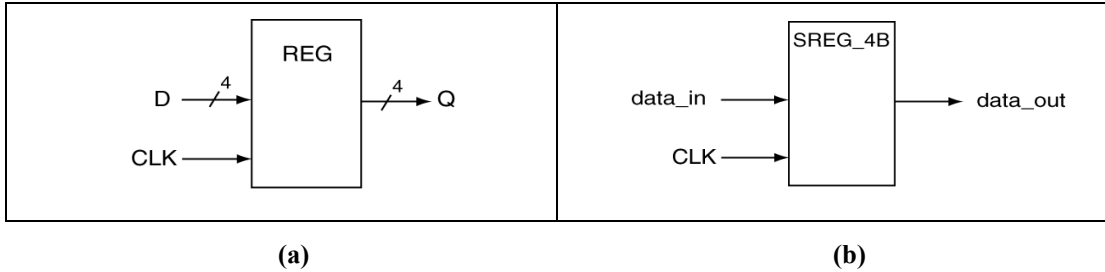


Figure 3.9: A block diagram for a 4-bit simple register (a) and a basic 4-bit shift register (b).

The operation of a shift register is simple but can be somewhat tricky when you first encounter it. Figure 3.10(a) shows a schematic diagram of a 4-bit shift register while Figure 3.10 (b) shows a model of the underlying circuitry. There is not a lot to say about Figure 3.10 as the fun stuff begins when you examine a timing diagram associated with this circuit.

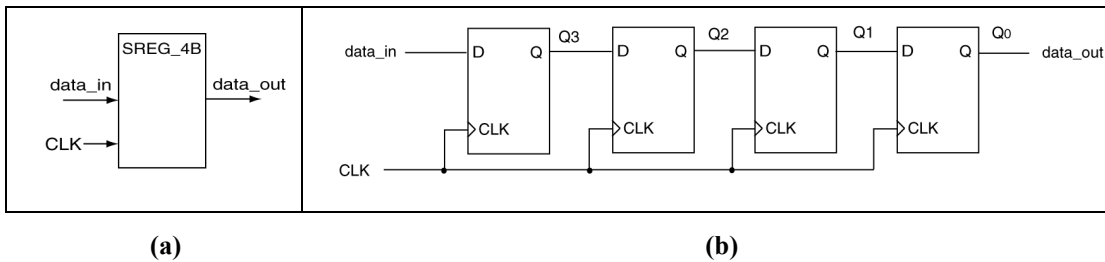


Figure 3.10: A block diagram for a 4-bit simple register (a) and a model of the underlying circuitry of a 4-bit shift register (b).

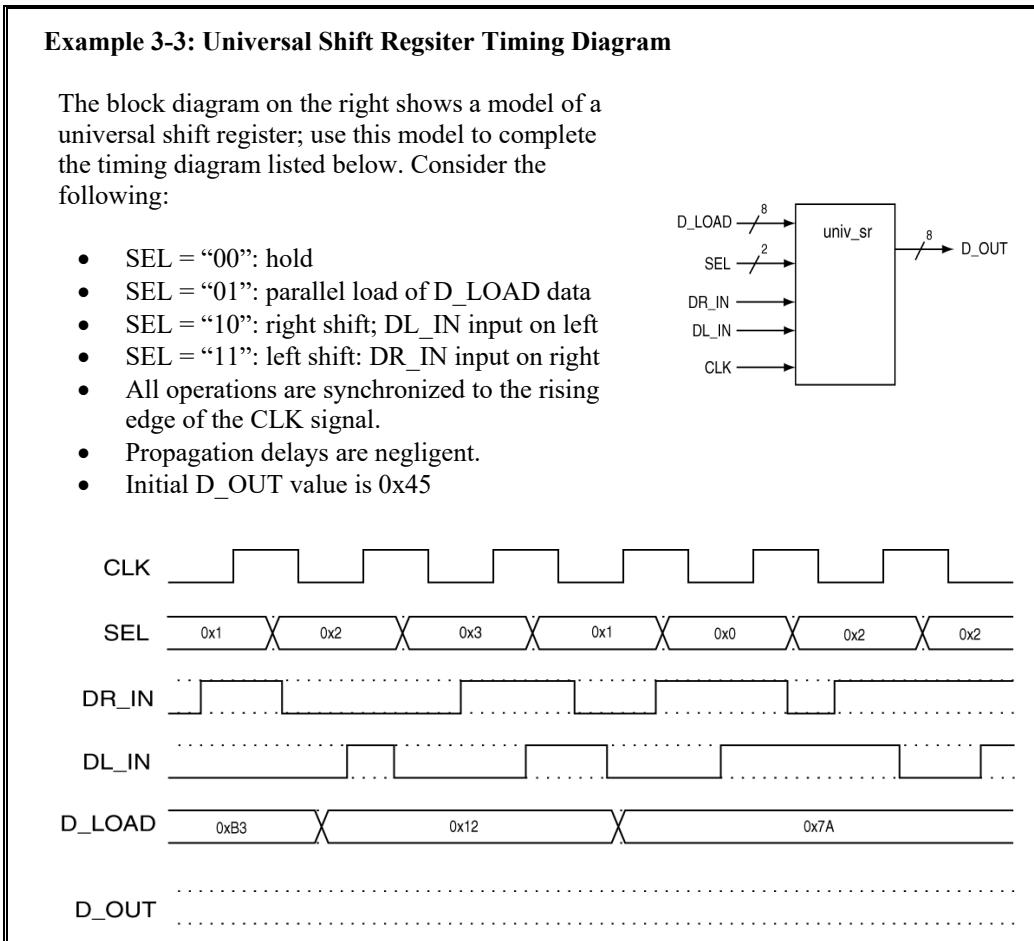
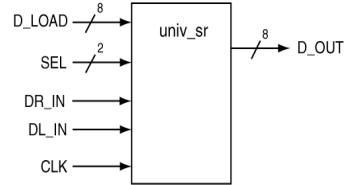
3.5.2 Universal Shift Registers

Shift registers that only shift in one direction are not overly useful in digital-land. Most shift registers do many more operations such as shift left, shift right, parallel load, parallel clear, hold (don't change state), pick up the spare, etc. The term in digital-land for shift registers containing features such as these is “universal shift register”, or “USR”. There is no one definition for universal shift registers; the only thing the term means is that you're dealing with some sort of shift register that does more than shift in one direction. From that point, you need to consult the datasheet or designer as to what exactly the device does.

Example 3-3: Universal Shift Register Timing Diagram

The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:

- SEL = "00": hold
- SEL = "01": parallel load of D_LOAD data
- SEL = "10": right shift; DL_IN input on left
- SEL = "11": left shift; DR_IN input on right
- All operations are synchronized to the rising edge of the CLK signal.
- Propagation delays are negligent.
- Initial D_OUT value is 0x45



Solution: The first step in any problem involving a sequential circuit is to establish the initial state of the storage elements. This problem states that the initial value of D_OUT value is 0x45; this value is the initial state of the shift register.

From there, a good approach to problems such as these is to list what actions the SEL signal is selecting throughout the timing diagrams. Figure 3.11 shows a partially annotated timing diagram highlighting the operations selected by the SEL signal. Note that we synchronize all annotations with the rising clock edge.

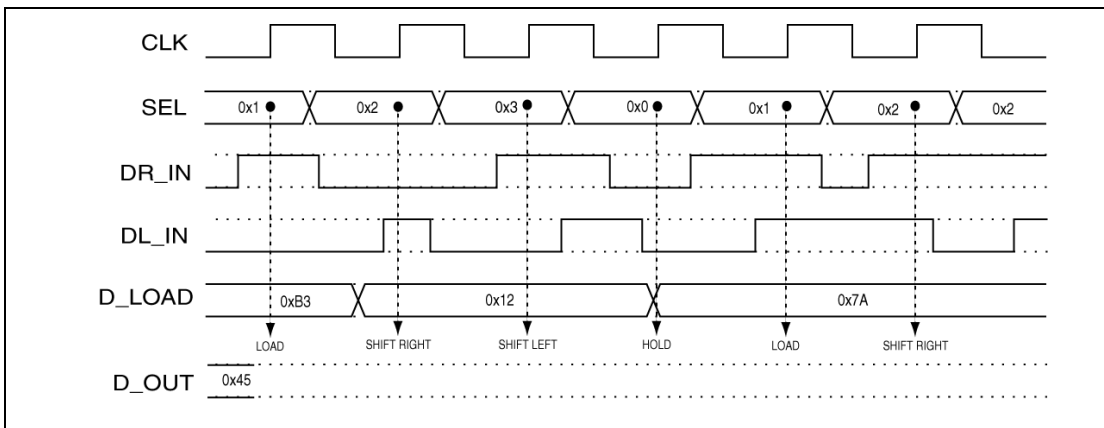


Figure 3.11: A black box diagram of the universal shift register.

Figure 3.12 shows the final timing diagram. As you can see, most of the changes in the DR_IN, DL_IN, and D_LOAD signals have no effect on the final output. The important thing to do for this problem is to verify for yourself that each of the values in the D_OUT is correct.

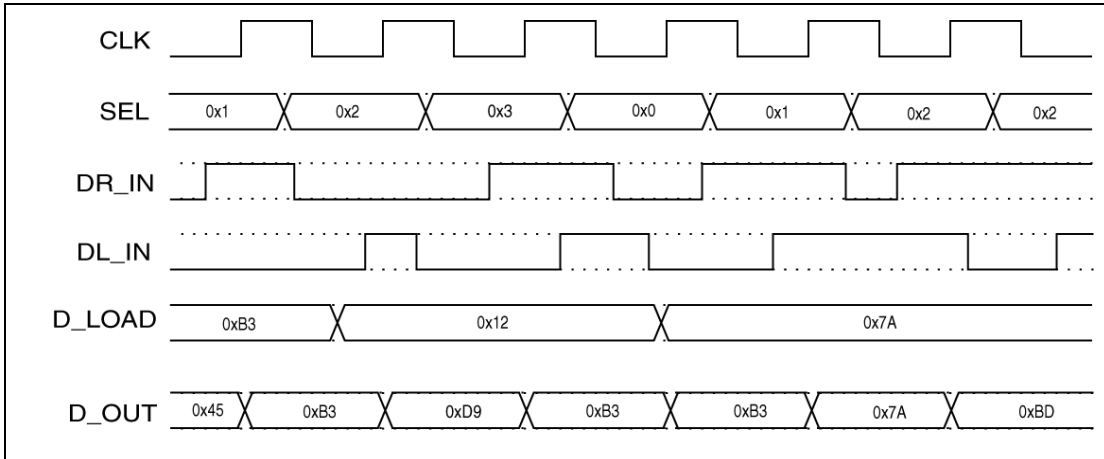


Figure 3.12: A black box diagram of the universal shift register.

3.5.3 Barrel Shifters

One of the common shifting-related operations out there is a “barrel shift”. The operation of barrel shifters is straightforward as it’s simply an extension of simple shifting operations. While simple shift registers only performed one shift per clock cycle, barrel shifters are effectively capable of performing more than one shift per clock cycle. As you would imagine, barrel shifters can shift either left or right.

The key to understanding barrel shifters is realizing the main reason they exist. Keep in mind that shift registers contain “bits” which generally represent binary numbers. The notion of shifting left and right are associated with multiplying by two (left shift) or dividing (right shift) by two. Thus, barrel shifters are then associated with multiplying and dividing by “powers of two” (such as 4, 8, 16, 32, etc.). What these operations provide are super-fast (namely, one clock cycle) multiply and divide operations. As you continue in digital stuff and/or computer programming, you’ll find that multiplying and dividing binary numbers is relatively time consuming relative to other computer operations (such as logic operations). Barrel shifters provide a cheap and fast, although somewhat limited alternative.

We commonly use barrel shifters in arithmetic applications where we do not require 100% accuracy of results. For example, there is always a big push to have your circuit perform “integer-based math” because working with integers is much less “computationally expensive” than working with other options such as “floating point numbers”. A good example of this is with non-professional cameras such as the ones on your cell phones. Because we partially judge cameras on these devices by their operational speed (such as how fast you can take pictures⁶), they generally use integer math. Using integer math causes you to lose some precision, but your eyes will never know the difference. All you know is that your tiny hand-held device is able to take high definition movies and do so without significant delay. Big wup.

Table 3.1 shows two examples barrel shifting operations. Both of these examples use an 8-bit value; the top example is the value before the active clock edge while the bottom value is the value after the active clock edge. The examples show both a starting and ending point for the barrel shifting operation described by the particular row in the table. The (a) row shows a 2x right barrel shift that arbitrarily inputs 0’s on the left side of the register. The (b) row shows a 2x left barrel shift that arbitrarily inputs 1’s from the right side of the register. The operation in the (a) row represents a divide by two; the operation in the bottom row is one the many open mysteries in this world.

⁶ In reality, there is a significant amount of processing taking place for even the most basic digital photograph.

	Description	Example
(a)	barrel shift right 2x; stuff in a two 0's from the left side.	
(b)	barrel shift left 2x; stuff in a two 1's from the right side.	

Table 3.1: Examples of possible barrel shifting operations.

The examples in Table 3.1 are arbitrarily barrel shift of “2x”. This syntax refers to the notion that the barrel shifter is “shifting two times” in one clock cycle. The truth is that it is only shifting one time, which implies there are connections each shift register element and the element that is two shift register elements away from the current element. As you can probably imagine, the barrel shifter requires the proper signal routing in order to accomplish this shift. As a result, barrel shifters out in digital-land are typically limited by the different flavors of barrel shifts (such as “2x”) and shift directions that they can perform. Barrel shifters in these applications are typically associated with specific mathematical operations and truly don’t have the general need to perform every possible shift length. Recall that for every barrel shift requires extra routing resources, which are generally not cheap in digital-land.

3.5.4 Other Shift Register-Type Features

Two more of the common shifting operations are rotates and arithmetic shifts. These operations are also simple in their basic states⁷. Rotate operations can be useful in many applications, though there is not one slam-dunk great example I can think of; in theory, these operations fall into the category of “bit tweaking”. Arithmetic shift operations are similar to simple shift operations but can work better with signed binary numbers.

Rotate operations include rotate left or a rotate right with the actual shifting occurring on the active clock edge. The notion with rotate-type shifts is that no bits from the original register values are lost by “shifting them out” of the register as was the case with simple shift registers. Specially, for a rotate right operation, the LSB of the register becomes the new MSB while all other bits are shifted one position to the right. For a rotate left operation, the MSB of the register becomes the new LSB while all other bits in the register are shifted one position to the left.

	Description	Example
(a)	rotate right; the LSB is transferred to the MSB;	
(b)	rotate left; the MSB transfers to the LSB.	

Table 3.2: Examples of rotate-type shifts.

⁷ The truth is that it can get really ugly out there. You may need to combine operations with as “barrel rotates” or “barrel arithmetic shift”, or some type of shift to enhance your bowling skills. We won’t go there in this chapter.

Arithmetic shifts are similar to simple shifts in their ability to perform mathematical operations⁸. The key difference is that arithmetic shifts work with signed binary number and preserved the “signedness” of the value they operate on. For an arithmetic shift left operation, the value of the sign bit does not change because of the shift. Thus, the left shift operation retains the sign of the number as well as the ability to perform fast multiplication with the left shift operation. For an arithmetic shift right operation, we both retain the sign bit as a sign bit and propagate the sign bit to the right with each shift. This sounds somewhat strange, but it truly both retains the sign of the value in the register as well as performing a fast division operation. I suggest working through a few examples on your own.

	Description	Example
(a)	An arithmetic shift right of a positive number in 2’s complement form; the operation copies the sign bit from sign-bit position to the next bit on the right with each shift. This is a divide by two on a signed number (positive).	<p>start: 0 0 1 1 0 1 0 0</p> <p>1st shift: 0 0 0 1 1 0 1 0</p> <p>2nd shift: 0 0 0 0 1 1 0 1</p>
(b)	An arithmetic shift right of a negative number in 2’s complement form; the sign bit is copied from sign-bit position to the next bit on the right with each shift (the sign bit remains unchanged). This is a divide by two on a signed number (negative).	<p>start: 1 0 1 1 0 1 0 0</p> <p>1st shift: 1 1 0 1 1 0 1 0</p> <p>2nd shift: 1 1 1 0 1 1 0 1</p>
(c)	An arithmetic shift left on a positive value in 2’s complement form. The left shift does not alter the sign; all other bits shift left and the operation arbitrarily stuffs a ‘0’ into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a signed number (positive).	<p>start: 0 0 1 0 0 1 0 0</p> <p>1st shift: 0 1 0 0 1 0 0 0</p> <p>2nd shift: 0 0 0 1 0 0 0 0</p>
(d)	An arithmetic shift left on a negative value in 2’s complement form. The left shift does not alter the sign bit; all other bits shift left and the operation arbitrarily stuffs a ‘0’ into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a signed number (negative).	<p>start: 1 0 1 0 0 1 0 0</p> <p>1st shift: 1 1 0 0 1 0 0 0</p> <p>2nd shift: 1 0 0 1 0 0 0 0</p>

Table 3.3: Examples of many flavors of arithmetic shifts.

⁸ When you read this paragraph, recall that we represent signed binary numbers using 2’s complement notation, AKA, “diminished radix complement” notation.

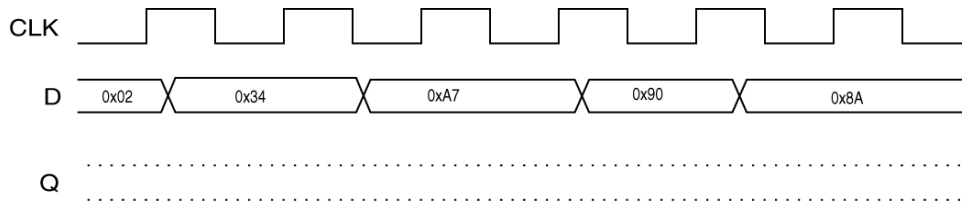
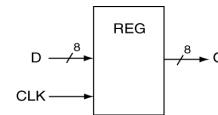
3.6 Chapter Summary

- **Registers:** A register is a sequential circuit that can be considered nothing more than a parallel combination of single-bit storage elements. These storage elements are modeled as a given number of D flip-flops that share a common clock signal and possibly other control signals typically associated with D flip-flops (such as pre-set and clear signals). The register is typically used to “latch” (and thus remember) an n-bit wide set of data on the active clock edge of the device.
 - **Tri-State Registers:** Tri-state registers contain tri-state buffers on the register’s output. The tri-state registers effectively allow the register to either place its data onto a shared routing resource with the tri-state outputs enabled, or effectively remove itself from the circuit altogether with the tri-state outputs disabled. When the tri-state register’s outputs are disabled, the circuit is “high-impedance”, or “high-Z” state. An extra input signal is typically used to control the circuit’s tri-state outputs. The driving notion behind tri-state registers is to share, and thus save circuit routing resources, but come at the expense of overall circuit flexibility.
 - **Bi-Directional Registers:** Bi-directional registers are registers that are tri-state registers that are configured at a low-level to appear to have shared input and output lines. Bi-directional registers also represent attempts to save circuit routing resources.
 - **Shift Registers:** Shift registers are in many ways similar to simple registers; their primary difference is with the inputs to the individual shift register storage elements. Shift registers are designed such that the data output from one shift register element becomes the data input to a contiguous element. In this way, data is said to be “shifted through” the shift register. In general, there is one “shift” per clock cycle. Shift register operations are often used to implement fast but limited mathematical operations with single right shift being a divide-by-two and a single left shift being a multiply by two.
 - **Universal Shift Register:** A type of shift register that performs more operations than a simple shift register. These operations can typically include both a shift left and a shift right, a parallel load, a preset and/or clear. Somewhere in here could also be arithmetic shift operations and various forms of rotate operations.
 - **Barrel Shifters:** A type of shift register that performs multiple shifts on a single clock edge. In reality, barrel shifters are wired such that they can shift multiple bit locations in one clock cycle, and probably do not perform multiple shifts. Barrel shifters are useful for mathematical operations including multiplication and division by powers of two.
-

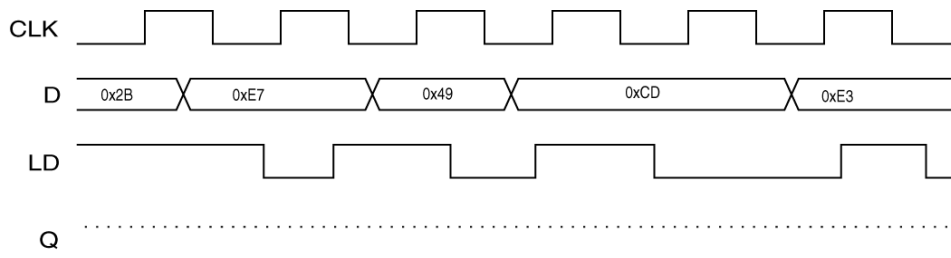
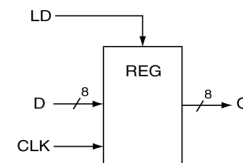
3.7 Chapter Exercises

- 1) List three different types of registers.
- 2) The notion of a latch is generally associated with a one-bit storage element. Briefly describe whether it is possible to have a multi-bit latch, and briefly describe the difference between a multi-bit latch and a register.
- 3) Briefly describe whether you can discern whether a register's control inputs are synchronous or asynchronous from looking at a schematic diagram.
- 4) Briefly describe why the third state in a tri-state register is not really a state.
- 5) Briefly explain the main benefit of using tri-state devices in your circuit.
- 6) Briefly explain the notion of using shared resources in a digital circuit.
- 7) Shift registers are known for doing "efficient integer math". Briefly explain why this is so.
- 8) Briefly explain why universal shift registers have no real solid definition.
- 9) Briefly explain why the hardware footprint for barrel shifter is larger than the footprint for a simple shift register.
- 10) Briefly explain the notion of bits being lost with a shift operation but not being lost with rotate operation.
- 11) Briefly explain whether it would be possible to use an arithmetic shift on an unsigned number.

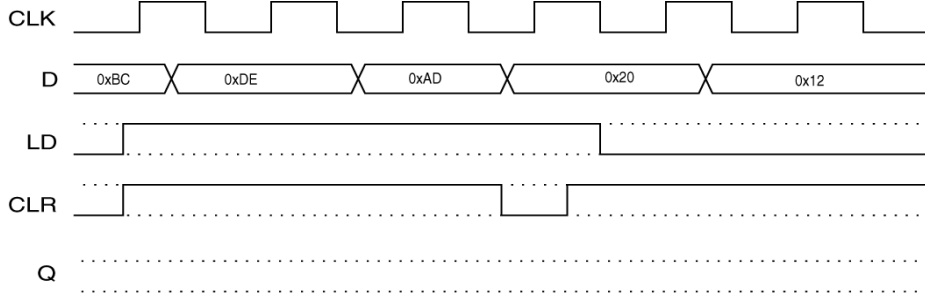
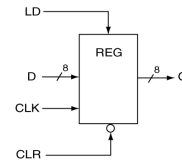
- 12) Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



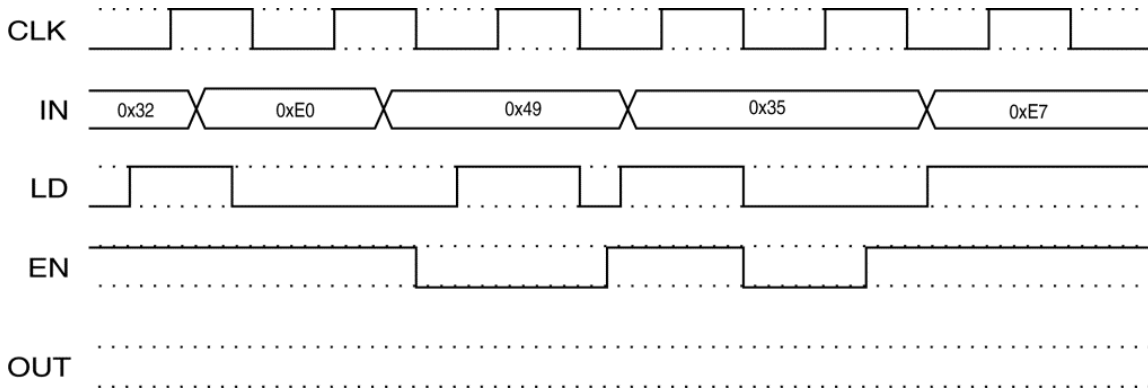
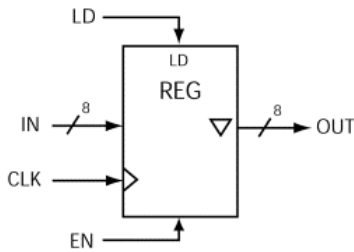
- 13) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



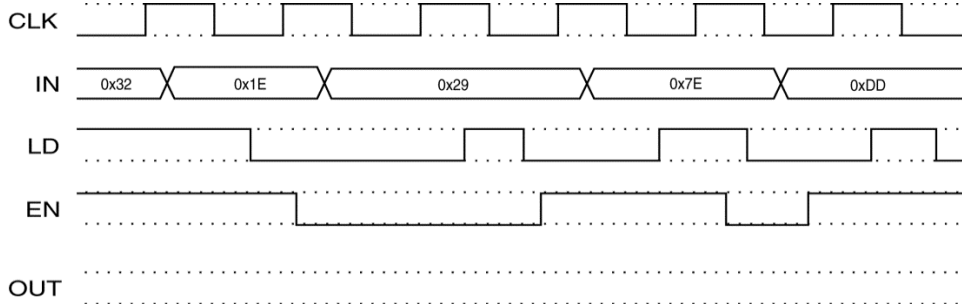
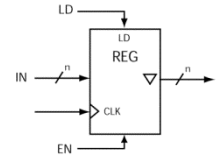
- 14) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. The CLR input is an asynchronous input that clears the register when asserted and has a higher precedence than the LD input. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



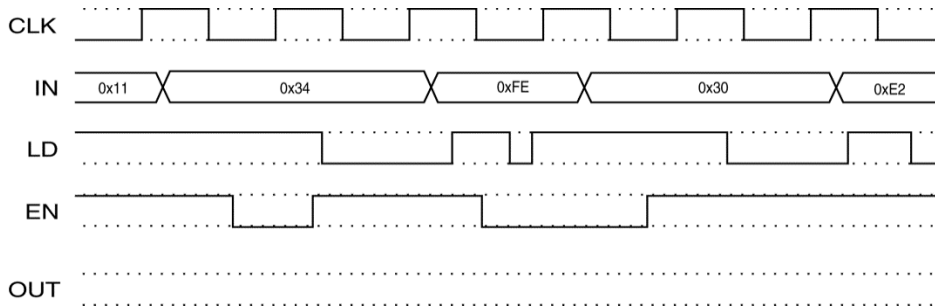
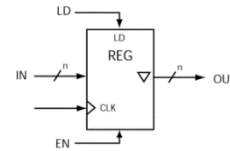
- 15) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.



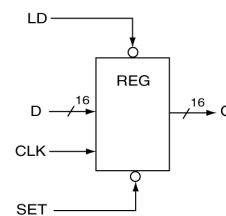
- 16) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xBA. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



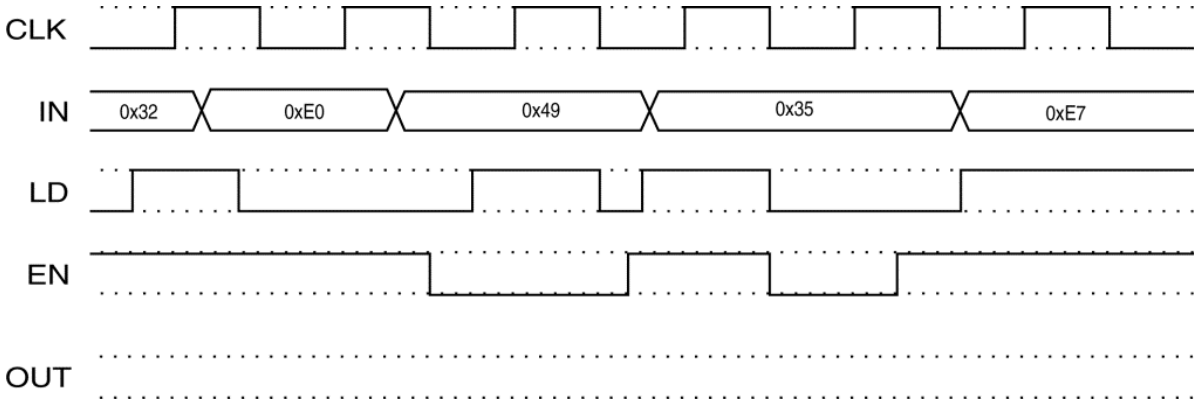
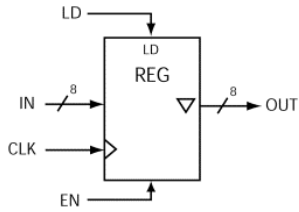
- 17) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xBA. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



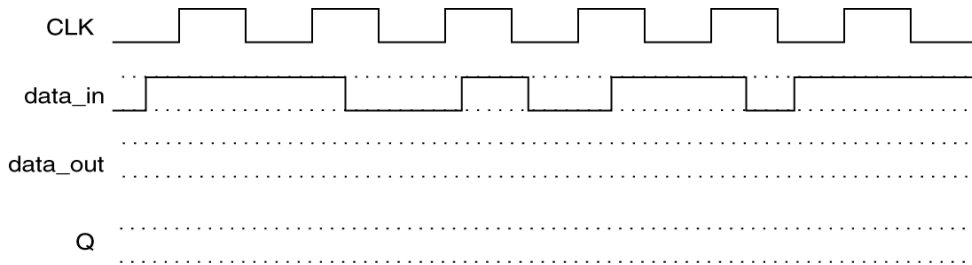
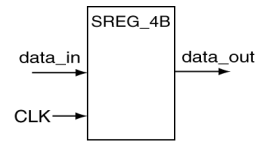
- 18) Using the block diagram on the right, provide a schematic diagram detailing how you would use this device to create a 32-bit register with all the same features listed on the 8-bit device.



19) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.

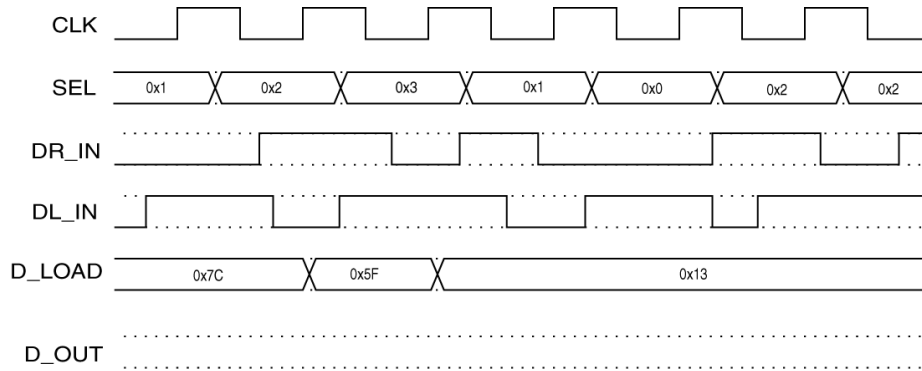
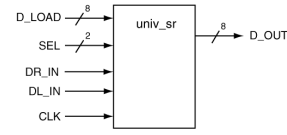


18) Use the block diagram on the right to complete the timing diagram below. Consider the circuit to be a 4-bit shift register (shifts from right-to-left) that is active on the rising-edge triggered of the clock signal. Consider the line labeled “Q” to represent the 4-bit value stored by the shift register and the “data_out” output to represent the value of the highest order bit stored by the shift register. Assume the initial value stored by the shift register is 0xC. Ignore all propagation delay issues with this circuit



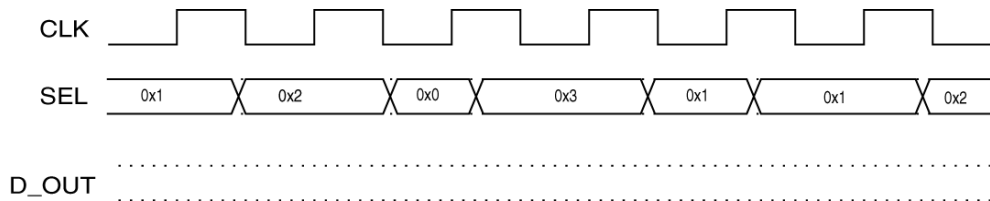
19) The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:

- SEL = "00": hold
- SEL = "01": parallel load of D_LOAD data
- SEL = "10": right shift; DL_IN input on left
- SEL = "11": left shift; DR_IN input on right
- The rising edge of the CLK signal synchronizes all shift register operations
- Propagation delays are negligent.
- Initial D_OUT value is 0xAB

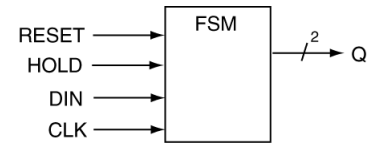


20) Complete the following timing diagram using the following USR characteristics. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need to make in order to complete this problem. Assume the 0x39 is the initial value stored by the shift register. Assume "D_OUT" is an 8-bit output representing the value stored by the shift register.

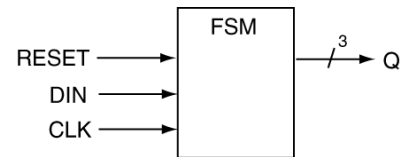
- SEL = "00": rotate right
- SEL = "01": rotate left
- SEL = "10": divide by 8 (bit stuff 0's)
- SEL = "11": multiply by 8 (bit stuff 0's)



- 21) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 2-bit shift register. Consider the Q output to be a 2-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs. When the HOLD input is asserted, the Q output does not change.



- 22) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 3-bit shift register. Consider the Q output to be a 3-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs.

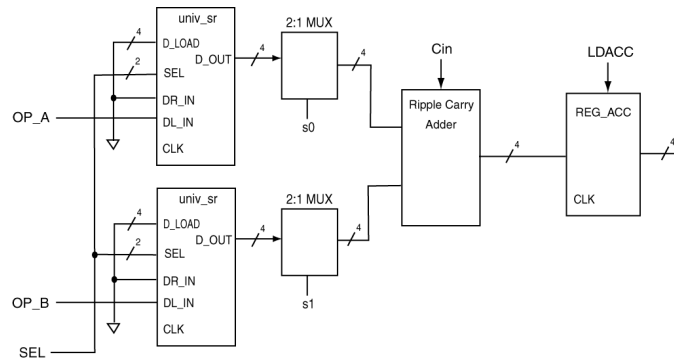


23) The following diagram shows a circuit that is used to perform a serial-to-parallel conversion on the OP_A and OP_B input and then perform a mathematical operation. In other words, two four-bit numbers will be provided serially (LSB first) on the OP_A and OP_B inputs. The two tables below describe the MUXes and the Universal Shift Register (USR).

- Provide a state diagram that could be used to control the circuit such that it performs $A - B$ and registers the result in REG_ACC (A & B are the parallelized versions of the OP_A & OP_B serial data). The serial to parallel conversion will initiate when the signal GO (not shown) is asserted. Minimize the number of states in your design. State any other assumptions you deem necessary.

<u>Assumptions:</u>	
<p style="text-align: center;"><u>MUX description</u></p> <pre> if (sx = 0) then out <= in; else out <= not in; end if; </pre>	<ul style="list-style-type: none"> • LSB is first to arrive in serial bit stream • DR_IN = right side input to shift register • DL_IN = left side input to shift register • CLK signals are connected • All setup and hold times are met • All Shift register operations are synchronous

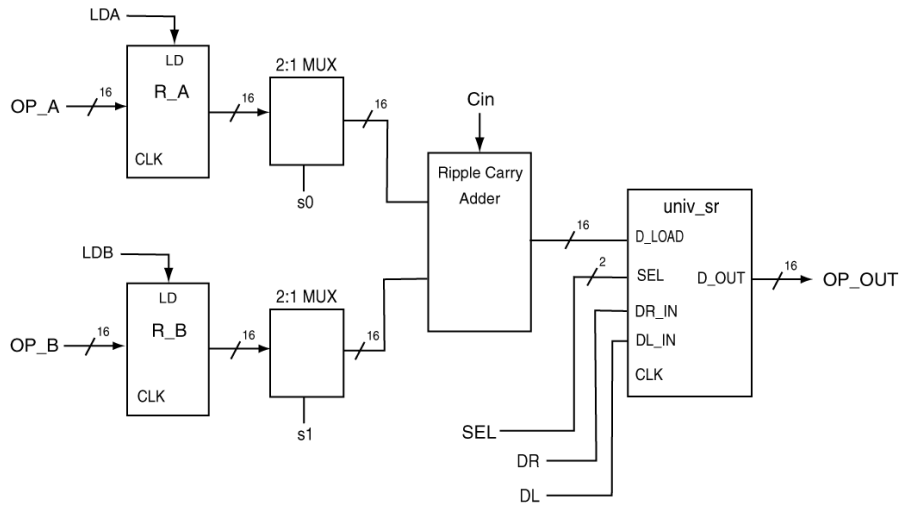
Shift Register Controls	
SEL	Operation
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



24) The following diagram shows a circuit that can perform a mathematical operation. The two tables below describe the MUXes and the Universal Shift Register (USR). The registers have a synchronous load input (LD). Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.*

- If a GO signal is received (GO is not shown in diagram), the following operation is generated and the result appears on the output: $OP_OUT = (OP_B - OP_A) \div 16$

<u>MUX description</u>	<u>Assumptions:</u>	<u>Shift Register Controls</u>										
<pre> if (sx = 0) then out <= in; else out <= not in; end if; </pre>	<ul style="list-style-type: none"> • DR_IN = right side input to shift register • DL_IN = left side input to shift register • CLK signals are connected • All setup and hold times are met • All Shift register operations are synchronous • Registers (non-USR) have synchronous load inputs (LD) 	<table border="1"> <thead> <tr> <th style="text-align: center;">SEL</th> <th style="text-align: center;">Operation</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0 0</td> <td style="text-align: center;">hold</td> </tr> <tr> <td style="text-align: center;">0 1</td> <td style="text-align: center;">parallel load</td> </tr> <tr> <td style="text-align: center;">1 0</td> <td style="text-align: center;">shift right</td> </tr> <tr> <td style="text-align: center;">1 1</td> <td style="text-align: center;">shift left</td> </tr> </tbody> </table>	SEL	Operation	0 0	hold	0 1	parallel load	1 0	shift right	1 1	shift left
SEL	Operation											
0 0	hold											
0 1	parallel load											
1 0	shift right											
1 1	shift left											



PART TWO: Advanced Digital Design

ACADEMONIC

© 2018 mealy

ADMINISTRATIVE
FOOD INSECURITY

I WONDER IF ALL THE
OVERPRICED CAMPUS
EATERIES COULD BE
GENERATING MORE
PROFITS...



STUDENT
FOOD INSECURITY

I WONDER IF I'll
HAVE ENOUGH MONEY
TO BUY FOOD...



4 Chapter: Register Transfer Notation

4.1 Introduction

Digital design is always abstracting things upwards in an effort to increase the efficiency of representing circuits. As we move on describing computer circuits, we need to come up with a new, higher-level of abstraction for representing circuits. The solution to this dilemma is what we refer to as *register transfer language (RTL)* or synonymously, *register transfer notation (RTN)*. This notation, or language, uses a simple syntax that provides a clear and concise description of a circuit. A set of register transfer language (RTL) statements can completely describe a digital system in a high-level manner, which is why it is so useful in computer design. Conversely, we can also describe a digital system by a set of RTL statements.

Main Chapter Topics

- **REGISTER TRANSFER NOTATION INTRODUCTION:** This chapter introduces the notion of register transfer notation (RTL). This chapter also discusses the motivations behind RTL and the most accepted syntax or RTL form.
- **MICROOPERATIONS:** This chapter classifies and describes basic operations that you can do with register and their relation to elementary operations.
- **DATA TRANSFER CIRCUITS:** This chapter describes three main types of data transfer circuits and provides examples of their usage, advantages, and disadvantages.

Why This Chapter is Important

This chapter is important because register transfer notation is highly useful in designing and/or describing computer operations because it provides a compact form to describe data transfers and the signals that control them.

4.2 Register Transfer Notation Specifics

Before we go here, there is one important fact that you need to keep in mind. RTL is not like an HDL in that it is not a compiled or interpreted language. With an HDL, there are many syntax-type rules you need to follow in order for your code to synthesize. The same is not true for RTL: the rules (if there really are any at all) are lax. A good analogy to this lack of rules is with the labeling of the inputs, outputs, and states of the state diagrams. The guiding principle in drawing state diagrams was to simply make it readable and understandable to anyone who has some idea of what the state diagram is modeling. Similar to state diagrams, since there is not absolute syntax that you can draw upon, so you must be clear with the convention you use to write RTL equations. Equation 4. shows an example of the general form of an RTL statement.

$$[\textit{conditions} :] \textit{destination register} \leftarrow \textit{source register} [\textit{,destination register} \leftarrow \textit{source register}, \dots]$$

Equation 4.1: The general form of a RTL statement.

The notation in Equation 4. reads as follows: *the contents of the source register is transferred to the destination register*. Here are the important points to realize regarding this notation:

- There can be conditions associated with these transfers (as indicated by the italics in the far left of Equation 4.) which allow the transfers to occur. We aptly refer to the left-pointing arrow as the replacement operator.
- There can be multiple transfers associated with one RTL statements.
- A clock signal is rarely (if ever) included in RTL statements. The transfer is understood to occur on the active clock edge associated with the system, thus the system clock synchronizes all microoperations.
- The result of this transfer does not generally change the contents of the source register (and if it did, the RTL statement would list it).
- The register transfer operations listed in one RTL statement happen in parallel. In the context of digital circuitry, this means all the transfers happen on the same system clock edge.

Example 4.1

Draw a circuit that would implement the following RTL statement: $R1 \leftarrow R2$

Solution: Once again, there is an interesting relationship between a RTL statement and the underlying hardware. This problem tells you what needs to be done and it is your job to design a circuit that does it. The RTL statement provides a guideline on what the underlying hardware should be able to do. If the hardware you generate can do it, you've got a right answer, but certainly *not the only answer*. In other words, there are generally many solutions to a given problem such as this one. There is usually a preferred solution based on the most efficient circuit so you should always strive for that option.

The thing to notice about the given RTL statement is that it lists two registers. Your final circuit therefore has at least two registers. Also, note that there needs to be a path so data can flow from the R2 register to the R1 register. These two facts spell out the answer to the example. Churn them around in your head and you'll arrive at the circuit shown in Figure 4.1(a). The width of the data signals has been arbitrarily set to eight for this and subsequent examples.

The data line labeled **A** represents the output of the R2 register from Figure 4.1(a). Since this signal is a bus, we use the shorthand notation to represent all of the signals on the bus as listed in Figure 4.1(b). The "0x" notation is C programming language notation that indicates the numbers that follow it are in hexadecimal format (thus representing the eight bits of the signal). Figure 4.1(b) shows that we represent the state of the eight bits on the signals labeled **A** and **B** with this notation. Figure 4.1(b) shows that the signals change on each clock edge. We show this dependency by using the arrows pointing from the rising clock edge to the changing data in the **B** signal. The values on the **A** signal are arbitrary as are the times they change; what's more important is that you understand the timing and data transfers. Note that the value of the **A** signal changes midway between the two clock pulses but the new condition is not transferred to the **B** signal until the rising clock edge comes along.

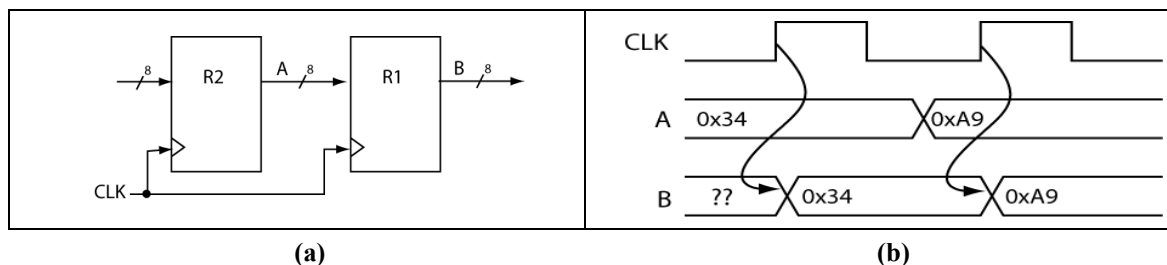


Figure 4.1: Solution and timing diagram for Example 1.

Example 4.2

Draw a circuit that would implement the following RTL statement: $C1: R1 \leftarrow R2$

Solution: This problem is similar to the previous problem but with a slight modification in the RTL statement. Note that in this RTL statement, there is a dependency. In other words, data is transferred from the output of R2 into the R1 register only if the C1 signal is asserted. Most RTL statements have some type of dependency but most are more complex than this as you'll see in the final example. The circuit in Figure 4.2(a) provides the functionality specified by the given RTL statement. Note that the R1 register contains a LD input, which enables the parallel loading of data into R1 on the active clock edge.

The timing diagram shown in Figure 4.2(b) is more instructive for several reasons. First, the C1 input is somewhat dependent upon the clock. The thought here is that the active clock edge causes a change in some other circuit that has an output that is currently driving the C1 input. Imagine that this signal is a Moore-type output from some FSM (control unit). Note that both the rising and falling edges of C1 are synchronized with the clock edge (with some delay included). The state of the C1 signal at the first clock edge is low so the data is not loaded from R2 to R1. Remember, both the C1 signal needs to be high and the rising edge of the clock must be present in order for the load to occur. The data on the A signal is arbitrary; the initial value of the B signal is arbitrarily placed in an unknown state but becomes known after the rising clock edge.

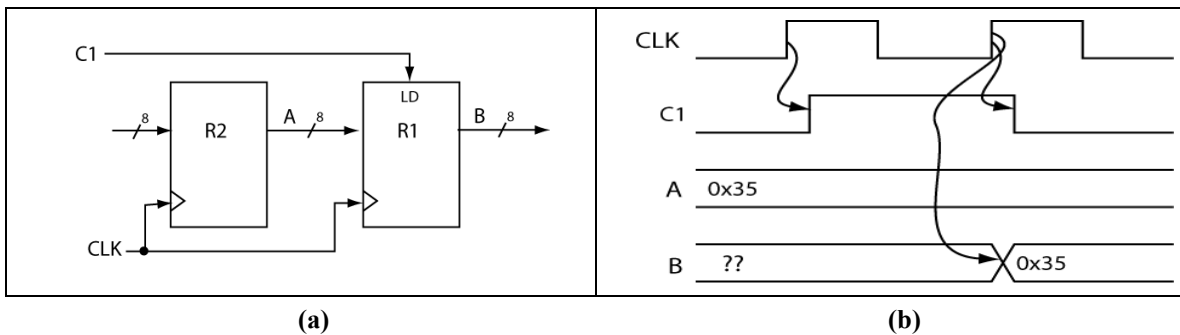


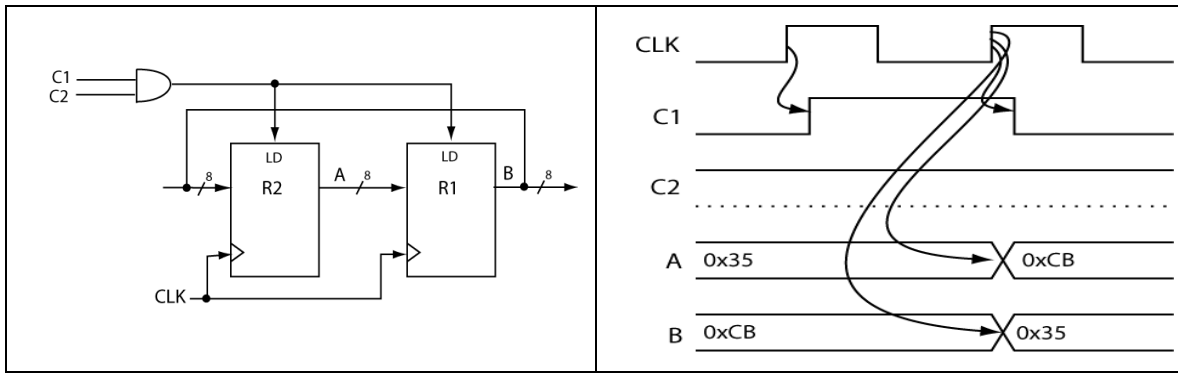
Figure 4.2: Solution and timing diagram for Example 2.

Example 4.3

Draw a circuit that implements the following RTL statement: $C1, C2: R1 \leftarrow R2, R2 \leftarrow R1$

Solution: This problem is slightly different from the previous problem. This type of RTL statement shows that more than one data transfer can happen simultaneously as indicated by the comma-separated equations on the right side of the colon. The left side of the colon indicates a more complex condition that allows the data transfers to happen. Figure 4.3(a) shows the circuit having this functionality. Note that the comma-separated conditions of "C1,C2" say that both C1 and C2 need to be asserted in order for the transfers to occur. The *and* in this statement can be nicely implemented as an AND gate as shown in Figure 4.3(a). Figure 4.3(b) shows an accompanying timing. This circuit swaps the data between the R2 and R1 registers.

One other important matter to concern yourself with in Figure 4.3(b) is the relation between the CLK signal and the C1 signal. The diagram lists that the state change in the C1 signal is caused by the CLK signal. The underlying and unspoken detail here is that some other circuit in the system (that is not listed) is going to change the state of the C1 signal. In other words, the C1 signal could be considered the output of some FSM that is subsequently a function of another unmentioned input.



(a) (b)
Figure 4.3: Solution and timing diagram for Example 3.

Example 4.4

Draw a circuit that is able to implement the following RTL statements. Assume you have a standard n-bit register available to you that has a LD (load) input.

$$C1 \oplus C2 : R2 \leftarrow R1 + R2$$

$$C1, C2 : R1 \leftarrow R1 + R2$$

Solution: The best approach to take when approaching these circuits is to start listing what you know about the problem. We list the things you should realize about this problem below; Figure 4.4(a) shows the final circuit.

- The “+” operator on the right side of the colon represents addition. If this operator had appeared in the left side of colon, it would have represented an OR operator. The presence of an addition operator implies that you have some hardware capable of performing the operation. In this case, the hardware is a simple *adder*, such as an RCA. The typical adder adds two n-bit numbers and outputs the results.
- The circuit requires two registers. You know this because you see that there is an R1 and an R2 but no other registers.
- The output of each register is going to be added. This means that the outputs of the registers must be the inputs to the adder.
- The result of the addition must be made available to the inputs of both the R2 and R1 register. This means the adder output is a source that has two destinations: the input of the R1 and R2 registers.
- There is some extra controlling logic required to enable the loading under the appropriate conditions. This includes the AND gate and an EXOR gate.

Figure 4.4(b) shows a timing diagram associated with the circuit solution of Figure 4.4(a). There are a few things to notice in this diagram:

- All transitions occur on the rising clock edge.
- Output data from the adder is transferred to the R2 register on the first rising clock edge (and not the second rising clock edge) because the conditions of C1 and C2 satisfy the loading logic for the register (the XOR gate).

- Output data from the adder is transferred to the R1 register when the state of signals C1 and C2 satisfy the logic for the load input of the R1 register (second rising clock edge only).

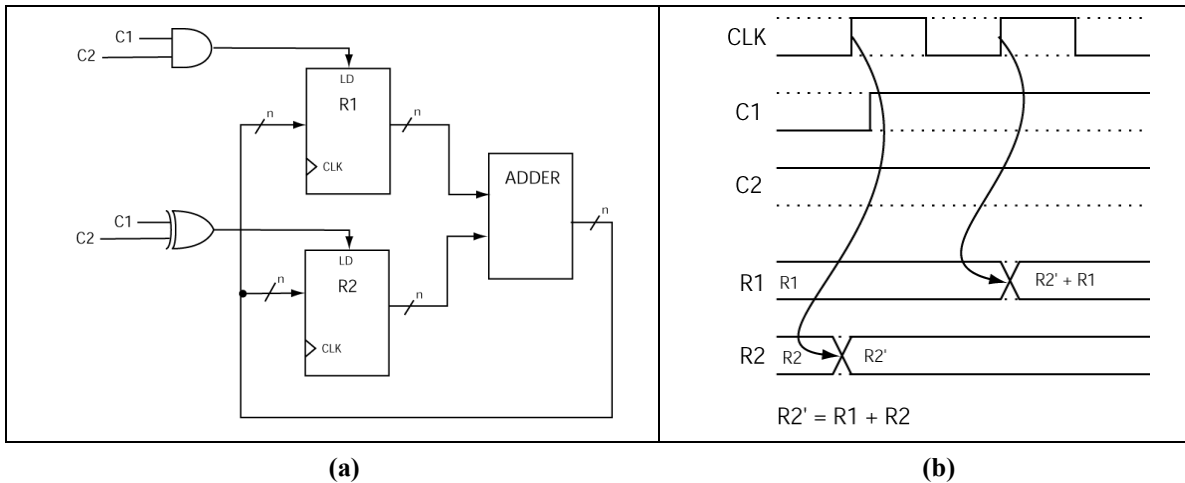


Figure 4.4: The solution for Example 1 (a) and an associated timing diagram (b).

4.3 Microoperations and Data Transfers

Microoperations are elementary operations that are performed on data stored in one or more registers. Note that the registers themselves have the ability to perform elementary operations. When a register performs one of these elementary operations, it is considered to be performing a microoperation. With this definition, we can state that any time data stored in a register changes, it is the result of a microoperation. For example, the data in a simple register changes when we clear the register or load a new value. Another example, then the output of a counter is incremented, it is a result of a microoperation. This means that the control inputs to our register-type circuits (simple registers, counters, and shift register) control what microoperations that particular circuit can perform.

Although we mentioned several types of microoperations in a previous chapter, we'll introduce more microoperations in this chapter and we'll divide them into specific types. The types we list are somewhat arbitrary in that they do not include every possible microoperation possible on any piece of hardware. Additionally, some of the microoperations we list can fall into more than one of the listed types. The following classification then is mostly for instructional purposes so don't try to read too much into it. The four major types of microoperations can be classified as follows:

- *Transfers* – data is not changed as data passes from one register to another
- *Arithmetic* – some arithmetic function is performed on data in registers
- *Logic* – some logical-type bitwise manipulation is performed on data in the registers
- *Shift* – the change in register data can be characterized by a shift in the data

4.3.1 Transfer Microoperations

Equation 4.2 shows a typical transfer microoperation represented by an RTL statement. In this equation, the contents of register R2 are transferred to register R1 under the condition that X is asserted. This transition, as are most all microoperation represented by RTL, is synchronized to some clock edge.

$$X : R1 \leftarrow R2$$

Equation 4.2: A typical transfer microoperation.

4.3.2 Arithmetic Microoperations

Table 4.1 shows some of the more popular arithmetic microoperations. The most important thing to remember about the microoperations listed in Table 4.1 is that writing the equation means that you can either currently perform the operation (the hardware, in this case some arithmetic circuit, exists) or you'll soon be able to perform the operation (you're designing the hardware capable of performing the given function).

Arithmetic Micro-ops	Worthy Comment	Hardware Possibilities
$!Cin : R1 \leftarrow R2 + R3$	Addition; source registers are not changed; assumes there is some circuitry that is capable of doing the addition; the values of R2 and R3 do not change.	The output of R2 and R3 is directed to the input of an adder. The output of the adder is connected to the input of R1.
$!Cin : R1 \leftarrow R2 + R1$	Addition; one source is destination; the value of R2 generally does not change.	The output of R2 and R1 are connected to the input of an adder. The output of the adder is connected to the input of R1.
$!Cin : R3 \leftarrow R3 + R3$	Addition; doubling circuit	The output of R3 is connected to both inputs of an adder. The output of the adder is connected to the input of R3.
$Cin : R2 \leftarrow R3 + \overline{R4} + 1$	Subtraction; the 2's compliment thing ($R2 = R3 - R4$);	The output of R3 and the complimented output of R4 connects to the input of an adder. The Cin input of the adder (considering an RCA) is set to '1' and is included in the addition.
$R5 \leftarrow \overline{R5}$	Complement contents of R5 (1's complement)	The output of R5 feeds into a row of inverters; the output of the inverters feed back to the R5 inputs.
$Cin : R5 \leftarrow \overline{R5} + 1$	2's complement negation (multiply by -1); value is R5 becomes -R5	The compliment of R5 and '0' are connected to the inputs of an adder. The Cin input is set to '1'.
$R1 \leftarrow R1 + 1$	Increment R1; R1 register changes	The magic increment input of a counter.
$Cin : R1 \leftarrow R2 + 1$	Add 1 to R2 and store result in R1; the value of R2 does not change.	The output of R2 is added to 0 and the Cin input is a '1'.
$R1 \leftarrow R1 - 1$	Decrement R1; R1 Register changes	The standard decrement operation of a counter.
$R2 \leftarrow R1, R1 + 1$	Assign R1 to R2; the R1 value increments.	The output of counter R1 is latched to register R2. At the same time, the value in the R1 register is incremented.

Table 4.1: Some popular arithmetic microoperations.

4.3.3 Logic Microoperations

There are a handful of logic microoperations that provide useful tools for manipulating the data in registers. Logic operations are generally considered to be bitwise in nature, meaning that the associated logic operator is applied to each of the bits in the registers on a one-to-one basis. Table 4.2 shows some of the more common logic microoperations.

Logic Micro-ops	Worthy Comment	Hardware Possibilities
$R5 \leftarrow \overline{R5}$	Logical bitwise complement (1's complement); complement the current value of R5 and return the new value to R5; The current value of R5 changes.	The output of register R5 is complimented and fed to the inputs of R5.
$R5 \leftarrow \overline{R2}$	Logical bitwise complement (1's complement); complement the current value of R2 and store the result in R5; The current value of R2 does not change.	The output of register is complimented and becomes the input of the R5 register.
$R0 \leftarrow R1 \text{ AND } R2$	Logical bitwise AND of R1 and R2; the result is stored in R0; the current values of R1 and R2 generally do not change.	The output of R1 is ANDed with the output of R2; the result becomes the input to R0.
$R1 \leftarrow R1 \text{ AND } R2$	Logical bitwise AND of R1 and R2; the result is stored in R1; the current value of R2 generally does not change.	The output of R1 is ANDed with the output of R2; the result becomes the input to R1.
$R3 \leftarrow R1 \text{ OR } R2$	Logical bitwise OR of R1 and R2; the result is stored in R3; the current values of R1 and R2 generally do not change.	The output of R1 is ORed with the output of R2; the result becomes the input to R3.
$R1 \leftarrow R1 \text{ XOR } R2$	Logical bitwise Exclusive OR of R1 and R2; the result is stored in R1; the current value of R2 generally does not change.	The output of R1 is EXORed with the output of R2; the result becomes the input to R1.

Table 4.2: Some popular logic microoperations.

4.3.4 Shift Microoperations

Here is the list of basic shift-type operations:

1. **Simple shifts:** The simple shift would include single shifts in either the left or the right direction. We refer to this shift as simple because the shifts that follow are somewhat less simple.
2. **Rotates:** The rotate operations (rotate left and rotate right) either feeds the MSB to the LSB (on a left shift operation) or the LSB to the MSB (on a right shift operation). All other bits shift accordingly.
3. **Arithmetic shifts:** The arithmetic shift is for operations where the bits stored in the register are considered to be a signed number. In this case, the MSB is considered the sign bit and its present state must be preserved in both the left and right shift operations.
4. **Barrel shifts:** A barrel shift essentially performs more than one simple shift (in any one direction) in a single clock cycle. The distance of the barrel shift is arbitrary but is indicated in the RTL equation with the "Xx" notation (where the capital X represents the effective number of bit shifts). These shifts are actually quite useful since they provide a fast multiplication and division (depending on shift direction). The only catch here is that the divisions and multiplications need to be by a factor of two. We can use the barrel shift to instantly scale a mathematical result thus saving clock cycles that you would need to expend to do the shifts (multiplication or division) on separate clock cycles.

Shift Micro-ops	Worthy Comment	Worthy Picture
$R0 \leftarrow sr R0$	shift right of R0; result is store in R0; some undetermined valued is feed in to the left side of the register.	
$R2 \leftarrow sl R2 (r-0)$	shift left of R2; feed in '0' from right side	
$R2 \leftarrow sr R2 (l-1)$	shift right of R2; result stored in R2; feed in '1' from left side	
$R2 \leftarrow rr R2$	rotate right; the LSB is transferred to the MSB;	
$R2 \leftarrow rl R2$	rotate left; the MSB is transferred to the LSB.	
$R2 \leftarrow bsr2x R2 (l-0)$	barrel shift right 2x (two simple shifts); result stored in R2; feed in '0' from left	
$R3 \leftarrow bsl2x R3 (r-1)$	barrel shift left 2x; result stored in R3; feed in '1's from right. This would be the same as two simple shift lefts that fed a 1 into the right.	
$R4 \leftarrow asl R4 (r-0)$	arithmetic shift left; sign bit is copied from left side with each shift; '0' is fed into the right side of the register; this is essentially a multiply by two on a signed number.	
$R5 \leftarrow asr R5$	arithmetic shift right; sign bit is not altered any shift; the sign bit is copied from the MSB to the MSB -1 on each right-shift; this is essentially a divide by two on a signed number.	

Table 4.3: Some popular shift-type microoperations.

One important thing to notice about the RTL equations written in Table 4.3 is that none of them contain conditions. Generally speaking, there will be some unit in your computer that handles all of these functions. The way you would officially tell the unit to perform a given function is to tweak the proper control signals (such as “select-type” signals). These control signal values should appear in the RTL statements above. The above equations do not because our discussion was primarily an introduction.

The last comment on the RTL matter is fact that only conditions appear on the left side of the colon. You need to remember this because the “+” operator sometimes represents a logical OR and at other times represents addition. An OR operation is considered a condition and can appear on the left side of the colon. However, an

addition operation could not be construed as a condition and would never appear on the left side of the colon. Therefore, a “+” operator has special context in RTL equations. For example, for the equation in Equation 4.3, the “+” operator on the left side of the colon represents an OR operation while the “+” operator on the right side of the colon represents an addition operator. If ever in doubt, feel free to spell it out absolutely clearly in written English, with footnotes, or with arrows.

$$K1 + K2 : R1 \leftarrow R2 + R3$$

Equation 4.3: Equation showing “+” operator but having two different meanings.

4.4 Data Transfer Circuits

As you can tell by now, a functional datapath passes data around in a useful manner. We need to get into some of the specifics of how the data is passed around; that is, we need to look at the underlying hardware and understand exactly how things are done so that we can orchestrate such transfers. There are roughly four different circuit styles for transferring data around:

- 1) MUX-based transfers
- 2) bus-based transfers
- 3) tri-state bus-type transfers
- 4) open collector

4.4.1 MUX-Based Data Transfers

Figure 4.5(a) shows a typical circuit that performs MUX-based data transfers. The table in Figure 4.5(b) includes some example microoperations and the control signals required to perform those operations. We use the **Sx** signals to control the two MUXes and the **LDx** signals are used to control the loading of data into the various registers. We assume the width of the bus for this example and the other examples that follow to be of generic width “n”. All transfers are synchronized on the rising edge of the clock. Below are a few other things to note about this circuit; Table 4.4 shows the bit control information in RTL form.

- If a register does not need to be loaded for a particular microoperation, the LD signal is held low. The state of the corresponding MUX control signal is therefore a “don’t care” but is listed as ‘0’.
- Most often, conditions of “don’t care” are not included in the RTL statement. In general, for a given RTL statement, you should specify all associated signals to leave no room for testy ambiguity.
- Each signal source contains one and only one destination.
- The fact that the data signals in this example are of “width n” implies that they are bundles. In computerland, the word bus is an overused and ambiguous term; the word bundle is a better term.

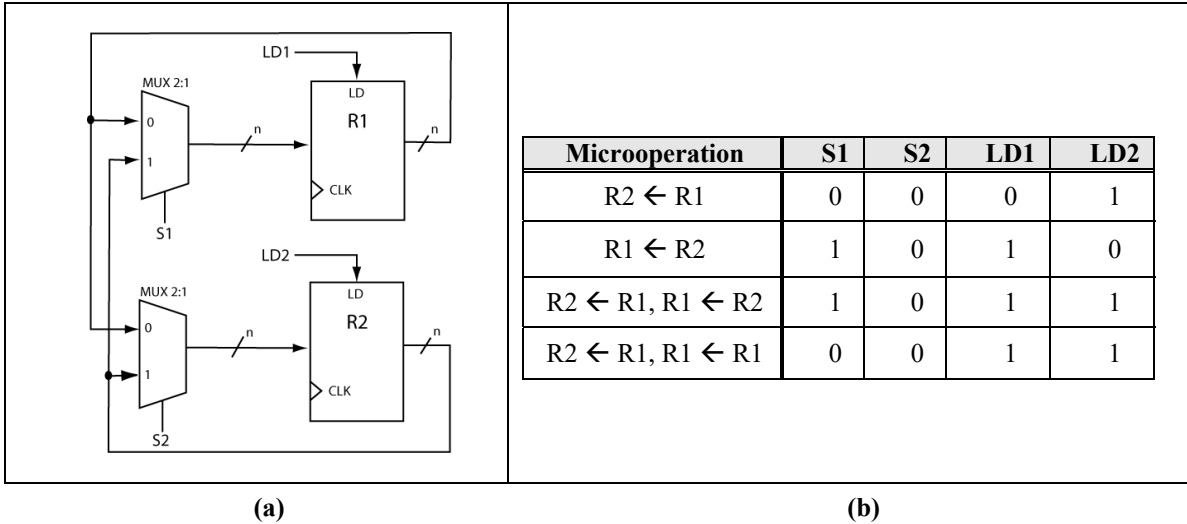


Figure 4.5: A circuit for MUX-based transfers (a) and control signals necessary to perform the listed microoperations (b).

Microoperation	S1	S2	LD1	LD2	RTL
$R2 \leftarrow R1$	0	0	0	1	$\overline{S2}, \overline{LD1}, LD2 : R2 \leftarrow R1$
$R1 \leftarrow R2$	1	0	1	0	$S1, LD1, \overline{LD2} : R1 \leftarrow R2$
$R2 \leftarrow R1, R1 \leftarrow R2$	1	0	1	1	$S1, \overline{S2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R2$
$R2 \leftarrow R1, R1 \leftarrow R1$	0	0	1	1	$\overline{S1}, \overline{S2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R1$

Table 4.4: The table from Figure 4.5(b) with associated RTL statements.

4.4.2 Bus-Based Data Transfers

Although MUX-based transfers are versatile, they can be a waste of hardware. The versatility comes from the fact that you can perform just about any action you can dream up, but it comes as the cost of extra hardware. Bus-based transfers are similar to MUX-based transfers but are not quite as versatile. Figure 4.6(a) shows a circuit for bus-based transfers. The microoperations in Figure 4.6(b) are the same ones listed in Figure 4.6(a). Here are a few things to note about this circuit:

- This bus-based circuit has less hardware than the MUX-based circuit. This ends up being a trade-off with functionality as is noted in the next bulleted item.
- Due to the limited hardware connections (compared to the MUX-based transfers), one of the desired microoperations cannot be done. Bummer!
- This is called a bus-based transfer because there is one bus that had one source but multiple destinations. Note that in the MUX-based transfers, each source had only one destination.

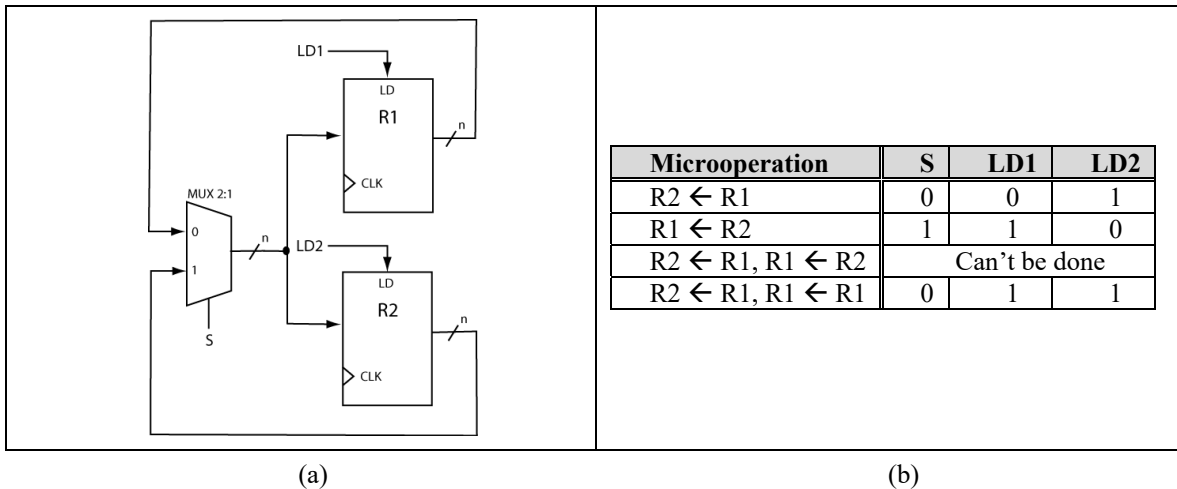


Figure 4.6: A circuit used for bus-based transfers (a); control signals to perform microoperations (b).

Microoperation	S	LD1	LD2	RTL Statement
$R2 \leftarrow R1$	0	0	1	$\overline{S}, \overline{LD1}, LD2 : R2 \leftarrow R1$
$R1 \leftarrow R2$	1	1	0	$S, LD1, \overline{LD2} : R1 \leftarrow R2$
$R2 \leftarrow R1, R1 \leftarrow R2$	Can't be done			bummer!
$R2 \leftarrow R1, R1 \leftarrow R1$	0	1	1	$\overline{S}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R1$

Table 4.5: The table from Figure 4.6(b) with added RTL statements.

4.4.3 Tri-State Bus-Based Transfers

These transfers are centered about the use of a tri-state buffer as in Figure 4.7(a). The name tri-state comes from the fact that the output of the buffer can have three possible states as is shown in Figure 4.7(b). Two of the three states are the now infamous 1's and 0's while the other state is the *high-impedance* state signified with the letter Z. When the circuit goes into the high-impedance state, no current flows through the device. Any time there is not current flowing through a conductive path, the path is considered an open circuit. In this case, if there is an open circuit, the device is effectively removed from the circuit. A better wording for this would be that the device has no significant effect on the circuit since no one is physically removing the device from the circuit. The EN (enable) input essentially enables the input to appear on the output of the device as in indicated with the truth table and compressed truth table of Figure 4.7(b) and Figure 4.7(c), respectively.

The hearts of tri-state bus transfers are registers that contains tri-state buffers on the output of the devices. In other words, each of the bits stored in the register contains its own tri-state buffer. The EN input is connected to each of the tri-state buffers in the register and controls each of the output bits in parallel. As is shown in the circuit of Figure 4.8(a), we indicate registers with tri-state outputs with the triangles on the outputs.

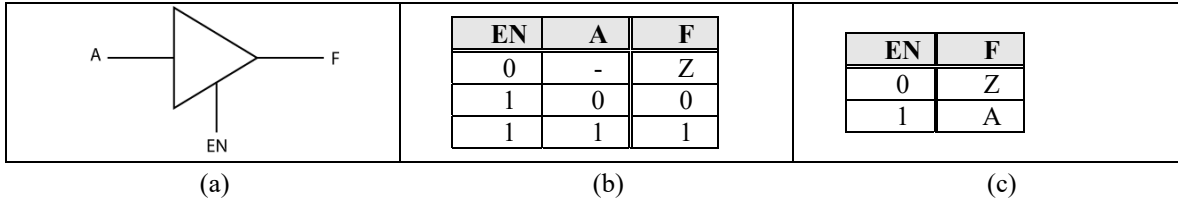


Figure 4.7: A tri-state buffer (a) and associated truth tables in full and compressed form (b) and (c).

The circuit in Figure 4.8(a) has one major difference from the two previous circuits. The differences between each the three types of transfers we're looking at are highlighted in Table 4.6. The fact that tri-state buses generally have more than one source means that there is possibility of *bus contention*. Bus contention occurs when two more output devices (registers in this case) drive their stored data onto the bus line at the same time. This results in indeterminate circuit behavior, so you should definitely avoid it. For example, if one output device drives the bus with all 1's and another drives the bus with all 0's, what would some input device see on these lines?

The way to avoid bus contention is to make sure that no more than one output device is driving the bus lines at one time. The way to drive the bus is to assert the EN input on the registers so only one of these should be asserted at any one time. When the device is not asserted, the device is essentially removed from the circuit (although the inputs of the device are generally able to latch data).

Transfer Type	Interesting Bus Characteristic for Buses
MUX-based	one source - one destination
Bus-based	one source - multiple destinations
Tri-state bus-based	multiple sources - multiple destinations

Table 4.6: The major differences between transfer types.

Figure 4.8(a) shows the resulting circuit. The microoperations listed in Figure 4.8(b) are the same microoperations for the previous types of data transfers. Once again, as you can see from the circuit diagram of Figure 4.8(a), there seems to be less hardware in the circuit as compared to MUX-based and bus-based transfers. As is shown in Figure 4.8(b), one of the RTL statements is still not possible. The most important thing to note from the table in Figure 4.8(b) is the fact that for any given RTL statement, only one of the register enables is active at a time. If more than one enable signal was active on a given bus line, there would be bus contention

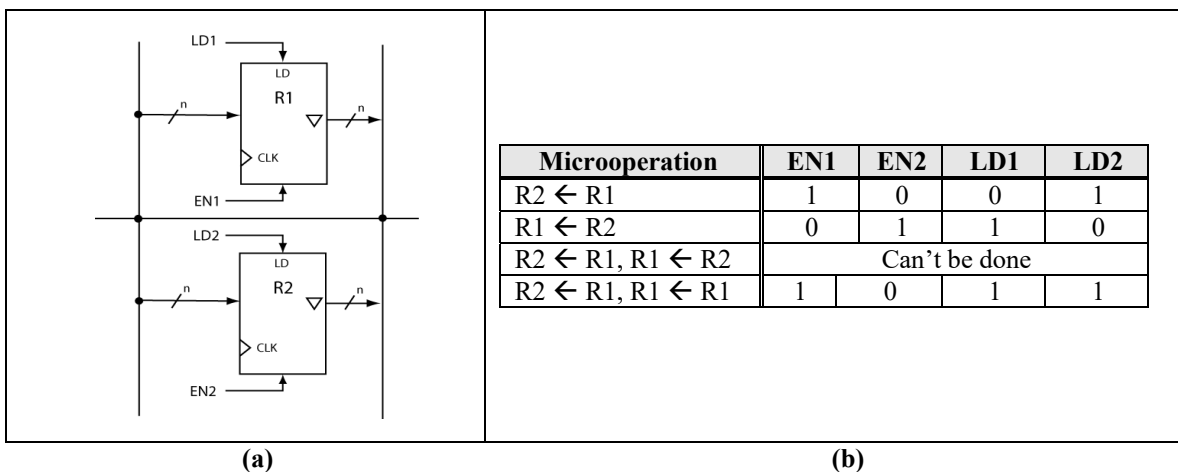


Figure 4.8: A circuit for tri-state bus-based transfers (a); signals controlling the microoperations (b).

Microoperation	EN1	EN2	LD1	LD2	RTL Statement
$R2 \leftarrow R1$	1	0	0	1	$EN1, \overline{EN2}, LD1, \overline{LD2} : R2 \leftarrow R1$
$R1 \leftarrow R2$	0	1	1	0	$\overline{EN1}, EN2, LD1, \overline{LD2} : R2 \leftarrow R1$
$R2 \leftarrow R1, R1 \leftarrow R2$	Can't be done				unkempt
$R2 \leftarrow R1, R1 \leftarrow R1$	1	0	1	1	$EN1, \overline{EN2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow$

Table 4.7: The table from Figure 4.8(b), with associated RTL statements.

And finally, there is an alternate method that is commonly used to draw the circuit of Figure 4.8(a). A somewhat shorthand notation for the tri-state bus transfer circuit of Figure 4.8(a) is shown in Figure 4.9. These two circuits are equivalent but note that the circuit of Figure 4.9 is much nicer to look at. The double arrows on the bus lines indicate that the lines are both inputs and outputs.

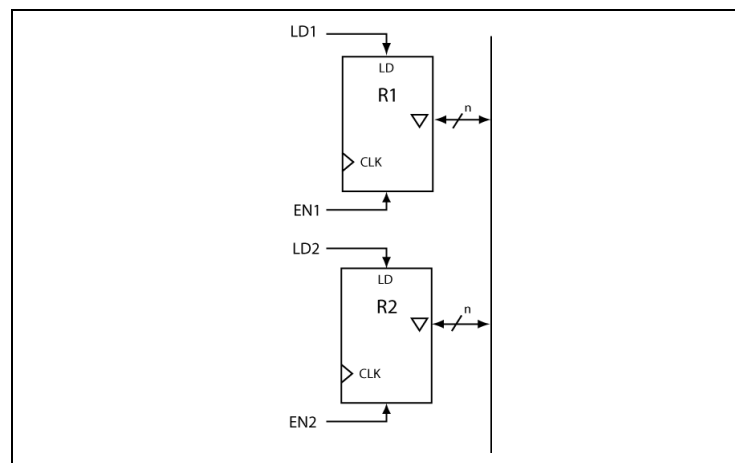


Figure 4.9: An alternative method to draw the circuit shown in Figure 4.8(a).

Example 4.5

Using the circuit shown in Figure 4.10(a), write the RTL equations that would accomplish the following list two sets of tasks: (write two different equations).

- 1) Transfer R1 to R2; increment R1
- 2) Transfer R2 to R1

Solution: Once again, there are a few quick things to notice about this circuit:

- Each of the registers has tri-stated outputs. This requires that the registers have enable signals, which must be asserted in order to drive that register's data on to the bus. This also means that only one of the enable signals (EN1 and EN2) better be asserted at one time.
- The R1 register has a CNT_EN input, which roughly stands for count enable. This implies that the R1 register is a counter. Looking at the first required transfer indicates an increment operation ($Rx \leftarrow Rx + 1$) which, by using the logic of the previous problem, requires an adder. However, since this register is a counter and counters typically count up one value at a time in a synchronous fashion, all you need

to do is assert the count enable to induce the required increment operation. In this case, assume that if the count is not asserted, the value stored in the register does not change.

Here are the required RTL equations:

$$\overline{\text{CNT_EN}}, \overline{\text{LD1}}, \text{LD2}, \text{EN1}, \overline{\text{EN2}} : \text{R2} \leftarrow \text{R1}, \text{R1} \leftarrow \text{R1} + 1$$

$$\overline{\text{CNT_EN}}, \text{LD1}, \overline{\text{LD2}}, \overline{\text{EN1}}, \text{EN2} : \text{R1} \leftarrow \text{R2}$$

Figure 4.10(b) shows the associated timing diagram. One important thing to notice about this timing diagram are the transfers that occur on the first rising clock edge. On that clock edge, the data in the R1 register transfers to the R2 register; at the same time, the data in the R1 register increments. Keep in mind that these diagrams represent actual circuits. At the instance of the rising clock edge, the data transfers from R1 to R2. Since CNT_EN is asserted, the increment of the count is also initiated on the clock edge but its effect does not happen in time for the incremented data to be transferred to the R2 register.

The above increment operation is typical in digital circuits. It's particularly important in basic computer circuits because a counter is used to "sequentially step through a stored program". Generally speaking, the output of the counter is used as an address to access an instruction in instruction memory. Once one instruction is read, the counter is incremented and then points at the next instruction in memory. We'll be looking at this in more detail in a later set of notes.

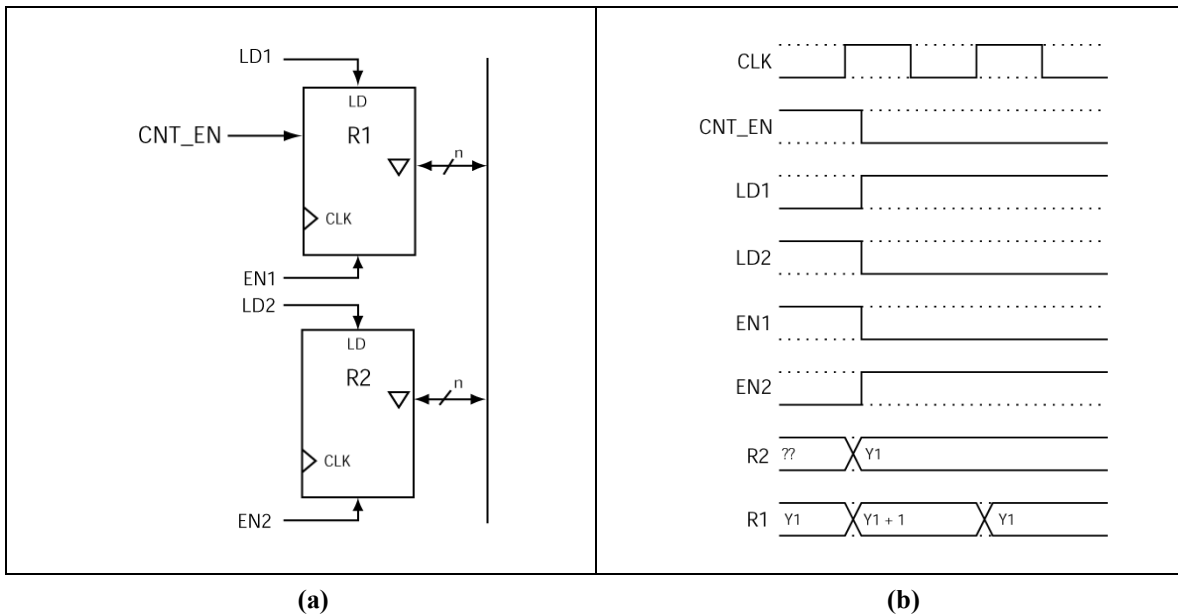


Figure 4.10: The circuit for Example 3 (a), and an associated timing diagram (b).

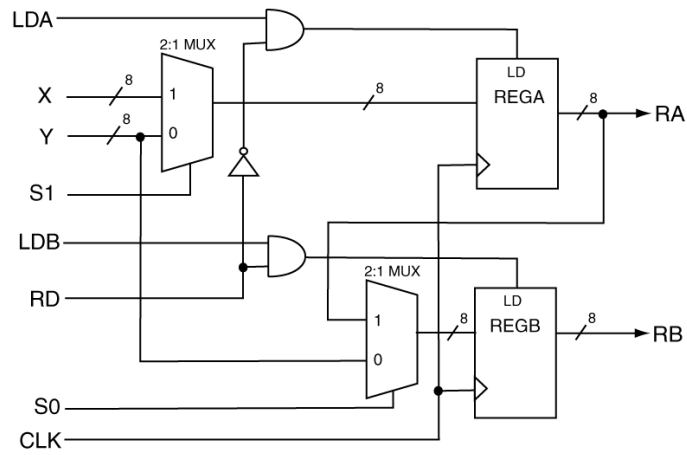
4.5 Chapter Summary

- Register transfer notation (RTN) provides a shorthand approach to both designing and describing circuits. RTN does not have absolute standards; each RTN approach may be different from other RTNs. RTN represents a continued abstraction to higher levels of design in order to facilitate designing and understanding relatively complex circuits.
 - RTN generally deals with the transfer of data from one register (the source register) to another register (the destination register).
 - Microoperations are elementary operations that are performed on data stored in one or more registers. We can describe the operation of many sequential circuits in terms of the various microoperations they are able to perform. There are many types of microoperations, but we generally attempt to categorize in order to support understanding their functions. Some of the more popular types of microoperations include transfers, arithmetic, logic, and shift operations.
 - The key of a working computer is the ability to transfer data from a source to a destination (generally register to register, register to circuit, or circuit to register). We generally attempt to classify type of data transfers in order to support our understanding of them. The most common transfer circuits include MUX-based, bus-based, and tri-state-based circuits. Each of the circuits has their advantages and disadvantages.
-

4.6 xxxxChapter Exercises

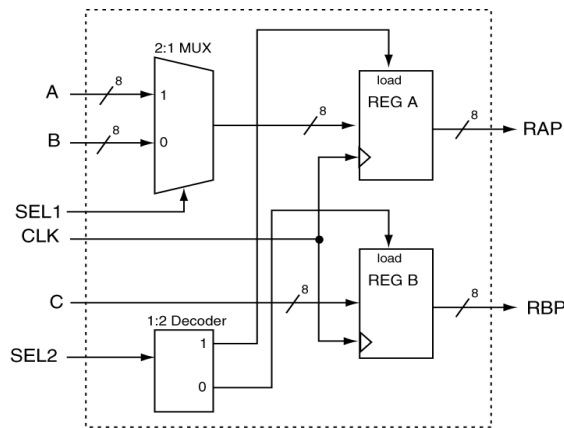
1) For this problem, complete the following two tasks:

- Write the minimum number of RTL statements that will transfer X to REGB and Y to REGA



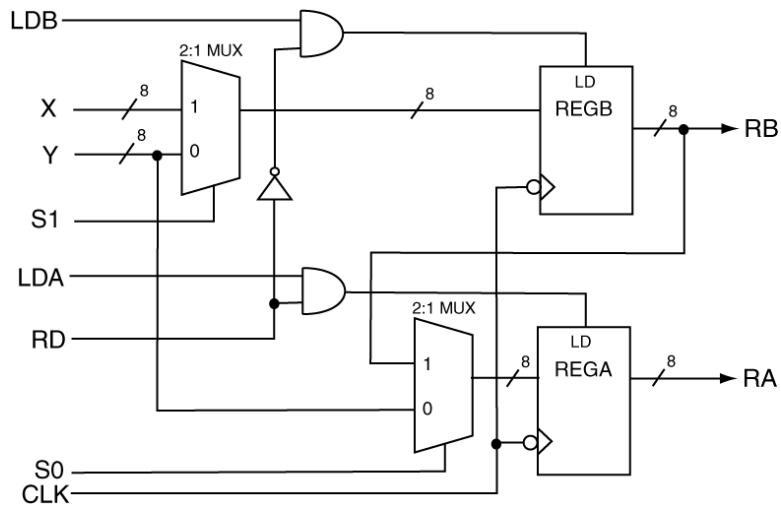
2) For this problem, complete the following task:

- Write the minimum number of RTL statements that place B into REGA and C into REGB



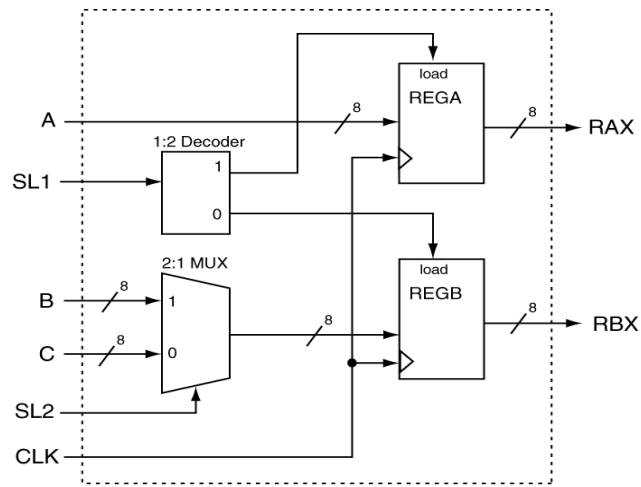
3) For this problem, do the following task:

- Write the minimum number of RTL statements that will transfer X to REGA and Y to REGB.



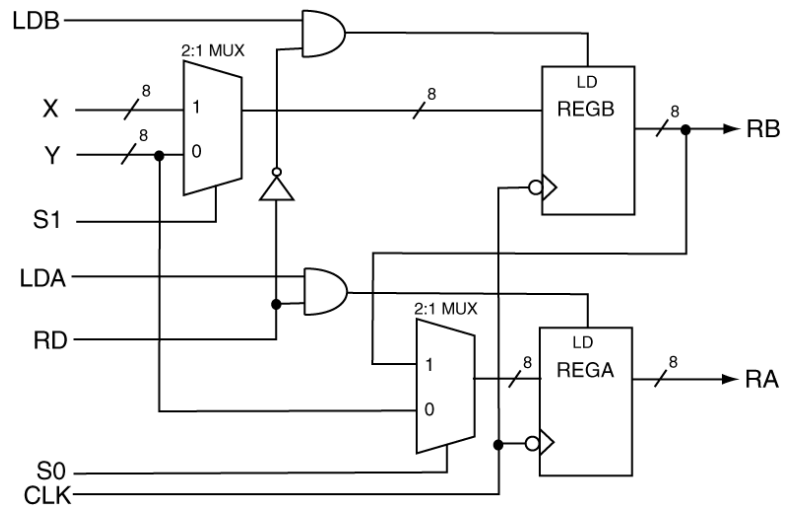
4) For this problem, do the following task:

- Write the minimum number of RTL statements that will transfer A to REGA and C to REGB.



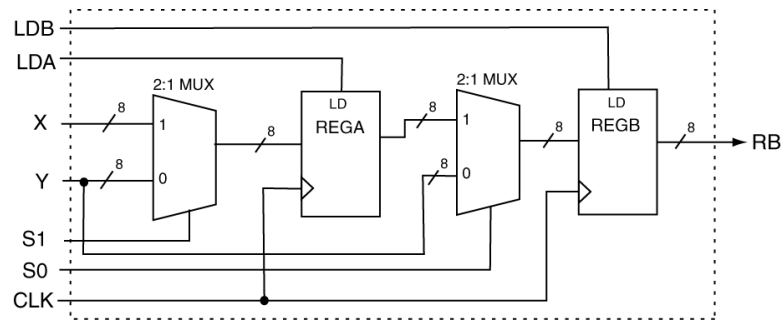
6) For this problem, do the following task:

- Write the minimum number of RTL statements that will transfer Y to REGB and X to REGA.



7) For this problem, do the following task:

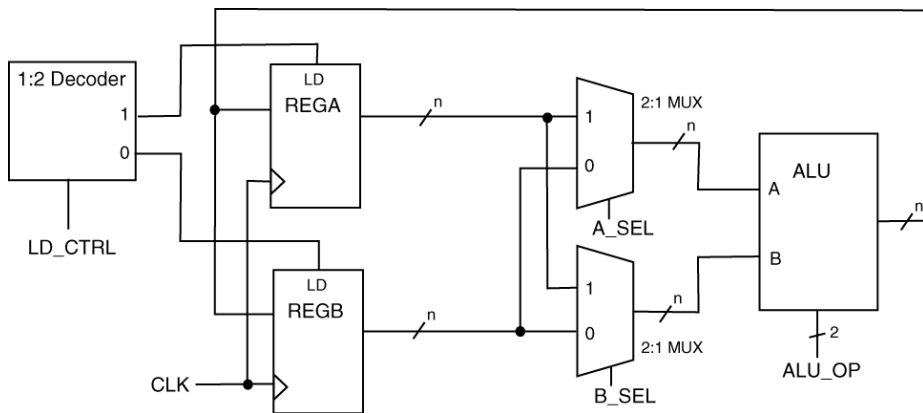
- Write the minimum number of RTL statements that will transfer X to REGA and Y to REGB.

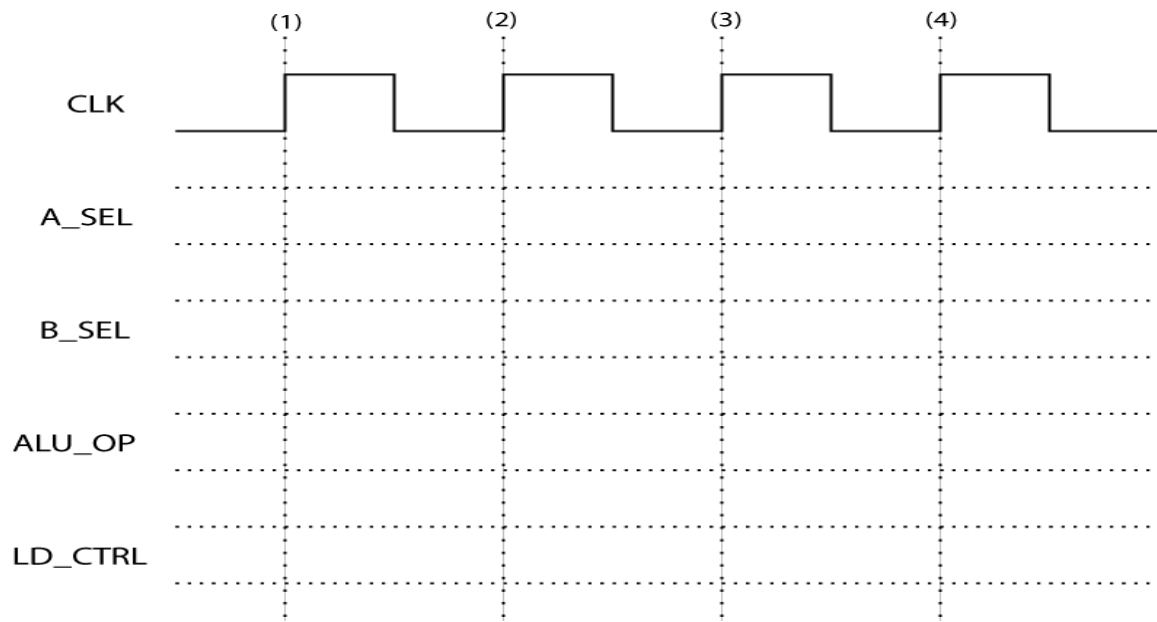


8) For this problem, provide the proper control signals that would allow the following RTL statements to occur. Complete the timing diagram below based on your provided control signals.

clock cycle	RTL
1	$RA \leftarrow RA \text{ AND } RB$
2	$RA \leftarrow RA + RA$; addition
3	$RB \leftarrow RA - RB$
4	$RA \leftarrow rr \text{ } RB$; rotate right B

ALU_OP	Operation
00	logical AND of A & B
01	rotate right B input
10	subtract B from A
11	add B to A



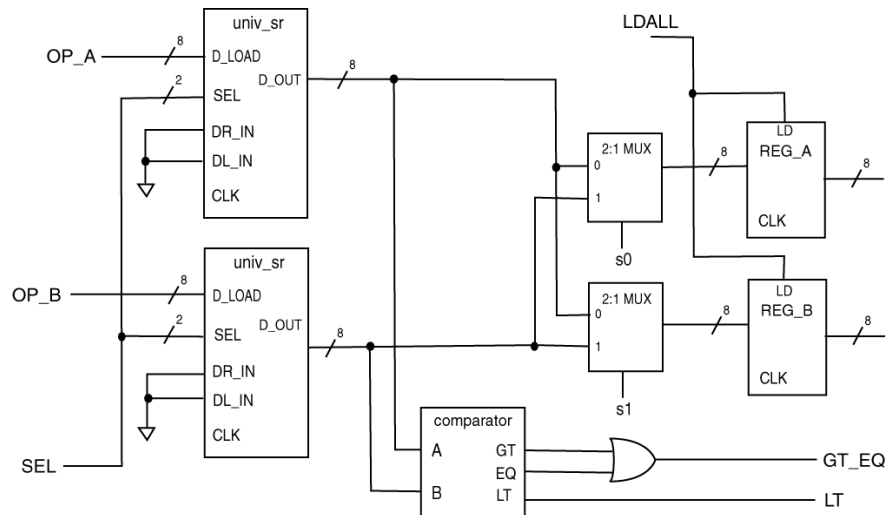


- 9) The following diagram shows a circuit that can perform a mathematical operation. The two tables below describe the comparator and the Universal Shift Register (USR). Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.*

* If a GO signal is received (GO is not shown in diagram), one of the following two operations occur:

- If $(OP_A * 4) \geq (OP_B * 4)$ then: $REG_A \leftarrow (OP_A * 4)$; $REG_B \leftarrow (OP_B * 4)$;
- otherwise: $REG_B \leftarrow (OP_A * 4)$; $REG_A \leftarrow (OP_B * 4)$;

<u>Comparator description</u>	<u>Assumptions:</u>	<u>Shift Register Controls</u>										
<pre> if (A > B) then GT <= '1'; else GT <= '0'; if (A = B) then EQ <= '1'; else EQ <= '0'; if (A < B) then LT <= '1'; else LT <= '0'; </pre>	<ul style="list-style-type: none"> • DR_IN = right side input to shift register • DL_IN = left side input to shift register • CLK signals are connected • All setup and hold times are met • All shift register operations are synchronous • Registers (non-USR) have synchronous load inputs (LD) 	<table border="1" style="width: 100%; text-align: center;"> <thead> <tr> <th style="width: 50%;">SEL</th> <th style="width: 50%;">Operation</th> </tr> </thead> <tbody> <tr> <td>0 0</td> <td>hold</td> </tr> <tr> <td>0 1</td> <td>parallel load</td> </tr> <tr> <td>1 0</td> <td>shift right</td> </tr> <tr> <td>1 1</td> <td>shift left</td> </tr> </tbody> </table>	SEL	Operation	0 0	hold	0 1	parallel load	1 0	shift right	1 1	shift left
SEL	Operation											
0 0	hold											
0 1	parallel load											
1 0	shift right											
1 1	shift left											

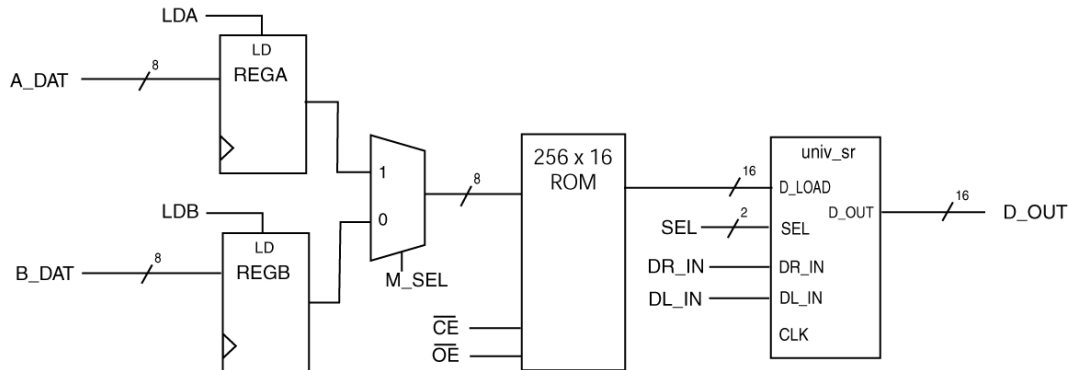


10) The following diagram shows a circuit that can perform a mathematical operation. The information below describes the Universal Shift Register (USR) and memory timing. Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.* Assume a 100MHz clock (10ns period) and a memory access time (t_{acc}) of 25ns.

* If a GO signal is received (GO is not shown in diagram), the following operation occurs:

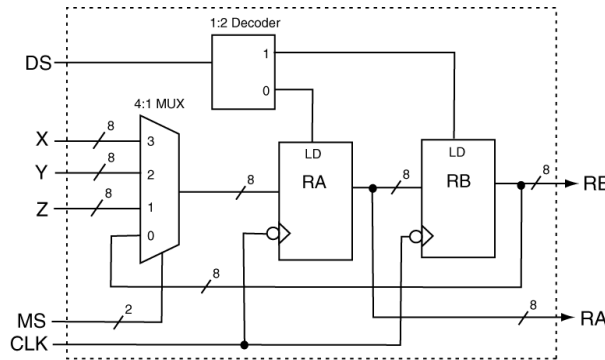
• $DOUT \leftarrow (B_DAT) \div 8$

<p style="text-align: center;"><u>Memory Timing</u></p>	<p style="text-align: center;"><u>Assumptions:</u></p> <ul style="list-style-type: none"> • DR_IN = right side input to shift register • DL_IN = left side input to shift register • CLK signals are connected • All setup and hold times are met • All shift register operations are synchronous • Registers (non-USR) have synchronous load inputs (LD) 	<table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="2">Shift Register Controls</th> </tr> <tr> <th>SEL</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0 0</td> <td>hold</td> </tr> <tr> <td>0 1</td> <td>parallel load</td> </tr> <tr> <td>1 0</td> <td>shift right</td> </tr> <tr> <td>1 1</td> <td>shift left</td> </tr> </tbody> </table>	Shift Register Controls		SEL	Operation	0 0	hold	0 1	parallel load	1 0	shift right	1 1	shift left
Shift Register Controls														
SEL	Operation													
0 0	hold													
0 1	parallel load													
1 0	shift right													
1 1	shift left													



11) For this problem, perform the following:

- Write the minimum number of RTL statements that will transfer Z to RB and Y to RA.

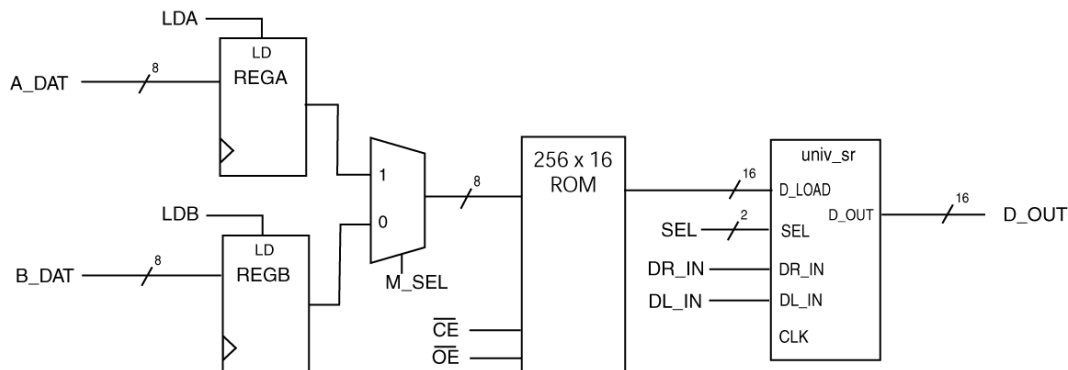


12) The following diagram shows a circuit that can perform a mathematical operation. The information below describes the Universal Shift Register (USR) and memory timing. Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.* Assume a 100MHz clock (10ns period) and a memory access time (t_{acc}) of 25ns.

* If a GO signal is received (GO is not shown in diagram), the following operation occurs:

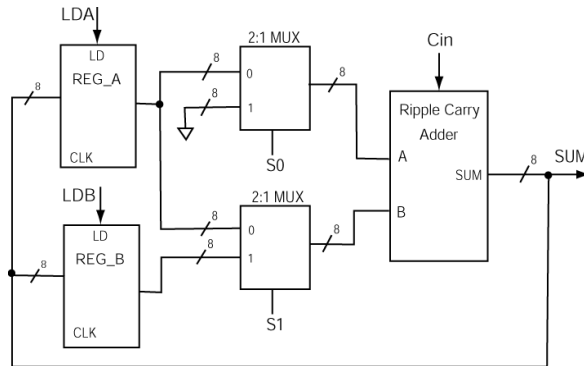
$$\bullet \text{ DOUT} \leftarrow (\text{B_DAT}) \div 8$$

<p style="text-align: center;"><u>Memory Timing</u></p>	<p style="text-align: center;"><u>Assumptions:</u></p> <ul style="list-style-type: none"> • DR_IN = right side input to shift register • DL_IN = left side input to shift register • CLK signals are connected • All setup and hold times are met • All shift register operations are synchronous • Registers (non-USR) have synchronous load inputs (LD) 	<table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="2">Shift Register Controls</th> </tr> <tr> <th>SEL</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0 0</td> <td>hold</td> </tr> <tr> <td>0 1</td> <td>parallel load</td> </tr> <tr> <td>1 0</td> <td>shift right</td> </tr> <tr> <td>1 1</td> <td>shift left</td> </tr> </tbody> </table>	Shift Register Controls		SEL	Operation	0 0	hold	0 1	parallel load	1 0	shift right	1 1	shift left
Shift Register Controls														
SEL	Operation													
0 0	hold													
0 1	parallel load													
1 0	shift right													
1 1	shift left													



13) For this problem, perform the following:

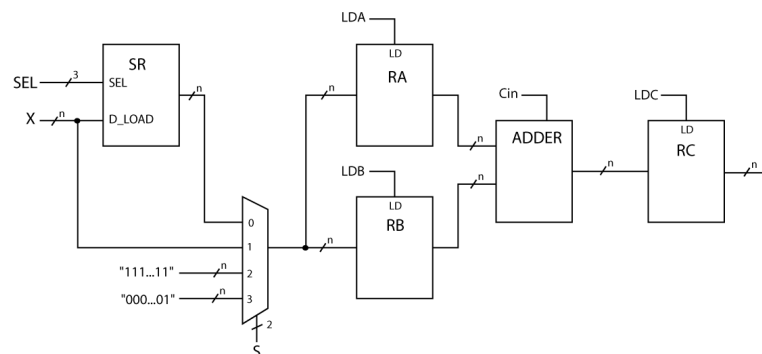
- Write the minimum number of RTL statements that will:
 - i. increment REG_B and store in REG_A
 - ii. add REG_B to REG_B and store in REG_A



14) For the following problem, assume the SEL inputs to the shift register (SR) cause the following operation in the SR. Assume "SR" refers to a shift register while "sr" refers to a shift right operation.

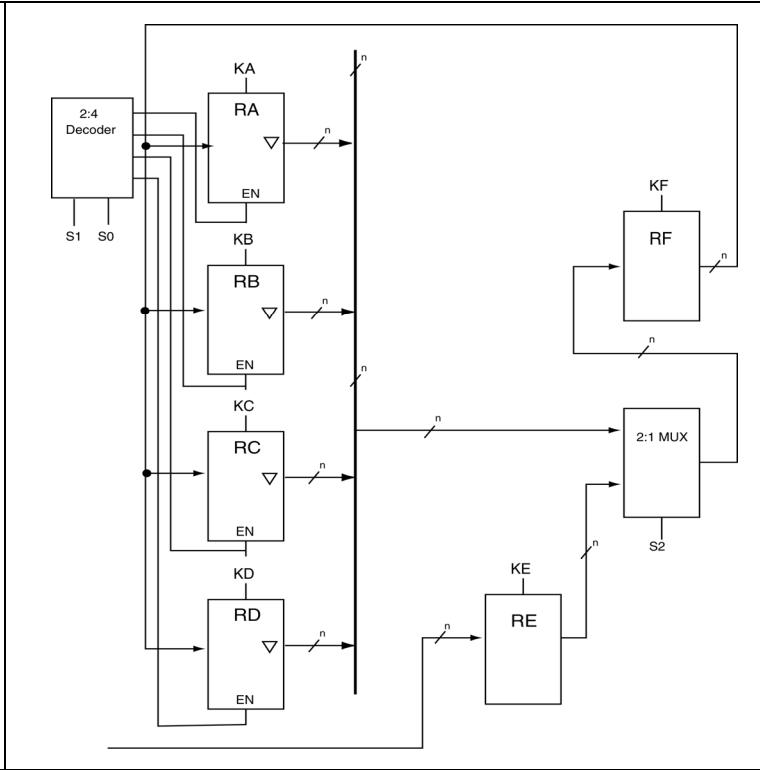
SEL	RTL Operation
000	$SR \leftarrow D \text{ LOAD}$
001	$SR \leftarrow \text{bsl}2x \text{ SR} (r-0)$
010	$SR \leftarrow \text{bsr}2x \text{ SR} (l-0)$
011	$SR \leftarrow \text{sr} \text{ SR} (l-0)$
100	$SR \leftarrow \text{asr} \text{ SR}$
101	$SR \leftarrow \text{asl} \text{ SR} (r-0)$
110	$SR \leftarrow \text{others} \Rightarrow '0'$ (loads all zero's)
111	$SR \leftarrow SR; (\text{hold})$

- (a) transfer $5X$ into RC (unsigned)
- (b) transfer $1.5X$ into RC (unsigned)
- (c) transfer $2X$ into RC (signed)
- (d) transfer $0.625X$ into RC (unsigned)
- (e) transfer $4X+2$ into RC
- (f) transfer $2X-1$ into RC

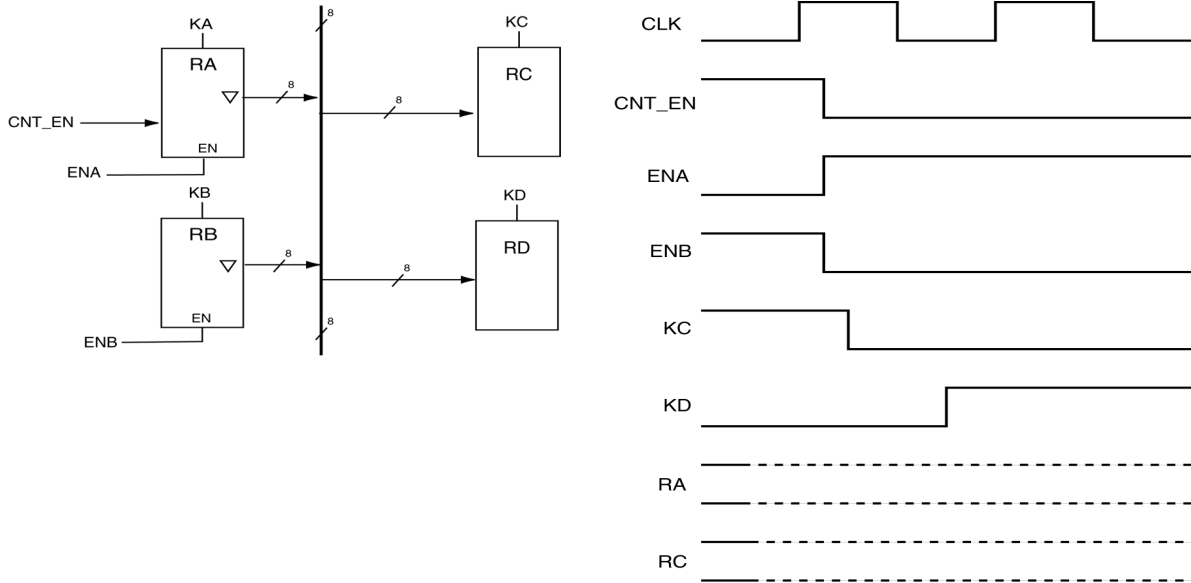


15) Write the minimum of RTL statements that will implement the following data transfers. Assume the upper-most values on the MUX and decoder start with 0 and work down to 1 and 3, respectively.

- a) Transfer RE to RD
- b) Transfer RB to RC
- c) Transfer RA to RC as well as RA to RD
- d) Transfer RD to RC and RB to RA
- e) Design a FSM that could implement the set of transfers listed in a), b), c), and d).

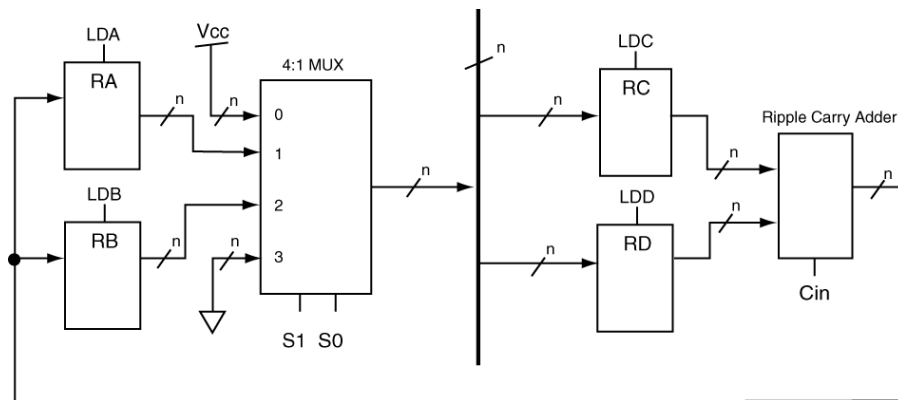


16) Use the following circuit to complete the accompanying timing diagram. Register RA is a counter with a count enable input. The initial value (in hex) on each register is RA=AA, RB=BB, RC=CC, and RD=DD. Consider the rising edge of the clock to be the active edge.



17) Using the circuit provided, write the minimum number of RTL statements that will implement the following data transfer. Consider the circuit elements with Rx labels to be registers with load inputs LDx (listed) as well as clock inputs (not listed). Data is loaded into the registers only on the active clock edge. The clock signal is not shown.

- Increment the value in RB and load the result into RA
- Add RA to RB and store the result in both RA and RB
- Decrement the value in RB and store the result in RB
- Transfer the value in RA to RB
- Clear RA and set RB (make all the bits 0 and 1, respectively)
- Load the -1 into RA
- Design a FSM that would decrease the value in RB by three and store the result in RA.



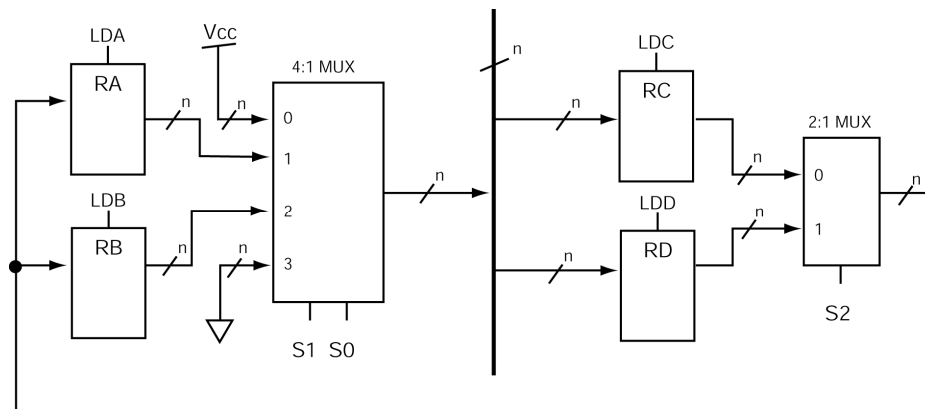
18) Using the circuit provided, write the minimum of RTL statements that will implement each of the following data transfers. Consider the circuit elements with Rx labels to be registers with load inputs LDx (listed) as well as clock inputs (not listed). Data is loaded into the registers only on the active clock edge. The clock signal is not shown.

A)

- Sets RD
- Transfer RC to RD

B)

- Clear RC
- Transfer RA to RB

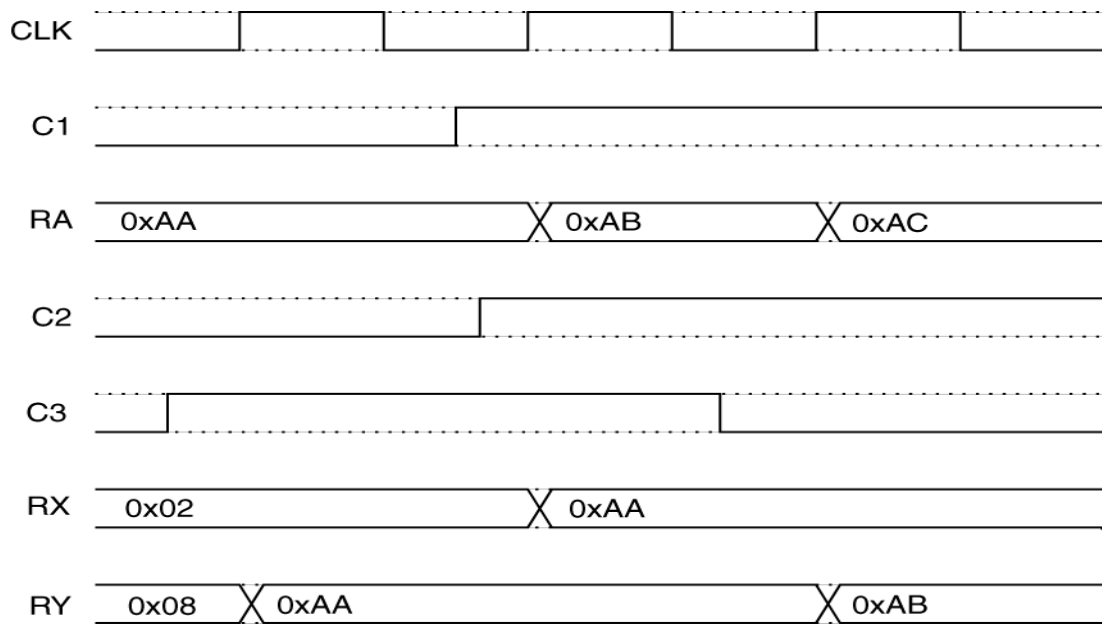


- 19) Design a circuit that can implement each of the following two RTL statements. Use busses where necessary. All registers are synchronous and contain a control input (LOAD_x). Registers may also contain other control inputs where necessary. EN_x signals are used to control tri-state outputs. Use and any other hardware you deem necessary. The “+” symbol represents an arithmetic addition.

ENA, LOADD, LOADC : RD \leftarrow (RA + RC), RC \leftarrow RD

ENB, LOADD, LOADB : RD \leftarrow (RB + RC), RB \leftarrow RD

- 20) Design a circuit that could implement the timing diagram shown below. Each of the “registers” in the design is 8-bits wide. You are not responsible for setting the listed initial values. Use any hardware you want. There are three clock cycles shown; provide an RTL statement describing the microoperations that occur at each clock edge.



5 Structured Memory: RAM and ROM

5.1 Introduction

The previous chapters dealt with basic memory elements in digital design, but on a relatively small scale (flip-flops and registers). While those types of memory are important, you typically find other types of memory in digital systems. We classify flip-flops and registers as “incidental” memory; this chapter introduces the notion of “structured¹” memory, which has significantly more storage capacity than incidental memory. You must learn a new set of skills and vernacular when you deal with structured memory; this chapter discusses some of the more basic aspects of memory.

Main Chapter Topics

- **OPERATIONAL OVERVIEW OF MEMORY:** This chapter provides an overview of the basic operational and performance characteristics of memory as well as common terminology associated with memory.
- **MEMORY TYPES:** This chapter introduces the two accepted main types of memory, RAM and ROM, by describing their differences and similarities.
- **MEMORY INTERFACE METRICS:** This chapter describes the basic interface issues involved in structured memory device.
- **STRUCTURED MEMORY MAPPING & MEMORY SYSTEMS DESIGN:** Memory systems are typically designed as many individual memory units as opposed to one single unit. This form of design requires a unique high-level perspective of the system and uses standard digital devices in their implementations. This chapter provides an overview and introduction to structured memory system design.

Why This Chapter is Important

- This chapter is important because it describes the basics of concepts associated with large memory devices such as well as working with and interfacing with those devices.

5.2 Memory Introduction and Overview

There are many different types of memory out there; most of them are beyond the scope of a basic digital design course. If you ever need to work with a new memory device, you’ll be ready because you’re familiar with the basic operation of structured memory.

Before we start, we need to make one clarification. Often time when we discuss the notion of memory, we sometime use the terms “data” and “information” interchangeably. In most cases, this is no big deal, but you need to understand there is a distinct difference. In the context of digital design, data is nothing more than a bunch of 1’s and 0’s, while information relates to the interpretation of the 1’s & 0’s. We often refer to data as having information content; there is actually a unit used to measure the information content of data². It is up to the user to interpret data as having certain information content or not. For example, consider a memory unit; if the stored data represents instructions to a computer, then you could consider

¹ I’ve adopted this term from the notion of “regular structures”, which roughly refers to larger semiconductor devices that have a large and repeated structure that is dedicated to a single purpose. In this case, the purpose is memory.

² Somewhat unfortunately, we use the term “bit” to measure the information content of data. This metric is a function of probability and is not related to the “binary digit” definition of bit that we use in this text.

the data to be information. On the other hand, if you have a memory that you have never written to, the memory is still full of 1's and 0's, but the data has no meaning.

5.2.1 Basic Memory Operations: READ and WRITE

The two operations associated with memory are reading and writing. The notion of a “memory read” or “reading from a memory” refers to the action of retrieving data currently stored in memory. Retrieving data specifically means that you’re copying the data from memory to another place, but not changing the data in memory. The notion of a “memory write” or “writing to a memory” refers to the action of placing new data in memory, which means you are changing the data stored in memory. Reading and writing memory are the copying of data from memory (reading) and the transfer of data into memory (writing), respectively.

5.2.2 Basic Memory Types: ROM and RAM

There are many different flavors of memory in digital-land; each of these memory types has their own acronym describing them. Despite this relatively high number of memory types, we classify all of them as either RAM or ROM, which are acronyms for random access memory and read only memory, respectively. These terms are rather misleading, particularly in regards to the attributes of modern memory. In an effort to classify memories as either RAM or ROM, these two acronyms have rather loose definitions. Here is the information embedded in those acronyms.

- The notion of a “read only memory”, or ROM, implies that you’ll only be reading from a memory, and never writing to it. Because the memory is a “read only” memory, you can only retrieve data from that memory; you cannot “easily”³ alter the data in that memory.
- The notion of a ROM brings up the issue of whom or what put the data into the ROM. This starts delving down into the various sub-types of ROM; we don’t want to go there because we want to keep this discussion general. Writing to a ROM is a “special” operation performed by “something”. All we’re interested in is that there is data in the ROM.
- The term random access refers to the fact that it requires the same amount of time to access (either reading or writing) each “chunk” of memory stored in the device. While this notion seems rather simple, not all memory devices fall into the category of “random access”. The two most obvious notions of non-random access memories are “hard drives” and “tape drives”. The time required to access data in your hard drive is different depending on the physical location of the data on the disk and the current location of the read/write heads. Recall that the hard drive is a mechanical storage device that requires motors to move a physical device (the read/write head) radially across the spinning media to access the data. If the heads are close to the data, it requires less time to access the data than if the heads must move a long way to access the data.
- Although the term ROM refers to read only memory, ROMs are also random access devices. Thus, you can access any of the chunks of data stored on a ROM in an equal amount of time.
- All memories have the notion of being either volatile or non-volatile. If a particular memory is volatile, the data stored in that memory is lost when you remove power from that circuit. Conversely, the data in non-volatile memory is not lost when you remove power. We generally accept that RAMs are volatile and ROMs are non-volatile.

Despite all these misleading terms and acronyms associated with structured memory, RAM and ROM do have accepted definitions. Table 5.1 lists these accepted differences and similarities.

Memory Type	Random Access	Operations	Volatility
RAM	yes	read & write	volatile
ROM	yes	read	non-volatile

Table 5.1: Accepted attributes of RAM and ROM.

³ Meaning that many types of ROM can be written to; we’ll not discuss those cases.

5.3 Software Arrays vs. Hardware Structured Memories

The notion of structured memory is not as new as it seems, as there is a direct analogy to the use of arrays in programming languages. Recall that an array in computer programming is a data structure that allows you to store values and later access those store values.

Accessing values in an array: This operation is analogous to a read of a memory. In computer programming, when you access a value in an array, your program must provide an index that indicates which value in the array you want to access. The array “returns” the requested value without changing that value in the array. In hardware, the circuit must provide value (the address) that indicates which address in the memory you want to read from. The memory then outputs that value; the read operation does not change the value.

Changing values in an array: This operation is analogous to a write of memory. In computer programming, when you place a new value into an array, your program must provide an index that indicates which value in the array you want to change. The array then replaces that value with the new value. In hardware, they circuit must provide a value (the address) that indicates which value in the memory you want to write to and the new data. The memory then changes the value at that address to the new value.

5.4 Memory Operation Details: Reading and Writing

Figure 5.1 shows a high-level diagram of a generic memory device. We can classify the various signals associated with interfacing with a memory device into three categories: address lines, data lines, and control lines⁴. The following is a general overview of these lines. In general, the widths of these bundles are associated with the specific capacity attributes of the memory; we deal with those issues soon.

Data Lines: The data lines are a set of signals that route the bits you’re writing or reading into or out of the memory device. The arrow associated with the data lines has an arrowhead on each end, which signifies that data on those particular lines can travel either into the memory (for read operations) or out of the memory (for write operations)⁵. The data lines can be either serial or parallel; the bundle notation in Figure 5.1 means the data lines are parallel. Figure 5.1 happens to show only one set of data lines; memories often separate input and output data lines.

Address Lines: The address lines are a set of signals that provide the memory with a “location” within the memory to write to or read from. The address lines are the method that the memory uses to differentiate between chunks of memory on the interior of the device.

Control Lines: The control lines are a set of signals that determine and direct the various operations associated with the memory. The best example of the responsibility of the control lines are with RAM devices that are both readable and writeable; the control lines allow the user to control which operation occurs. The underlying notion of control lines is that simple memories have few control lines; more complex memories have more control lines⁶.

We soon delve further into the details of memory interfacing; for now, you can consider the general interfacing operation of a memory read as: 1) give the memory an address, 2) tweak the control lines, and 3) wait for the data. For memory writes, you generally 1) give the memory an address, 2) give the memory the data, 3) tweak the control lines, and 4) wait for the data to write to memory.

⁴ In this context, the notion of “lines” refers to a bundle of wires or signals. You often hear the term “lines” associated with standard bundles such as “data”, “address”, and “control” lines.

⁵ But not both directions at the same time.

⁶ In an effort to increase memory capacity while keeping physical size small, interfacing some modern memories have become rather complicated and thus have a relatively large number of control signals.

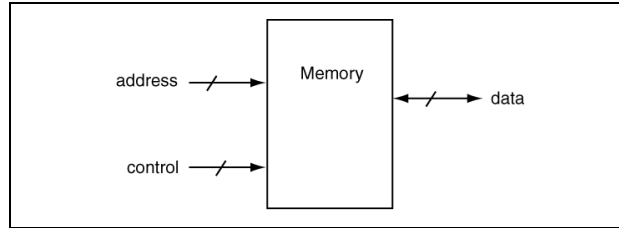


Figure 5.1: A general diagram of a memory integrated circuit.

5.5 Memory Specification and Capacity

When working with memory and memory systems, the two most important pieces of information are the capacity and the speed of the memory. The memory capacity refers to how much data the memory can store while the memory speed refers to how fast you can access (read or write) that data.

People in digital-land describe memory capacity in many different ways. As is typical in any human oriented pursuit, people attempt to make their “thing” sound better than it really is; the same idea applies to memory capacity specifications. While these statements are not lies, they are misleading. You, the digital designer must see through the smoke and hand waving and understand the characteristics of the memory you’re working with.

We know that memory stores bits, and these bits are stored at certain addresses within the memory, but memories are rarely bit-addressable. In other words, specific memory devices only allow you to access larger chunks of data. If you need to read or write a single bit, you must start with the minimum chunk of addressable data specified by the device. Making memory bit-addressable would create an inefficient device, so memories generally compromise by providing data only in chunks.

Memories usually store data in groups of bits, which we refer to as a word. The official definition of a word is the smallest addressable unit (or chunk of bits) in a memory. This term is important because we typically described memories and memory systems in terms of words rather than bits. Referring to memory in terms of words is the honest approach.

Figure 5.2 shows a diagram of a generic memory including some typical memory characteristics. The metrics in the diagram are typical of most memory devices. Here is an overview of the most important aspects of Figure 5.2 while Table 5.2 summarizes all the gory details.

- The “ $2^m \times S$ ” notation is how we state the capacity of a memory. The underlying notion is that we are modeling the memory as a two-dimensional grid, as the “ x ” in “ $2^m \times S$ ” indicates.
- Everything having to do with memories relates to binary. The term “ m ” refers to the width of the address bus or number of address lines, which is the number of memory chunks that a memory can access is two raised to the number of address lines. The true capacity of a memory (the amount of data it can store) relates to the number of address lines.
- The term “ S ” is the width of the data bus or data lines, or the word width for the memory. Datasheets often state this metric in bits, but should state it in word capacity.
- The total word storage capacity for the memory is how many words the memory can store. For this particular memory, the word storage capacity is thus 2^m .
- The total bit storage capacity for the memory is a product of the number of words and the number of storage locations in the memory. Thus the bit storage capacity is given by $2^m \times S$.
- We don’t include a bundle width indication on the control lines in order to keep the discussion general. The notion of $2^m \times S$ is common; the control lines for memory modules tend to vary greatly across different devices.

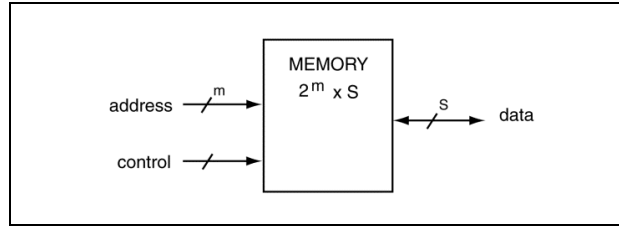


Figure 5.2: A diagram of memory indicating notions of storage capacity.

$$\text{capacity (in bits)} = 2^m \times S$$

Equation 5.1: Closed form formula for memory storage capacity in bits.

Symbol	Definition
m	Bit-width of address bus
S	Bit-width of data bus (word size)
2^m	Memory capacity in words
$2^m \times S$	Memory capacity in bits

Table 5.2: Summary of memory definitions and properties.

5.6 Memory Interface Details

This section examines the control lines and their relation to the data and address lines for basic read and write operations on a generic memory. Recall that a memory write transfers a word to be stored in memory while a memory read prompts a memory to output the contents of memory. The reading and writing of memory is controlled by the “control lines” of the memory device. Every memory has its own method of reading and writing; specifically, each memory has its own protocol for tweaking the control lines in such a way as to obtain the desired function from the memory device.

Memory Writes: For a memory write operation, you provide the memory with data that overwrites data currently stored in the memory. The information on the address lines provides the location of where the word is stored. The bits on the data lines provide the data that we transfer and store on the memory device. The write operation overwrites the data currently stored at the address indicated by the address lines.

Memory Reads: For a memory read operation, you prompt the memory device to output the data currently stored at a specific location in memory. The information on the address lines provides the location in memory of where you want to read from. Thus, the address lines provide the memory location of the word that transfers out of the memory; the transfer occurs by placing the data at the specified address onto the data lines. Read operations don’t alter values stored in the memory device.

Steps for Memory Writes	Steps for Memory Reads
Apply the information representing the memory location of where you desire to store the given word to the address lines.	Apply the information representing the memory location of where you desire to retrieve the given word to the address lines.
Apply the information representing the actual data bits to be written to the data lines.	Tweak the control lines to make the read operation occur.
Tweak the control lines to make the write operation occur.	Wait for valid data to be output
Wait for data to write	

Table 5.3: Summary of generic steps required for memory reads and writes.

5.7 Memory Performance Parameters

When we speak about memory devices, we're talking about actual physical electronic devices. This means that read and write operations require finite amounts of time to happen. Most of the associated performance parameters are outside the scope of this discussion, but some are basic enough for an overview here.

Figure 5.3 shows a BBD for a simple RAM. This RAM has two control inputs: CLK and WE, where WE is a common acronym for write enable. The BBD for this RAM does not completely describe how the device operates; you need more information, as we use this device in several examples. Here is what we need to know about the device in Figure 5.3:

- The RAM in has an asynchronous read. This means that the RAM outputs the requested data as soon as it is physically capable after it receives a new address value; the read operation is not dependent upon the clock signal. The WE enable remains unasserted for read operations.
- The RAM in has a synchronous write. This means that write operations are synchronized with the active edge of the clock, which we assume is the rising edge in this example. The device initiates the write operation when it detects an asserted WE signal at the same time as a rising clock edge. The write operation requires a finite amount of time to complete.

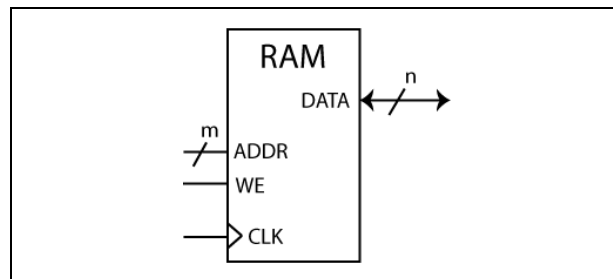


Figure 5.3: A typical control sequence for a memory read operation.

Figure 5.4 and Figure 5.5 show generic timing diagrams associated with typical read and write operations, respectively. For this device, the number of address and data lines does not matter for this discussion

Figure 5.4 shows a timing sequence for a memory read operation. Because the reads are synchronous, we don't need to show the CLK input. The one control input of interest is the WE, which remains unasserted for the read operation. Once a valid address appears on the ADDR input, the RAM outputs the data at that storage address after a finite amount of time, which we refer to as the read access time.

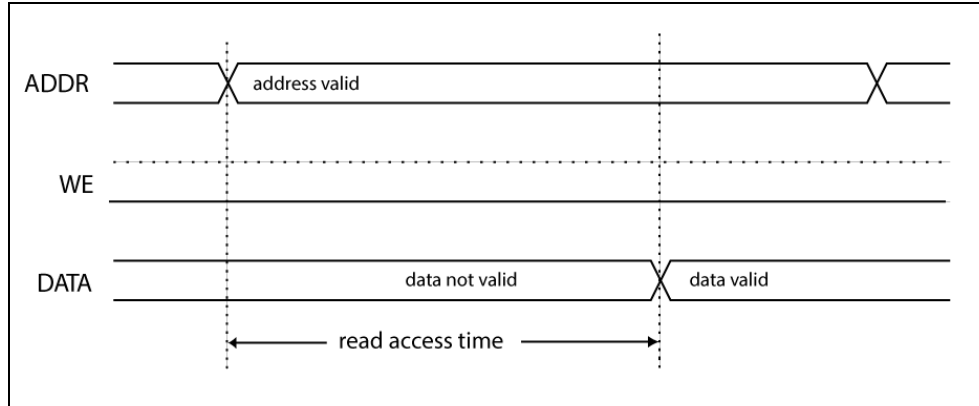


Figure 5.4: A typical control sequence for a memory read operation.

Figure 5.5 shows a timing sequence for a memory write operation. Because this RAM has synchronous writes, we include a CLK signal in the timing diagram. The writing of new data to the RAM is initiated by two control signals: CLK and WE. For a write to initiate, the WE control input must be asserted when a rising edge appears on the CLK input. The physical writing of data to the RAM occurs a finite amount of time later, which we refer to as the write cycle time.

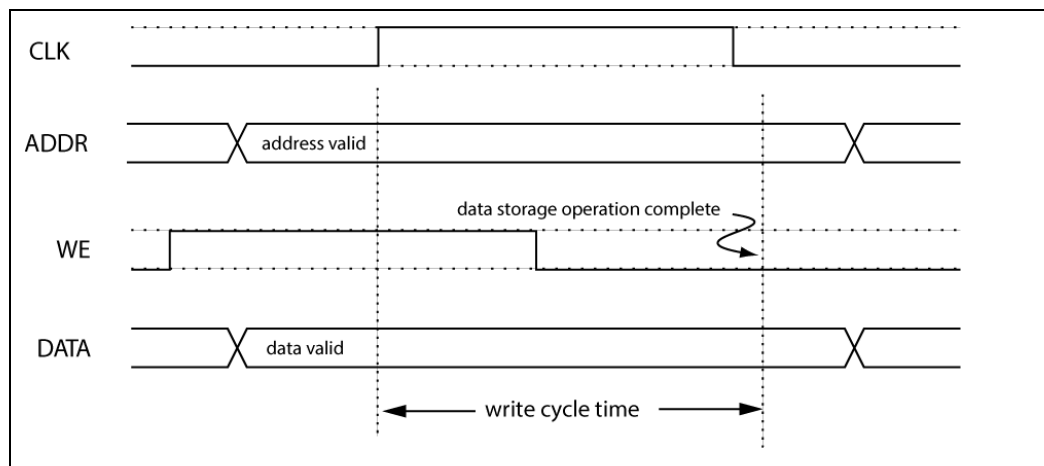


Figure 5.5: A typical control sequence for a memory write operation.

We use three main parameters to describe memory performance, which states how fast you can read from memory (read access time), how fast you can write to memory (write cycle time), and roughly how much data you can pass back and forth to and from the memory (bandwidth). Figure 5.4 and Figure 5.5 show graphic examples of the read access and write cycle times, respectively. The list below provides a more detailed description of these three performance parameters.

Memory Read Access Time: The minimum time required to access a word from memory. This is the amount of time measured from the application of a valid address to the address lines to the appearance of the valid data on the data lines.

Memory Write Cycle Time: The minimum time required to write a word to memory. This is the time measured from the application of a valid address lines to the completion of the internal operations required to successfully store the data in memory.

Memory Bandwidth: The maximum data transfer *rate* for a memory device. Since both read and write operations require finite amounts of time, it's worthwhile knowing the amount of data that we can physically transfer to and from memory in a given amount of time.

As with just about everything in digital-land, the faster something can operate, the more highly regarded that devices. This is maybe even more so true with structured memory devices as they are typically a major component in many digital systems, particularly computer systems. Moreover, in many digital systems, more than one device in the system must access memory. Often times more than one device must simultaneously access memory; this situation creates what we refer to as a bottleneck. This condition is undesirable in the one or more devices must wait to access memory⁷. The notion of “waiting” in digital-land means your device is probably doing nothing, thus probably lowering the overall throughput of your system. Roughly speaking, the faster your memory operates, the less chance of a bottleneck; or the less problematic that bottleneck is if you had a slower memory.

Any time you work with a new memory device, you’ll find yourself concerned with the above parameters. Probably one of the most informative items regarding working with memory devices is the associated timing diagram, which you can find in the associated datasheet. There is almost a special language used to specify all the timing parameter associated with memory devices, once you start working with memories, you’ll quickly get the hang of things.

Example 5.1: Design #1: RAM Summation

Design a circuit that sums the values in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The summation begins when a GO signal asserts. The final sum remains on the circuit’s output until another assertion of the GO signal. Assume the circuit contains numbers in unsigned binary format. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit’s FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

Solution: The first step in your solution is drawing the top-level BBD. The problem statement generally states the exact characteristics of outputs in problems such as these (though sometime not overly explicit), but this problem requires some extra thought and calculation. We need to show the width of the output, which represents a summation of the 16 values in the RAM. The width of the data in the RAM is 8-bits, and we know they are unsigned values. This means the largest value of the sum is $16 \times (2^8 - 1)$. We could break out the calculator, but it’s better to note that we’re working with powers of two, so the maximum summation is $2^4 \times 2^8$, or 2^{12} . Therefore, the width of the summation is 12 bits. Figure 5.7 shows the top-level BBD for this problem.

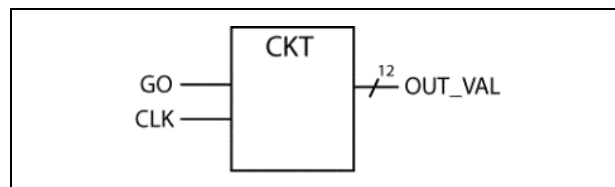


Figure 5.6: The top-level BBD for this example.

The next step in the solution is to create an inventory of the modules our solution requires. The following is an outline of our thought process.

- We know this problem has a RAM because the problem description says so.
- Any RAM we work with in this text uses the output of a counter to provide an address input to the RAM. Many different circuits or modules can provide the address inputs, but the simplest approach for this text is to use a counter output provide the address.

⁷ There is a notion of “multi-port” memories. These memories typically allow some type of parallel operation such that two devices can simultaneously read from two different memory locations. These types of memories become expensive and certainly exercise the inherent trade-offs in digital systems designs.

- The circuit also is summing all the values in the RAM. Because the RAM can only output one value at a time, we need a circuit that keeps a running total of the RAM's stored values. This calls out for an accumulator, which is a combination of an RCA and a register. The accumulator's register provides a persistent output.
- Something must control this circuit, and this control is non-trivial, which calls out for a FSM.

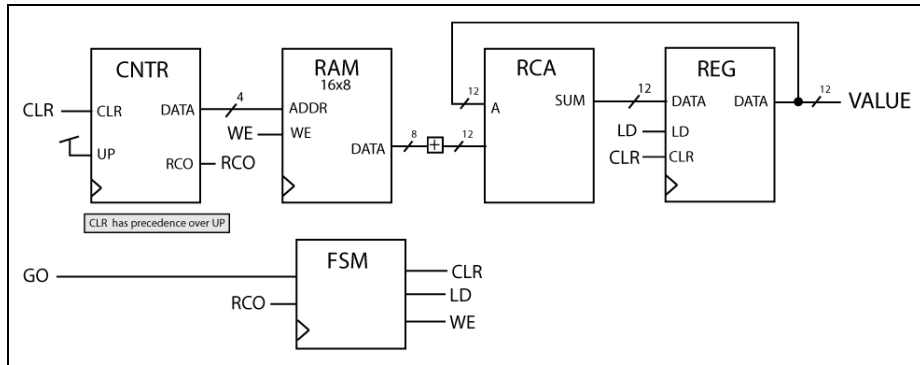


Figure 5.7: The lower-level BBD for this example.

Figure 5.7 shows the final circuit for this problem; meaningful commentary follows the diagram.

- The counter always counts up when it's not loading.
- We need to zero-extend the RAM data to make it 12 bits, which makes the RAM output compatible with the output of the accumulator's register, and the other input to the RCA. We use the square symbol with a "+" in the center to do this (which is arbitrary).
- We had to include an annotation stating that the counter's CLR input has precedence over the UP control input.

Figure 5.8 shows the state diagram for this example; here are a few items of interest to note about the state diagram.

- We drew the state diagram using two states, which requires treating CLR as a Mealy-type output. This approach was arbitrary, but it saved drawing an extra state.
- In the "wait" state, the register's LD input is disabled; we enable it while the circuit is summing.
- We always disable the RAM's WE input as this problem requires no writing to the RAM.
- The FSM remains in the "sum" state until the counter asserts RCO.

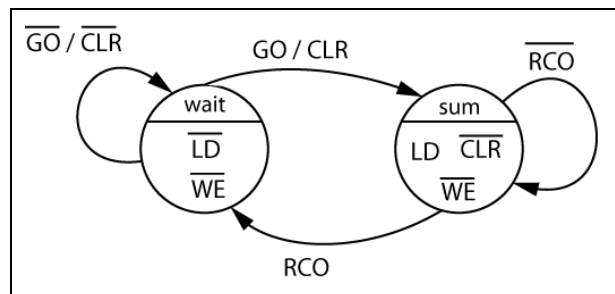


Figure 5.8: The state diagram associated with this example.

The FSM controls both the LD and CLR inputs, while the UP input of the counter is hardwired to always count up. The GO signal is a form of external control. This circuit thus has external, circuit, and internal controls.

The counter has 16 unique count values that it steps through after receiving a GO signal. The first clock cycle causes the FSM to transition from the “wait” state to the “sum” state. The summing operation for this circuit thus requires 17 clock cycles.

Example 5.2: Design #2: Minimum Value & Address Displayer

Design a circuit that finds the smallest value in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The summation begins when a GO signal asserts. The circuit’s output shows the minimum value as well as the address where that value resides in RAM. Both the value and the address remain on the circuit’s output until another assertion of the GO signal. Assume the circuit contains numbers in unsigned binary format and that every value in the RAM is unique. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit’s FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

Solution: This is another problem that requires iterating through all the values in a RAM. In this case, the circuit outputs the minimum value in RAM as well as the address of that minimum value. Figure 5.9 shows the top-level BBD for this solution.

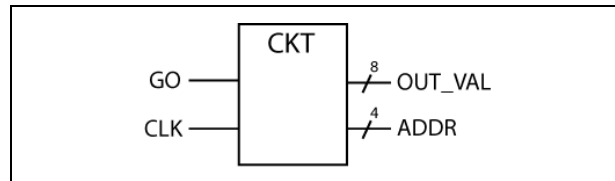


Figure 5.9: The top-level BBD for this example.

The next step in the solution is to create an inventory of the modules our solution requires. Here is the general thought process.

- The problem description states that the circuit contains a RAM; we then know that the circuit then uses a counter to generate an address for the RAM. There are 16 values in the RAM, so the width of the counter’s output is 4-bits.
- The circuit needs to store two values: the smallest value in the circuit and the location in RAM of the smallest value. These values both need to be persistent after the algorithm completes, so we know that the circuit requires two register. The register storing the smallest value is eight bits while the register storing the address of that value is four bits.
- This circuit needs to do continual comparisons to find the smallest value, so we also require an 8-bit comparator.
- In an effort to make this circuit generic, we first pre-load the 8-bit register with the minimum possible unsigned 8-bit value. The first step in the algorithm is then to load “all 1’s” into the register that holds the minimum value, which we do in order to reduce the complexity of the overall circuit. This is somewhat of a trick, but it is something you see often.

- We use a MUX to select what value appears on the minimum value register's DATA input. We first need to load the register with the maximum 8-bit value; after that, we need to be able to load the register with the current RAM value when the comparison result dictates.
- We need to state that CLR has precedence over the UP input for the counter, and that the CLR input has precedence over the LD inputs for the two registers.

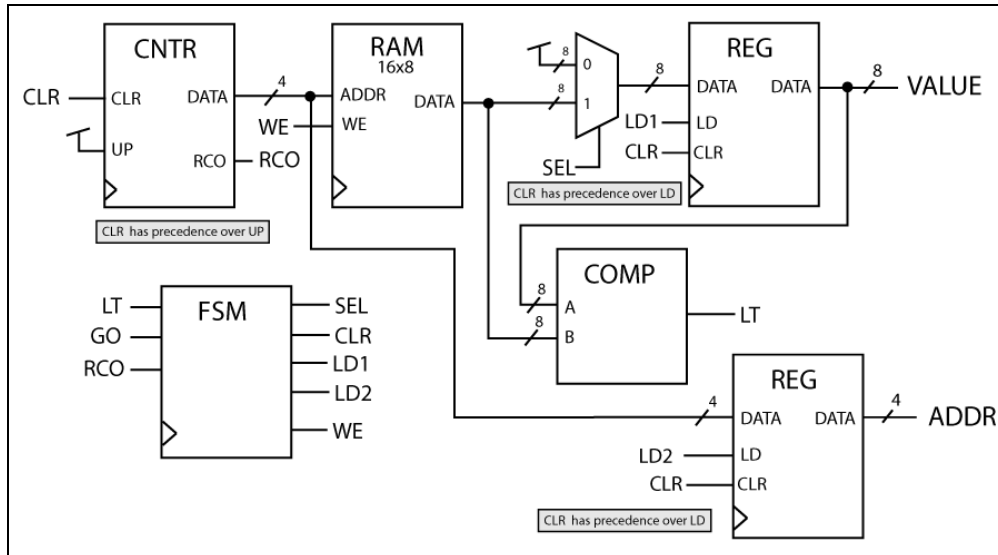


Figure 5.10: The lower-level BBD for this example.

Figure 5.11 shows the state diagram for this example. Although it looks quite busy, it's actually very structured, as the following items indicate.

- We model the LD1 and CLR as Mealy-type outputs in the “wait” state, which is arbitrary. We did this in order to save a state in the state diagram.
- When the GO signal asserts, the FSM clears the address register and counter, and loads the minimum value register with the largest possible 8-bit unsigned binary value.
- This circuit does not write to RAM, so we always disable the WE signal.
- The “search” state appears busy, but it's actually structured. Two things are happening. First, when the LT signal is not asserted, we don't load any new values to either register (LD1 & LD2 are not asserted). When the LT signal is asserted, we load the current address (the output of the counter) to the address register, and load the current RAM data output to the minimum value register. One of these two operations always happens no matter whether the RCO signal is asserted or not. When the RCO signal is asserted, that means the counter's output is at its maximum value and we must terminate the algorithm by transitioning back to the “wait” state.
- There are four arrows leaving the “search” state; each of these arrows has the four different possible combinations of the RCO & LT inputs.

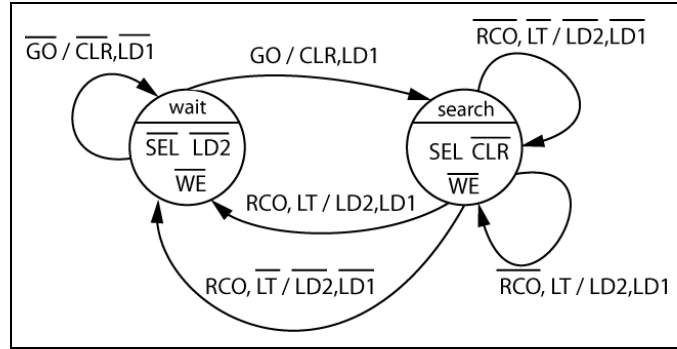


Figure 5.11: The state diagram associated with this example.

The FSM controls both the LD and CLR inputs for both registers, while we hardware the UP input of the counter to always count up. The GO signal is a form of external control. This circuit thus has external, circuit, and internal controls.

The counter has 16 unique count values that it steps through after receiving a GO signal. The first clock cycle causes the FSM to transition from the “wait” state to the “search” state. This circuit thus requires 17 clock cycles to locate the minimum value for the circuit.

Example 5.3: Design #3: Value Event Counter

Design a circuit that finds the number of times the value 0x47 appears in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The search for the given value begins when a GO signal asserts. The circuit’s output persistently shows the resultant count value until another assertion of the GO signal. Assume the circuit contains numbers in unsigned binary format. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit’s FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

Solution: This is another problem where we need to carefully choose the width of the output value. This problem asks that we count the number of value in the RAM that are equivalent to 0x47. The greatest count is when all the values in the RAM are 0x47, which is a count of 16. We thus require an output data width of five bits. Figure 5.12 shows the top-level BBD for this problem.

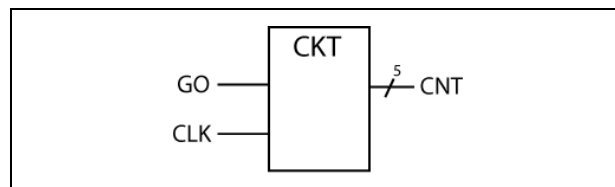


Figure 5.12: The top-level BBD for this example.

The next step in the solution is to create an inventory of the modules our solution requires; here is our module inventory thought process.

- We know the circuit requires a RAM, so we know the circuit then uses a counter to generate an address for the RAM. There are 16 values in the RAM, so the counter’s output is 4-bits wide.

- We are looking for the value of 0x47, which means we need to compare the data at each RAM location with that value. Our circuit thus requires a comparator.
- We must determine the number of times the 0x47 appears in the RAM, so the first thought may be that our circuit requires an accumulator. We could use an accumulator, but we can satisfy our circuit's needs with an event counter, which is a counter that increments when it detects a certain event. The event we are detecting is the presence of 0x47 in the RAM.
- We need a FSM to control our circuit.

Figure 5.13 shows the lower-level BBD for our solution. Here are a few interesting items in that BBD:

- The comparator hardwires one the “event” value to one of its inputs.
- We don't need to provide a note for the event counter regarding the precedence of the LD and CLR inputs; the FSM handles that aspect of the circuit.
- The CLR signal on the two counters are physically the same signal.
- The DATA input to the RAM is hardwired to zero; when we find the value of 0x47 at a particular address, the circuit writes 0x00 to that address location.

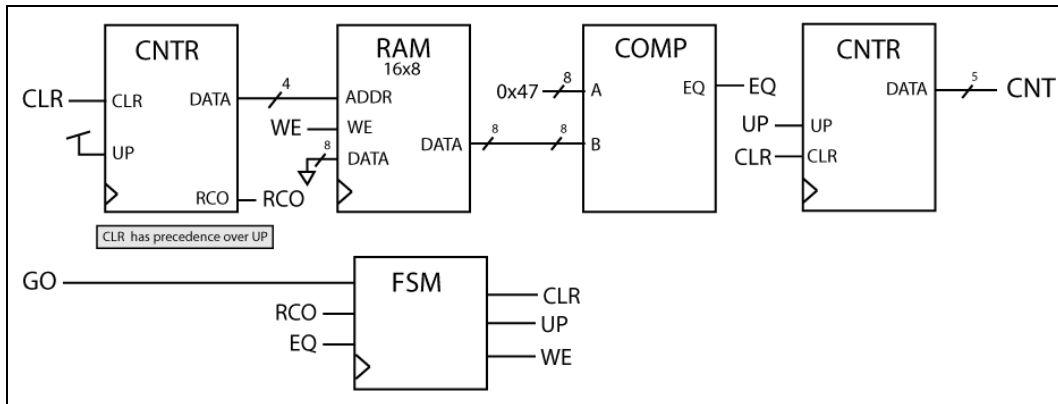


Figure 5.13: The lower-level BBD for this example.

Figure 5.14 shows state diagram for our solution. The state diagram looks rather busy, but once again, it is nicely structured. If you see and understand that structure, the state diagram seems relatively simple. Here the full story:

- We model the LD and CLR as Mealy-type outputs in the “wait” state, which is arbitrary. We did this in order to save a state in the state diagram.
- This WE input is always disabled in the “wait” state. The state of the WE signal in the “scan” state depends on the EQ input, where it writes a new value to RAM when the EQ is asserted, or does not change the RAM contents otherwise. We thus model the WE input as a Mealy-type output in the “scan” state and as a Moore-type output in the “wait” state.
- The “scan” state has four arrows leaving the state, where each arrow represents one combination of the two inputs (RCO & EQ).
- When the RCO is not asserted, the circuit either increments the count and clears that corresponding address in RAM, or it does nothing; it then transitions back to the scan state. When RCO is asserted, it performs the exact two actions, but the FSM then transitions to the “wait” state.

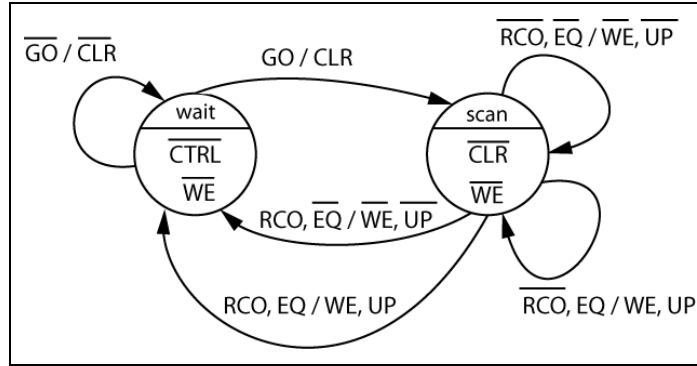


Figure 5.14: The state diagram associated with this example.

The FSM controls the LD, CLR, and WE inputs for the counters and RAM. The GO signal is a form of external control. Thus, this circuit has both circuit an external control.

The counter has 16 unique count values that it steps through after receiving a GO signal. The first clock cycle causes the FSM to transition from the “wait” state to the “search” state. This circuit thus requires 17 clock cycles to locate the minimum value for the circuit.

5.8 Memory Mapping

You can typically find the notion of memory mapping in microcontroller (MCU) and microprocessor-based (MPU) systems. The idea is that different areas of memory are typically used for different purposes in most digital systems. The notion of a MCU or MPU-based system generally implies that the system is becoming relatively complex. The segmenting of memory, or the designation of certain parts of memory to specific purposes, aids in the overall understanding of the system. This makes it well work looking at in this chapter.

The notion of memory mapping is an exercise in the study of binary and hexadecimal numbers. Even a simple digital system is large enough not to deal with binary values and ranges, so we quickly convert to hexadecimal notation to describe memory mapping. The use of hex notation is not complicated, but it is somewhat of a language all its own. The good news is that once you see a few tricks and work with it for a while, you’ll for sure find it rather straightforward.

The main idea behind memory mapping is to use multiple smaller chunks of memory space to create a larger memory space. The larger memory space is not “full” in all cases, as there may be areas of memory space that have no physical memory. Under these conditions, the thing we’re most interested in is the address ranges associated with a particular chunk of memory as it relates to the larger chunk of memory. The best approach to understanding these issues are with a few example problems. It grinds down to a binary math problem, which heavily implies tweaking around with powers of two.

Before we proceed, let’s show some important numbers regarding binary number ranges and their hexadecimal representations. Table 5.4 shows the relation between the number of address bits of a given memory and the associated address range. The first column in Table 5.4 shows the number of address bits associated with a given memory while the other three columns show the zero-based address ranges possible from those given address bits. Note that the decimal representations quickly become barely perceptible. We don’t even bother writing out the binary equivalents, as we would quickly inundate your brain with 1’s and 0’s.

There are a few other important things to realize about Table 5.4 The “Address Range” column provides the associated address range in an 8-digit hexadecimal format. Note the maximum address in any range is associated with all the address bits being at a ‘1’ value. This subsequently provides the “1→3→7→F” format associated with the first non-zero digit reading from left to right. Also note for both the third and fourth columns of Table 5.4 that the number ranges double as you proceed downwards in the table. This is a by-product of the underlying binary nature of memories.

# of Address Bits	Decimal Range	Address Range (hexadecimal)	Abbreviated Range
1	0-1	0-00000001	-
2	0-3	0-00000003	-
3	0-7	0-00000007	-
4	0-15	0-0000000F	-
5	0-31	0-0000001F	-
6	0-63	0-0000003F	-
7	0-127	0-0000007F	-
8	0-255	0-000000FF	-
9	0-511	0-000001FF	-
10	0-1023	0-000003FF	0-1K
11	0-2047	0-000007FF	0-2K
12	0-4095	0-00000FFF	0-4K
13	0-8191	0-00001FFF	0-8K
14	0-16383	0-00003FFF	0-16K
15	0-32767	0-00007FFF	0-32K
16	0-65535	0-0000FFFF	0-64K
17	0-131071	0-0001FFFF	0-128K
18	0-262143	0-0003FFFF	0-256K
19	0-524287	0-0007FFFF	0-512K
20	0-1048575	0-000FFFFFF	0-1M
21	0-2097151	0-001FFFFFF	0-2M
22	0-4194301	0-003FFFFFF	0-4M
23	0-8388607	0-007FFFFFF	0-8M
24	0-16777215	0-00FFFFFFF	0-16M
25	0-33554431	0-01FFFFFFF	0-32M
26	0-67108863	0-03FFFFFFF	0-64M
27	0-134217727	0-07FFFFFFF	0-128M
28	0-268435455	0-0FFFFFFF	0-256M
29	0-536870911	0-1FFFFFFF	0-512M
30	0-1073741823	0-3FFFFFFF	0-1G
31	0-2147483647	0-7FFFFFFF	0-2G
32	0-4294967295	0-FFFFFFFF	0-4G

Table 5.4: Number of bits and associated number ranges.

Example 5.4: Memory Mapping with Two Devices

Show the address ranges in both binary and hexadecimal associated with the use of two 64x8 memories to form one 128x8 memory.

Solution: The first part of this solution is to draw a diagram to enhance our understanding of the sparsely worded problem. Figure 5.15 shows a diagram we'll use to solve this problem. As you can see from Figure 5.15, we've virtually attached two 64x8 memories, which as the effect of forming a 128x8 chunk of memory⁸.

⁸ We could have created a 64x16 memory, but that is a topic we'll deal with in a later section.

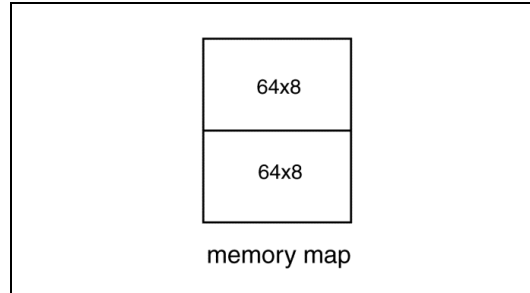


Figure 5.15: The diagram associated with Example 5.4.

The first thing to note is that a 64x8 memory uses 6 bits for its address lines. Indexing 6 bits into Table 5.4 shows that associated range is 0→3F, or “000000”→”111111”. The using two of these memories would create twice the capacity of a 64x8 memory, for an overall memory capacity of 128x8. In other words, you can model two 64x8 memories as one 128x8 memory. Indexing into Table 5.4, you can see that a 128x8 memory is associated with seven address bits. The only difference between these two memories in the 128x8 configuration is the most significant bit of the virtual 7-bit address; the six lower bit ranges are essentially equivalent. The final address value for the lower-order memory in Figure 5.16 is thus ‘0’ appended to the 64x8 range while the final address value for the higher-order memory is ‘1’ appended to the 64x8 range. In other words, pasting two memories together requires some other mechanism for differentiating between the two memories, which we do quite easily by adding one more bit in the most significant bit position.

Figure 5.16 shows the final solution to this example problem. The hexadecimal numbers on the left side of the diagram show the range of associated address values; the numbers on the right side show the binary equivalent to the hexadecimal values. This is a massively important diagram for several reasons. First, note how the 1-bit values change on the memory boundaries: the MSB becomes a ‘1’ and the other bits all become ‘0’. Secondly, notice that the original 6-bit ranges are the same for both the lower and higher-order memories.

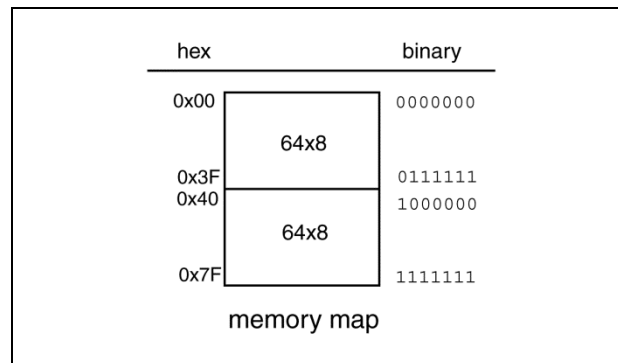


Figure 5.16: The final solution to this example.

Example 5.5: Memory Mapping with Four Devices

Show the address ranges in both hexadecimal and binary associated with the use of four 8Kx16 memories to form one 32Kx16 memory.

Solution: The good news is that this problem is similar to the previous example. The first step in this problem is to draw a diagram to help up see what exactly the problem is asking us to do. Figure 5.17 shows a block diagram that is modeling four 8Kx16 individual memories as one 32Kx16 memory.

Our next step is to examine Figure 5.17 and figure out some of the metrics we'll need to use to complete this problem. First, an 8K memory requires 13 address lines while a 32K memory requires 15 address lines. What this tells us is that the most significant two bits in the 15-bit address are what we'll need to use to differentiate between the least significant 13 bits. Also from Figure 5.17 is the notion that the 13-bit range for an individual memory is [0x0000,0x1FFF]. We've seen this problem before; each 8Kx16 chunk of memory has a different upper two bit; these bits are "00", "01", "10", and "11". Yep, it's that binary sequence yet again. We complete this problem by pasting this set of bits in front of the address range values given for the 8K memory: [0x0000,0x1FFF]. Figure 5.18 shows the final solution to this problem with gory details included.

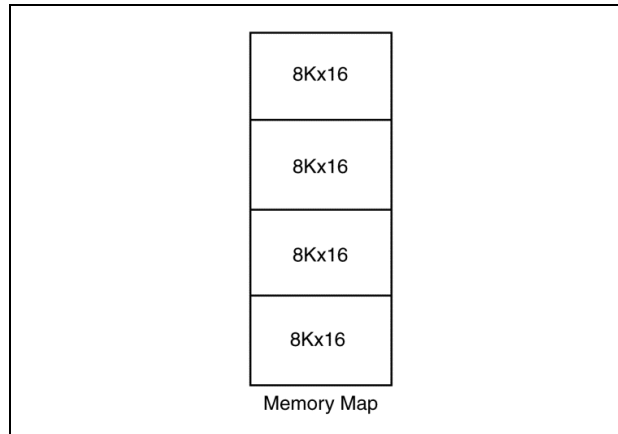


Figure 5.17: The diagram associated with this example..

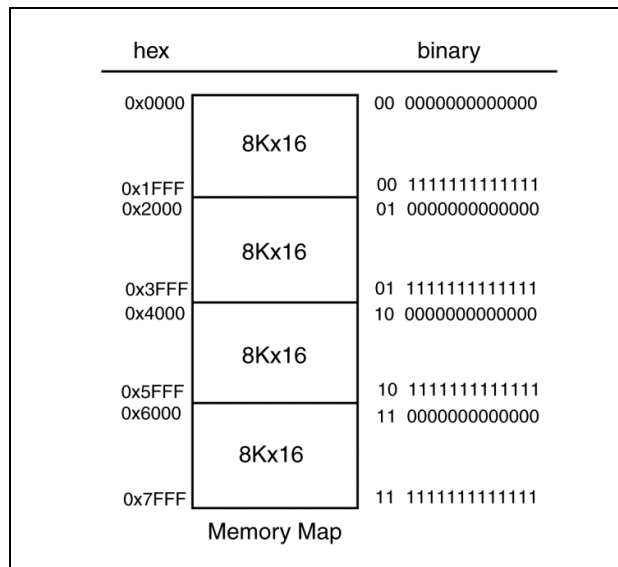


Figure 5.18: The final solution to this example.

5.9 Memory Organization

Discrete memory devices do not come in every possible configuration and capacity. This means if you need some type of special capacity, you're going to need to synthesize it from various memories of smaller capacities. The architecture of the overall memory created from smaller memories is what we refer to as memory organization. If a digital system you are working with has a specified memory capacity but it is created from

many smaller memories, you immediately know the capacity of the memory but you don't necessarily know anything about the organization of memory.

The most appropriate title for this section would be something like: "using many smaller memories to create a larger memory", but that title would run across two lines. The choice of "memory organization" attempts to reflect the notion that many times you'll need to satisfy your particular memory needs by configuring many smaller memory devices in a system such that it creates some particular form of a larger memory. You could use many different approaches to doing this; this section outlines only one major approach.

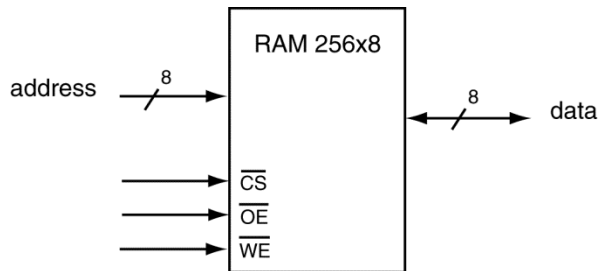
If you need to create a larger capacity memory from a many smaller capacity memories, you can do it in two basic ways: 1) increase the number of words that the memory can store, or 2) you can increase the effective width of the words stored in that memory. You could also do both, but we'll not deal with such problems in this section. The two subsections within this section deal with increasing word size and increasing the number of words store.

5.9.1 Extending Memory Word Length

Extending word length is the most straightforward approach to building larger capacity memories. The fun starts when we extend the number of addressable words in the next subsection. Extending word length is more straightforward because it only requires special circuit configuration and but typically does not require additional circuitry as does extending the addressable memory space. The best approach to explaining the concept of extending word length is through an example problem.

Example 5.6: Extending Memory Word Length

Show a circuit diagram that uses two of the following listed 256x8 RAMs to effectively create one memory with a 256x16 capacity. Assume the memory has bi-directional data lines. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



Solution: This problem is straightforward because we don't need to tweak the address lines in order to address all the possible words in the 256x16 memory. Figure 5.19(a) shows one possible architecture for the solution to this example. Here are the worthy things to note from the solution.

- The two 256x8 RAM devices share all the same control lines as both RAMs always need to act simultaneously for both reads and writes.
- The final circuit comprises of the two 256x8 RAM sharing the address lines. This is possible because the overall number of words for the larger capacity memory does not change.
- The money shot of the solution lies in the interpretation of the outputs of the two 256x8 RAMs. The output of each 256x8 RAM becomes half of the final data width for the 256x16 RAM. In other words, each 256x8 RAM contributes eight bits to the final 16-bit output of the 256x16 RAM.
- The schematics in Figure 5.19(a) and Figure 5.19(b) are functionally equivalent. Sometimes it is clearer or easier to draw the schematic one way rather than the other.

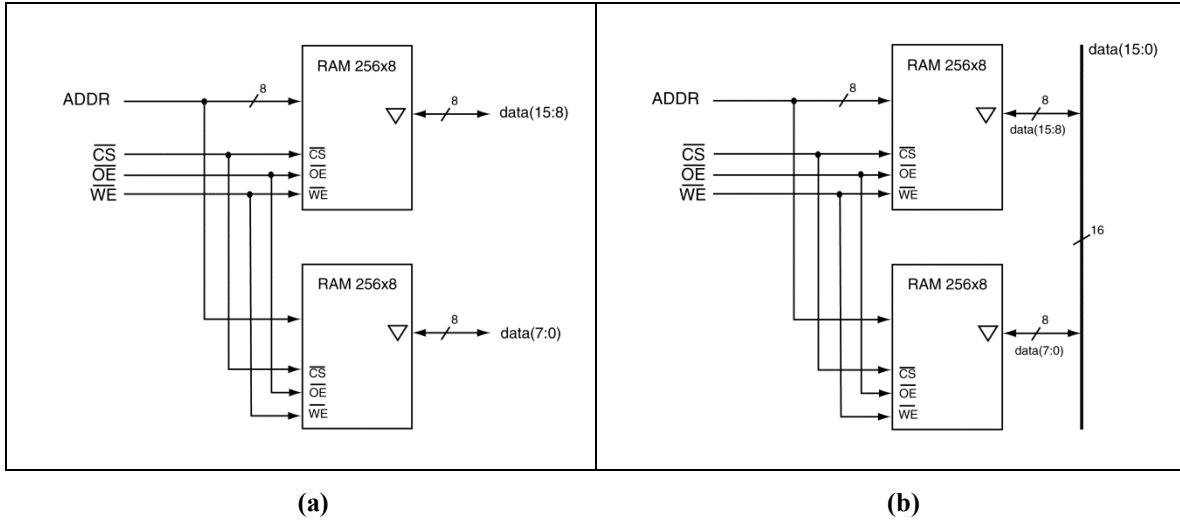


Figure 5.19: Two different ways of representing a 256 x 16 memory.

5.9.2 Extending Memory Address Space

Although using multiple memories to extend the data width of an aggregate memory is straightforward, using multiple memories to extend the overall address range can be slightly tricky. This section describes these issues and provides some options for solutions.

Figure 5.20 highlights the main issue involved with extending address space in a multiple memory system. For this problem, we don't consider extending the data width. As you can see from Figure 5.20, the issues lie in how to handle the addressing needs of the individual memories. Figure 5.20 shows four $M \times N$ memories; we intend to include these four memories into one system. The resulting memory space is thus $4M \times N$. In order to have sufficient address space to address the $4M \times N$ overall memory, we must increase the number of address lines on the system by two. In this way, the two extra address bits are sufficient to address one of the four internal memories.

One important thing to notice about Figure 5.20 is that the individual memories have tri-stated outputs. The characteristic helps define the overall problem: When we present the memory system with an "M+2" address, we need only one of the internal memories to drive their data onto the data lines. We somehow need the "M+2" address lines to effectively apply an address that addresses the full memory space but only actuates one of the internal memories. We only want one internal memory activated because the internal memories are sharing one set of data lines.

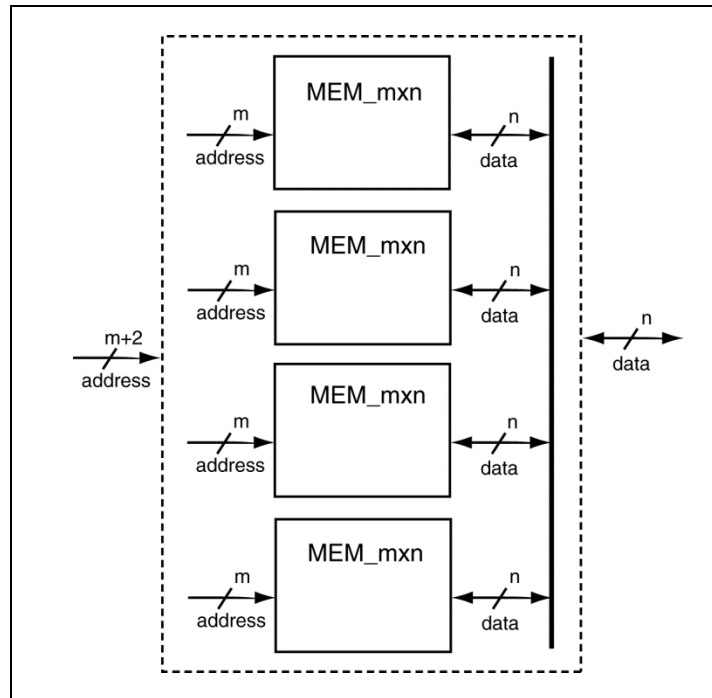


Figure 5.20: An overview of the extending address space dilemma.

Figure 5.21 facetiously shows an overview of our approach to the extending memory space. The approach we'll take is to insert circuitry into the "Magic_CKT" module. This module is then responsible for translating the full "M+2" address space into m address lines and an appropriate number of control lines. We'll of course need at least two control lines, but there could be more depending on the control requirements of the internal memories. We'll present some relatively straightforward examples highlighting an approach based on relatively simple control requirements of the internal memories.

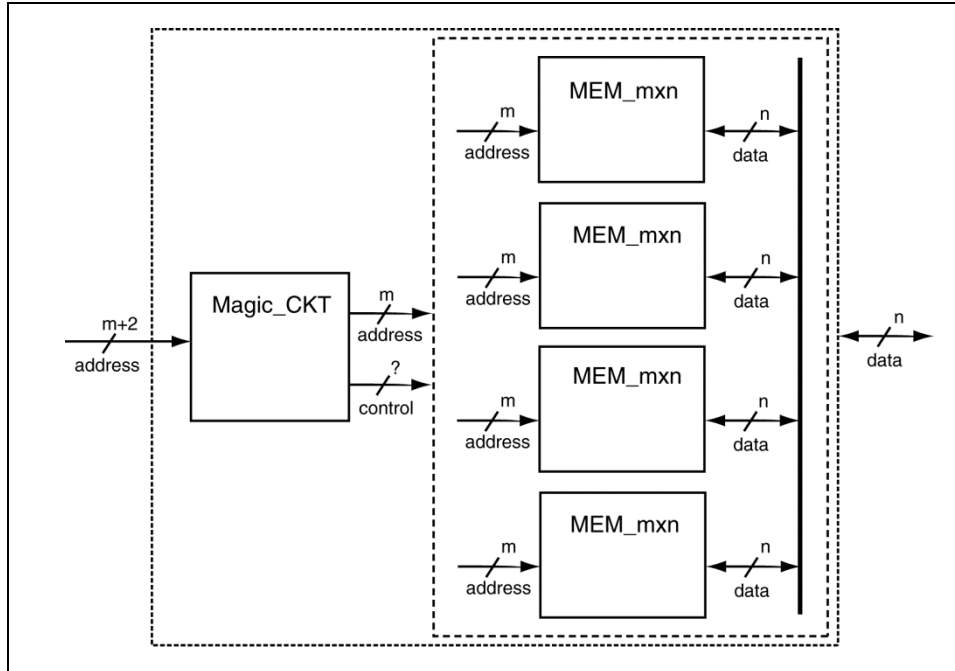
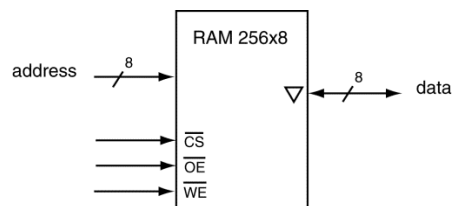


Figure 5.21: An overview of the solution to the extending address space dilemma.

Example 5.7: Extending Memory Address Space

Use as many of the following RAMs as you need to create a memory with a 512x8 capacity. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



Solution: Figure 5.22 shows the first part of the solution to this problem, which is to draw a black box diagram of the final solution. In order to create an address space that accesses 512 memory locations, we need nine address lines. Since the underlying memories can address 256 memory locations, we’ll need two of these memories to access the required 512 locations.

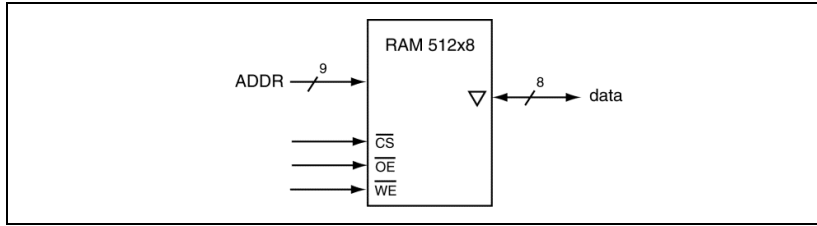


Figure 5.22: The black box diagram for the solution to this problem.

Relative to Figure 5.22, we'll need to add circuitry that divides the address lines between the lines that are common to the each 256x8 device and the extra lines required for the larger memory configuration. For this problem, we'll have eight address lines for the underlying memory devices and one extra address line to differentiate between the underlying memories based solely on the "8+1" address lines. The approach we'll take is to insert a standard 1:2 decoder to handle the extra address line. Standard decoders are ideal devices for this application as they have only one active output at any given time. Figure 5.23 shows the final circuit solution to this problem. Here is some happy information regarding the solution in Figure 5.23.

- The decoder in Figure 5.23 is a standard 1:2 decoder with active-low outputs. We chose a decoder with active-low output in order to have those outputs properly interface with the active-low CS inputs of the individual memory devices. This standard decoder has only one active output a time; being that the outputs are active-low, we can consider the outputs of the decoder as "one-cold"⁹. This configuration provides the controls to enable only one memory device at a time. The input to the 1:2 decoder thus becomes the most significant bit in the overall 9-bit address. When the input to the decoder is '0', the decoder actuates the top memory; when the input to the decoder is '1', the decoder actuates the lower memory. Recall from the problem statement that when the memory's chip select is not active, the device effectively provides high-impedance to the data lines.
- The two memory devices share the lower eight address lines. Once again, the CS signal determines which memory device is active based on the ninth address line (the input to the decoder).
- The two memory devices share the OE and WE lines. The notion here is that some outside device utilizes these controls as required. There are generally no loading issues associated with these signals as only one memory device is actuated at a given time.
- The total number of address lines is independent of the size of the standard decoder. The characteristic that determines the minimum size of the standard decoder is the number of memory devices internal to the overall memory system. In other words, the standard decoder's responsibility is to use the appropriate address bit(s) to actuate the proper memory associated with all the address lines.

⁹ For example, a decoder with one-cold outputs has one output at a '0' state and all other outputs at a '1' state.

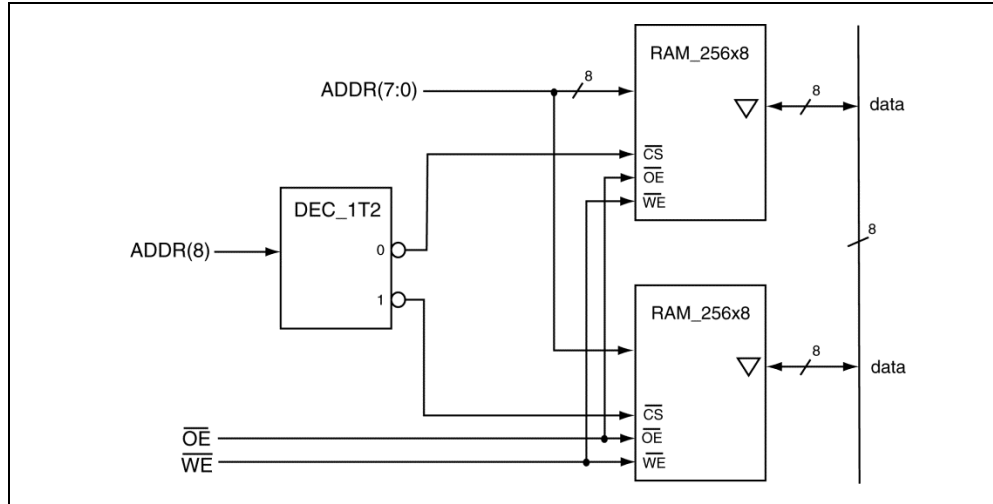


Figure 5.23: The final circuit solution to this example.

The final part of this solution is to generate a memory map that shows the overall memory space as well as the addresses associated with the underlying memory modules. The solution to this part of the problem is similar to the memory space discussion of a previous section. Figure 5.24 provides the final solution to this example with some supporting notes to follow.

- In accordance with Figure 5.24, the nine-bit addresses for the overall device start at all 0's and end with 0x1FF, or all 1's.
- Figure 5.24 shows the addresses of the underlying memories in both hexadecimal and binary. Note that the binary addresses have a space inserted in the number to highlight the notion that the ninth address bit is used to delineate between the two memories.

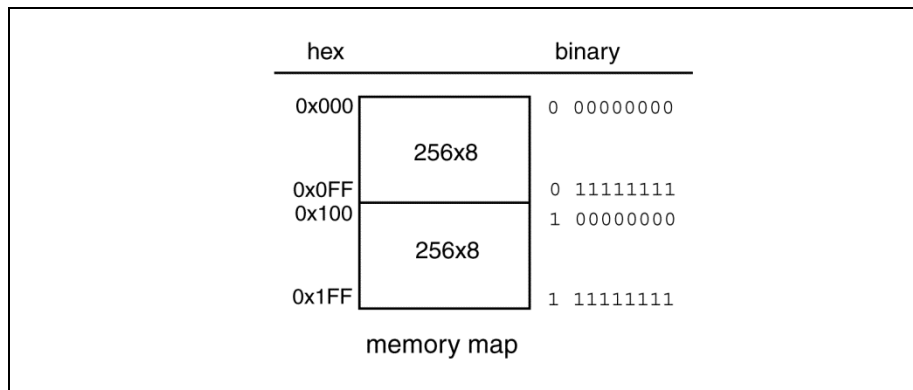
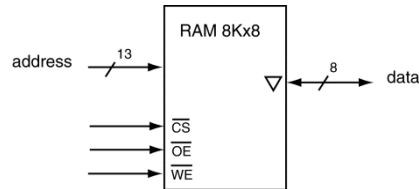


Figure 5.24: The memory map associated with this example.

Example 5.8: Extending Memory Address Space with More Devices

Use as many of the following RAMs as you need to create a memory with a 32Kx8 capacity. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



Solution: The first thing to note is that this problem is very similar to the previous example problem. That being the case, the first step in this problem is to draw a diagram of the final high-level object. Figure 5.25 shows the high-level schematic diagram associated with this problem. The first thing we see is that we need four devices with 8K worth of memory space to create a memory with 32K addressable memory locations. This means that we’ll need four 8Kx8 devices in the final circuit. Using Table 5.4 we see that a memory with 32K memory locations requires 15 address lines.

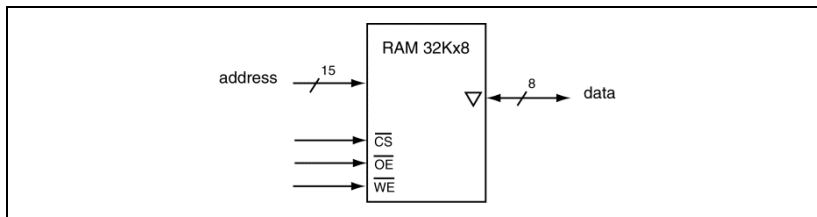


Figure 5.25: The black box diagram for the solution to this example.

The biggest similarity between this problem and the previous problem is with the use of a standard decoder to handle the “magic_ckt” in Figure 5.21. The standard decoder in this problem is slightly different in that it needs to choose between four different memories. This simply requires that the final circuit use a 2:4 standard decoder in place of the 1:2 decoder of the previous problem.

Figure 5.26 shows the final solution to the circuit portion of this problem. Note that this solution is similar in overall structure to the solution of Figure 5.23, with the main difference being that we now need to choose between four discrete memory devices instead of the two devices. A standard 2:4 decoder easily handles this task by effectively using the two most significant bits of the 15-bit address lines as the select inputs to the standard decoder.

The memory map in Figure 5.27 shows the second part of this solution. Figure 5.27 highlights the mechanics of this solution with the binary numbers on the right side of the memory map. As you can see, each of the lower 13-bits for the individual memories are the same; only the two most significant memory bits differentiate the 15-bit address. As you would expect, since the circuit must choose between activating one of four memory devices, the circuit must use a minimum of two bits for this task.

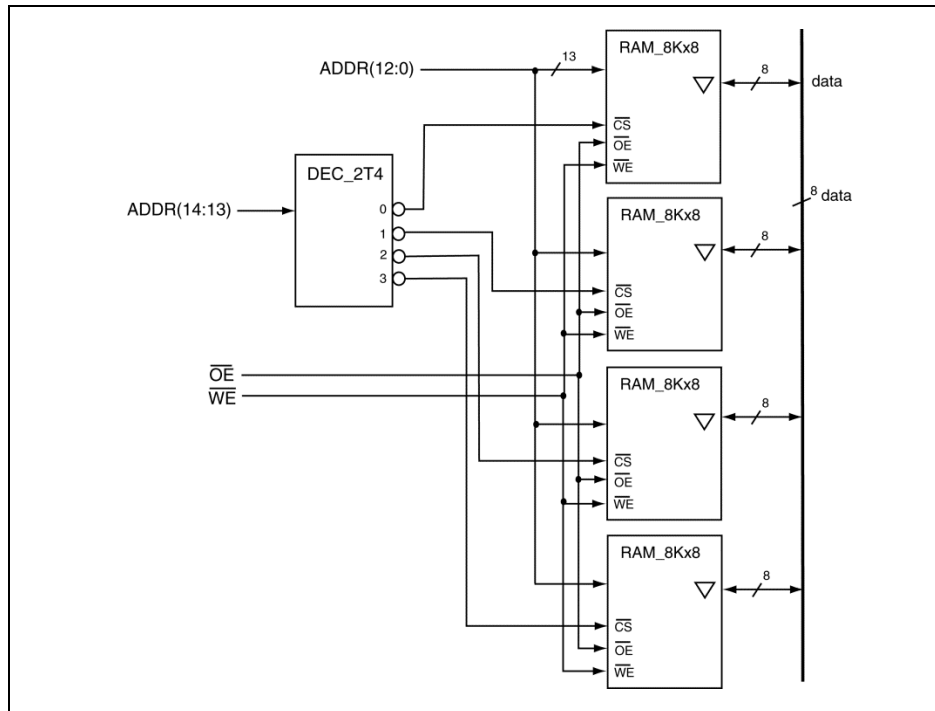


Figure 5.26: The circuit diagram solution for this problem.

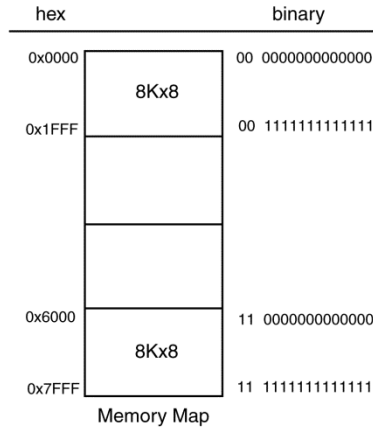
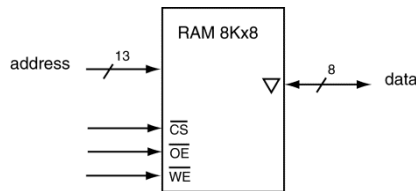
hex	binary
0x0000	00 000000000000
0x1FFF	00 111111111111
0x2000	01 000000000000
0x3FFF	01 111111111111
0x4000	10 000000000000
0x5FFF	10 111111111111
0x6000	11 000000000000
0x7FFF	11 111111111111

Memory Map

Figure 5.27: The memory map associated with problem.

Example 5.9: Circuit Design for a Memory Map

Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only 2:4 standard decoder, and 2) using only one 1:2 standard decoder. Use two of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



Solution: This problem is slightly different in that it does not use the entire memory space listed in the problem description. In other words, the memory space has is divided into four sections, but only two of those sections contain active memory. Because there are only two sections with active memory, the solution only requires the use of a 1:2 standard decoder. We’ll do this problem in two different ways in order to highlight the differences in using standard decoders of different size.

Figure 5.28 shows the most straightforward solution to this problem, which is to use a 2:4 decoder. The good thing about this solution is that each address in the 32K address space associated with the memory map is unique. The second part of this problem describes this issue further. The downside of the solution in Figure 5.28 is that the 2:4 standard decoder is partially unused. This implies the device is probably physically bigger than it needs to be, which may or may not be an issue¹⁰. Another possible upside of this solution is that it facilitates a later possible expansion of the memory map in that all the support hardware is in place; expansion would thus be a matter of dropping other 8Kx8 memory devices.

Figure 5.29 shows the solution for part 2) of this example. This solution replaces the 2:4 standard decoder from the part 1) solution with a 1:2 standard decoder. The upside of this solution is that it uses a smaller decoder. The possible downside of this solution is that each memory location in each 8Kx8 RAM is accessible using two different addresses. The problem results from effectively no longer using the ADDR(13) in the circuit solution. Because the solution is not using this address, the address bit is effectively a “don’t care”. As a result, the addresses of 0x1FFF and 0x17FF effectively access data from the same location in the lower-order 8Kx8 memory. In other words, for this solution, 0x1FFF and 0x17FF access memory location 0xFFF in the lower-order memory. While this certainly is an issue to consider, it may or may not be a problem for your particular system.

The main possible problem with this particular circuit design is that the memories may be driving the data bus at times where the memory is not being access. Recall that this is a relatively simple memory that uses the chip-select input signal to actuate the memory; when the memory is not actuated, the memory outputs are effectively in a high-impedance state.

¹⁰ Keep in mind that if you plan on modeling the 2:4 decoder with VHDL and then synthesizing it, the synthesizer will probably mitigate the size issue of the standard decoder.

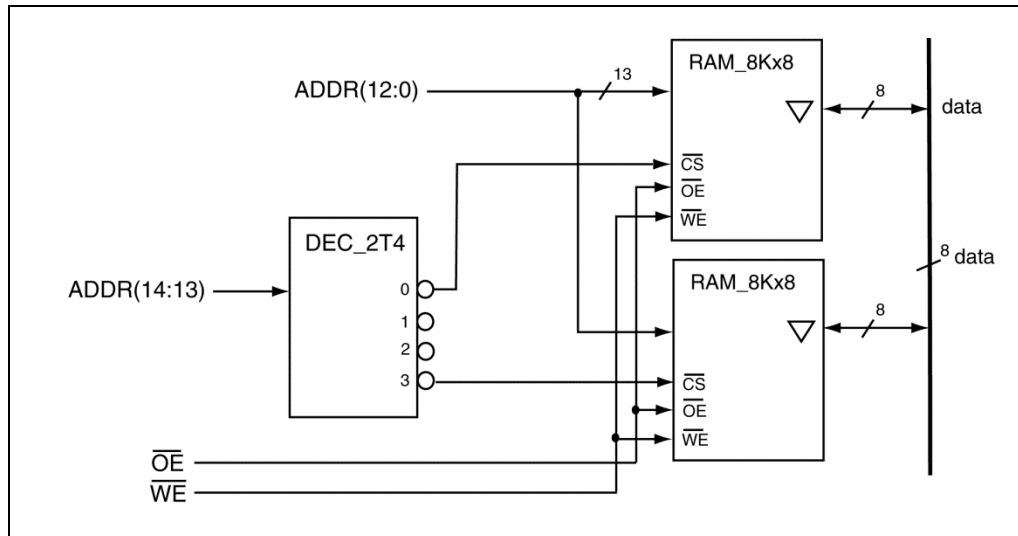


Figure 5.28: The black box diagram solution for solution for this example part 1).

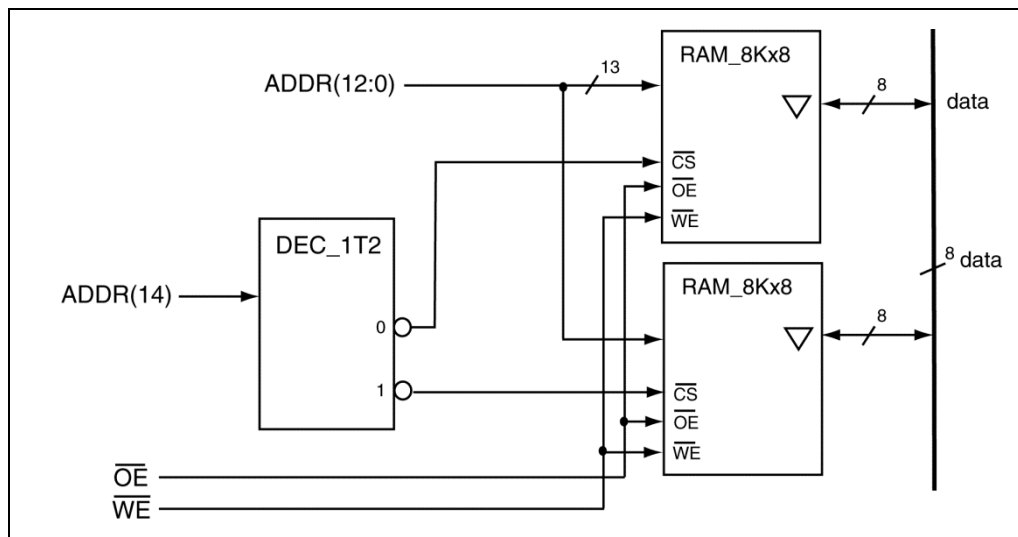
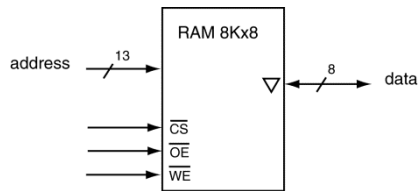


Figure 5.29: The black box diagram solution for this example part 2).

Example 5.10: Modeling a Memory Map

Design a circuit that implements the following memory map using a 3:8 standard decoder. Use three of the RAM devices listed below in your solution. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



Solution: This problem presents a slightly different twist beyond the previous problems. The memory map is not full and the memories in the memory map are not contiguous. Neither of these characteristics are a big deal, they do present some interesting challenges. Thus, Figure 5.30 shows the first step to this solution, which is to draw the black box diagram of the high-level final circuit.

The most useful piece of information presented in Figure 5.30 is the notion that the circuit requires 16 address lines. We know this from examining the memory map in the problem statement. What we look for in problems like this the minimum number of address bits required to solve the problem. The worst case in the problem is the address with the most number of bits. In this problem, we can see that 0x4000 and 0x5FFF only require 14 bits, but every other listed address requires 16 bits. Thus, our final circuit requires 16 bits for the address lines. However, be sure to keep in mind that we won't be using every address in the 16-bit range, which is why we did not state any memory capacity information in the circuit diagram of Figure 5.30.

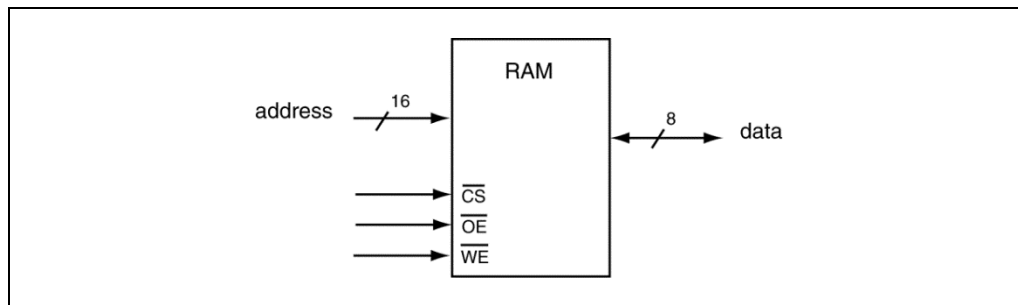


Figure 5.30: The black box diagram solution for this example.

The next part of this solution is broken into two parts. The best approach is to first list all the address ranges in both hex and binary for both the segments that include memory as well as the segments that are not associated with a memory device. The 8K address space requires 13 bits; the “magic_CKT” portion of the circuit handles the other three bits. In all likelihood, we'll be able to handle the “magic_CKT” portion of the circuit using a 3:8 standard decoder as stated in the problem. Figure 5.31 shows the result of this step.

Figure 5.31 conveniently shows the 16-bit addresses divided into 3-bit and 13-bit segments in the binary address listing. From this listing, you can see that the hex address values from the problem statement fall on 8K

boundaries. This is good news as this allows us to easily use the 3:8 standard decoder in the solution. The important information in Figure 5.31 includes an accounting for every address in the 16-bit address space. Note that a 16-bit address space is associated with a 64K memory. Since we're implementing this memory with eight 8K memory device, there should rightly be eight 8K segments in Figure 5.31. The truth is that some of unused segments represent more than one 8K with of address.

For example, the first segment listed in Figure 5.31 has no associated memory. This address range scans two 8K segments worth of memory, a fact that Figure 5.31 does not explicitly show. In reality, the first segment in the first address range covers 0x0000 through 0x1FFF while the second 8K segment covers 0x2000 through 0x3FFF. The other two ranges that do not have memory only cover one 8K segment.

The important thing to note in Figure 5.31 is that in the binary numbers associated with the memory, the range of the least significant 13 bits of the address is always 0x0000 to 0x1FFF. This means that the most significant upper three bits differentiate between the 8K memory spaces. We know that a 64K memory can be created from eight 8K memories; in this problem, we're only interested in the 64K space while the total memory capacity is only 24K. The most significant three bits in the binary address of Figure 5.30 then become the inputs to the associated 3:8 standard decoder. Figure 5.32 shows the final solution to the first portion of the problem.

hex		binary
0x0000		000 0000000000000
0x3FFF		001 1111111111111
0x4000	8Kx8	010 0000000000000
0x5FFF		010 1111111111111
0x6000		011 0000000000000
0x7FFF		011 1111111111111
0x8000	8Kx8	100 0000000000000
0x9FFF		100 1111111111111
0xA000		101 0000000000000
0xBFFF		101 1111111111111
0xC000	8Kx8	110 0000000000000
0xDFFF		110 1111111111111

Memory Map

Figure 5.31: The full memory map (all addresses listed) for this example.

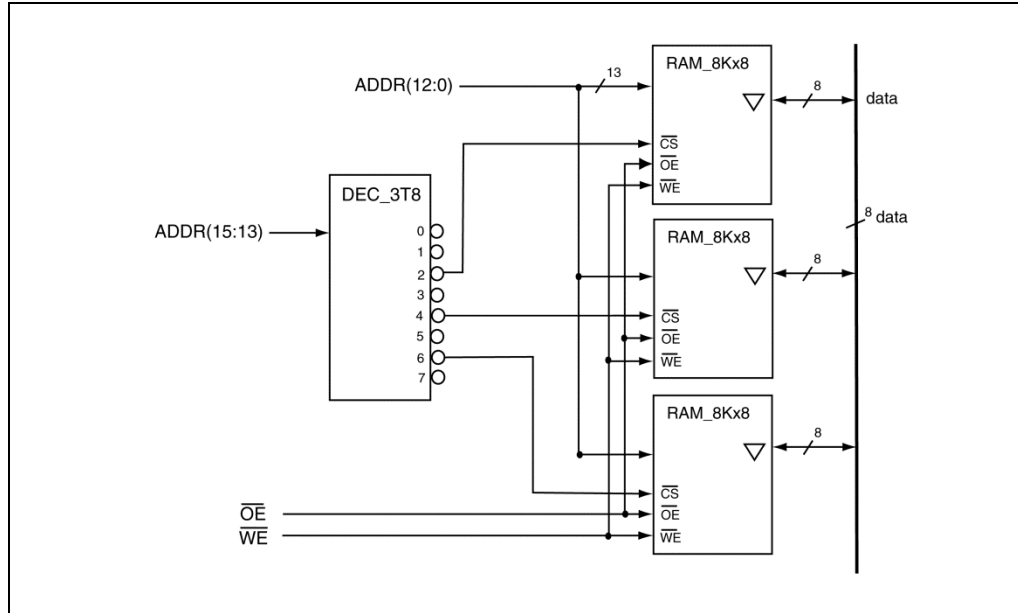


Figure 5.32: The black box diagram solution for first part of this problem.

The potential problem with the circuit solution in Figure 5.32 is the notion that the 3:8 standard decoder is vast overkill for the problem. The problem could have used a 2:4 standard decoder since it only needed to control three memories. On the other hand, if we had used a 3:8 decoder, it would have required extra circuitry in addition to the decoder in order to make the address actuate the correct memories.

Figure 5.33 shows the block diagram for the solution to the second part of this example. You can see from this diagram that our mission is to design a circuit that implements the “GEN_DEC” portion of the circuit. As the name of the box implies, our solution is simply a generic decoder. This decoder has three inputs and three outputs; the inputs are the most significant address lines while the outputs actuate the appropriate discrete memory when the 16-bit address conditions are correct.

The solution to the second portion of the problem is approaching trivial once you realize all two things. First, the solution is a generic decoder. Second, the information presented in Figure 5.33 provides us with all the information we need to create the required generic decoder. Note that in Figure 5.32, the standard decoder actuates on output only when the inputs are “010”, “100”, and “110”. More specifically, “010” actuates CS2, “100” actuates CS4, and “110” actuates CS6. If none of these three addresses is selected on the input lines, the module’s outputs turns off all the discrete memory devices.

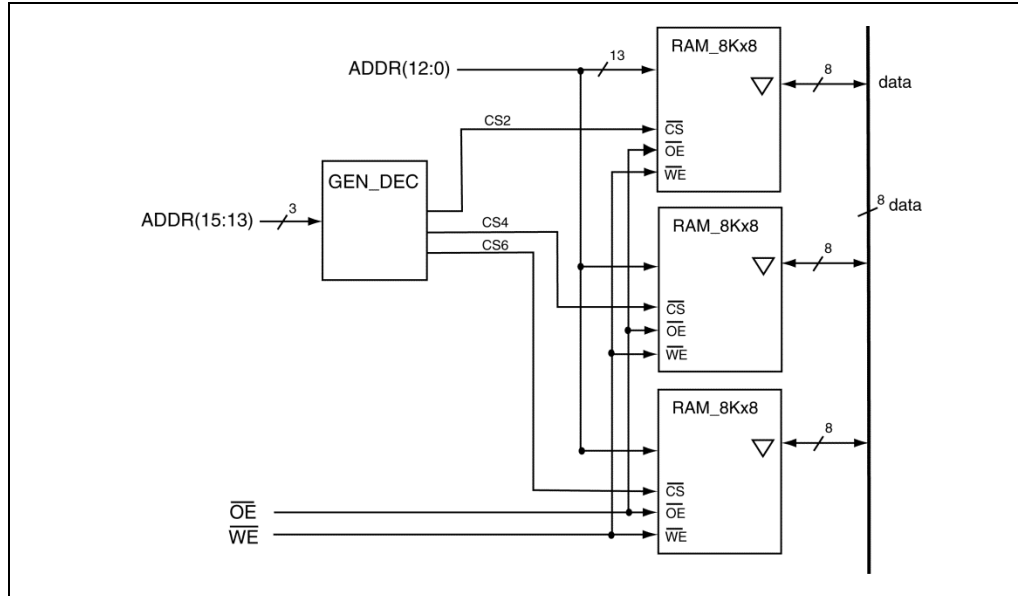
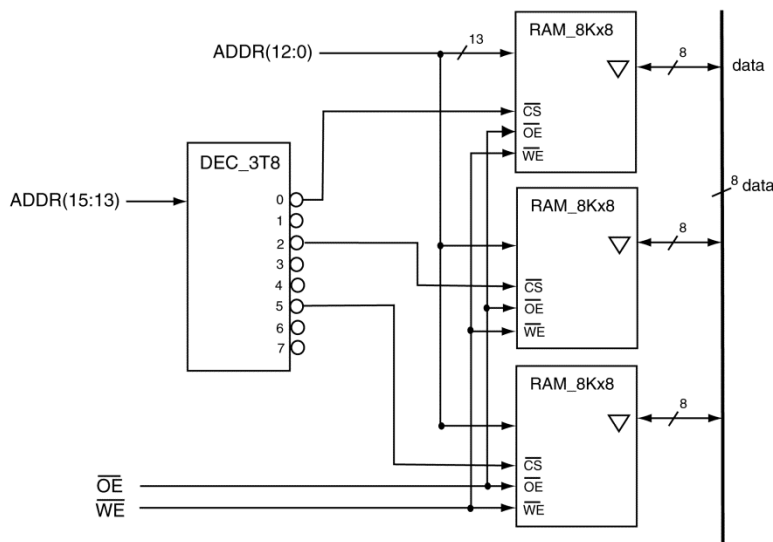


Figure 5.33: The black box diagram for the second part of the solution to this example.

Example 5.11: Generating a Memory Map From a Circuit

Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



Solution: This problem is the same old memory mapping problem but in the reverse order. This problem provides a circuit and you are responsible for generating a memory map. The first thing to notice about this problem is that there are three 8K memories in the circuit. The second thing to notice is that the problem uses a

standard 3:8 decoder for the “magic_CKT” part of the circuit. Thus, the inputs to the standard decoder are effectively the three most significant address bits (15:13), which implies the circuit can possibly address up to a 16-bit address space. The problem only uses 3/8 of the total possible address space, or 24K.

Each 8K RAM shares the same 13 bits of address lines. This means the address range for any 8K RAM by itself is 0x0000→0x1FFF, with the standard decoder handling the other three bits. The best approach to take for this problem is to note that the 64K address space is divided into eight 8K segments, but the circuit uses only three of the possible eight segments. The first used segment is associated with the three most significant address bits of “000” as indicated by that RAM’s chip select being connected to the ‘0’ output of the standard decoder. This makes the range for the first segment 0x0000→0x1FFF. The next 8K segment is not used; the starting address of this unused segment is one greater than the last valid address associated with the previous segment, or 0x2000. The ending address of this unused segment is 0x1FFF greater than 0x2000, or 0x3FFF.

You can continue this same type of analysis for all the other segments in the problem. The only slightly tricky thing to note is that the segments associated with 3-4 and 6-7 represent two 8K segments, which means the range effectively has 16K worth of segment space, or 0x0000→0x3FFF. After you realize all of this, the problem becomes somewhat of a math problem. Figure 5.34 shows the final solution to this problem with the addresses listed in both hexadecimal and binary for your viewing convenience.

hex		binary
0x0000	8Kx8	000 0000000000000
0x1FFF		000 1111111111111
0x2000		001 0000000000000
0x3FFF	8Kx8	001 1111111111111
0x4000		010 0000000000000
0x5FFF		010 1111111111111
0x6000		011 0000000000000
0x9FFF	8Kx8	100 1111111111111
0xA000		101 0000000000000
0xBFFF		101 1111111111111
0xC000		110 0000000000000
0xFFFF		111 1111111111111

Memory Map

Figure 5.34: The final memory map for this example.

5.10 Digital Design Foundation Notation: RAM

We consider the RAM to be a Digital Design Foundation module. The RAM is a controlled circuit. Figure 5.35 shows the digital design foundation notation for the counter. This foundation module is both data inputs and data outputs, both of which are the same width. We use a simple device for the foundation model and consider read operations to be asynchronous and write operation to be synchronous. The WE signal controls whether the device is reading or writing, where WE is asserted for write operations and unasserted for read operations. We consider ROMs to be a subset of RAMs; ROMs are not able to write. Table 5.5 shows the foundation description for the RAM.

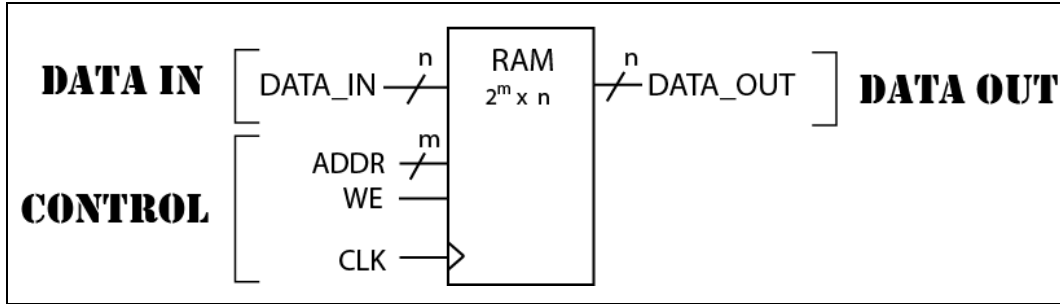


Figure 5.35: Typical data, control and status signals for RAM. .

	Signal Name	Description
INPUT DATA	DATA_IN	Data to be synchronously written to RAM.
OUTPUT DATA	DATA_OUT	Data stored in the RAM at the address given by the ADDR input.
CONTROL	CLK	The CLK signal synchronizes the writing of data to the RAM
	ADDR	The RAM stores the value of IN_DATA at the address associated with the value of ADDR on the active clock edge (synchronously) when the WE signal is asserted.
	WE	When asserted, allows the loading of DATA_IN to the RAM location specified by ADDR, which is a write operation. When unasserted, the RAM outputs the data stored at the location specified by the WE input.
STATUS	n/a	-

Table 5.5: The foundation description for a RAM.

5.11 Chapter Summary

- Memory is a form of a sequential circuit, but we further divide memory into two categories: “incidental memory” and “structured memory”. Incidental memory refers to items such as flip-flops and registers (relatively small) while structured memory refers to larger capacity regular structures.
 - There are many type of memory in digital-land, but we can roughly classify them all as either ROM or RAM. ROM is “read only” memory while RAM is “random access” memory. Both of these memories have the random access attribute in that all of the data on the devices is accessible in the same amount of time. ROMs are considered non-volatile while RAMs are not. RAMs can be both written to and read from while ROM can only be generally read from.
 - The notion of reading from a memory, or a memory READ, consists of making the data within the memory at a given address available to entities external to the memory. Memory reads generally do not alter the data stored in the memory. The notion of writing to a memory, or a memory WRITE, consists of overwriting data contained in the memory at a given address with data provided by some entity external to the memory.
 - Interfacing with memory generally requires tweaking one the three types of I/O associated with memory. The three types of memory I/O are address lines, data lines, and control lines. The address lines provide an index into the memory and allow access to a particular chunk of data stored in memory. The data lines provide a path for data to flow into (write) or out of (read) memory. The control lines provide a structured approach to read from and/or writing to the memory device.
 - Memories are generally rated by the capacity (how many bits they can store) and the speed (how fast you can read and/or write the memory). The term “word” is used to refer to the smallest chunk of memory available at a given address in the memory. Memory capacity can be stated in bits or words; any other approach is suspect as it can be misleading
 - Memories typically store two raised to an integral power number of words. The integral power in this case is the number of address lines on the memory. The number of address lines is sometimes referred to as the width of the address bus.
 - Memory speed is rated by how fast you can read from it and/or write to it. The term “read access time” refers to how fast you can read from a memory. The term “write cycle timing” refers to how fast you can write data to a memory. The term “memory bandwidth” refers to the maximum amount of data going to and coming from a particular memory in a given amount of time.
-

5.12 Chapter Exercises

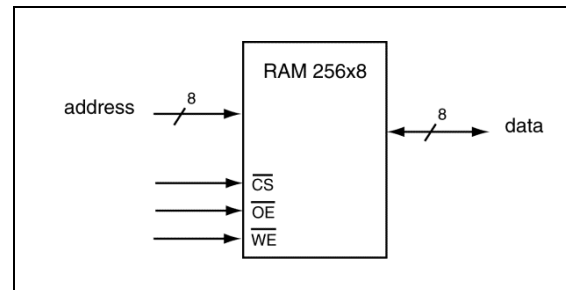
- 1) In your own words, describe what is meant by the term “random access” in the context of computer memories
- 2) In your own words, describe what is meant by the term “volatile” in the context of computer memories.
- 3) In your own words, describe the accepted functional differences between RAM and ROM.
- 4) In your own words, explain how read and write access times affect the bandwidth of a given memory.
- 5) Describe a circuit situation where having a large memory bandwidth would be important.
- 6) Faster memories are typically more expensive than slower memories. Speculate on why you feel this would be the case.
- 7) A given RAM capacity is specified as 1Kx24.
 - a) List the capacity of this RAM in both bits and bytes.
 - b) List the number of address lines this RAM would contain.
- 8) A given RAM capacity is specified as 1Kx32.
 - List the capacity of this RAM in both bits and bytes.
 - List the number of address lines this RAM would contain.
- 9) A given RAM capacity is specified as 8Kx32.
 - List the capacity of this RAM in both bits and bytes.
 - List the number of address lines this RAM would contain.
- 10) A given RAM capacity is specified as 16Kx24.
 - List the capacity of this RAM in both bits and bytes.
 - List the number of address lines this RAM would contain.
- 11) Fill in the missing information in the following table.

Memory Specification	Address Bus Width	Memory Capacity in Bits	Memory Capacity in Bytes
256 x 8			
256 x 24			
1K x 16			
2K x 8			
8K x 32			
32K x 12			
64K x 16			
256K x 8			
1M x 16			
4M x 32			
8M x 8			
64M x 48			
128M x 32			

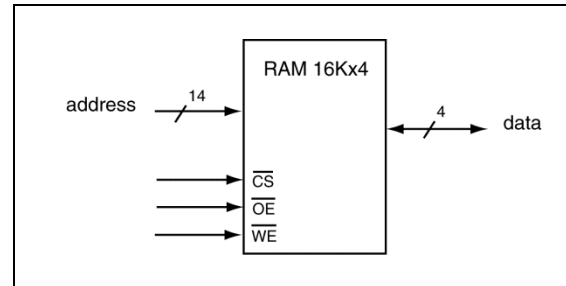
- 12) Show the address ranges in both binary and hexadecimal associated with the use of two 256x8 memories to form one 512x8 memory.

- 13) Show the address ranges in both binary and hexadecimal associated with the use of two 4Kx8 memories to form one 8Kx8 memory.
- 14) Show the address ranges in both binary and hexadecimal associated with the use of four 4Kx8 memories to form one 16Kx8 memory.
- 15) Show the address ranges in both binary and hexadecimal associated with the use of four 128Kx8 memories to form one 512Kx8 memory.
- 16) Show the address ranges in both binary and hexadecimal associated with the use of eight 1Kx8 memories to form one 8Kx8 memory.

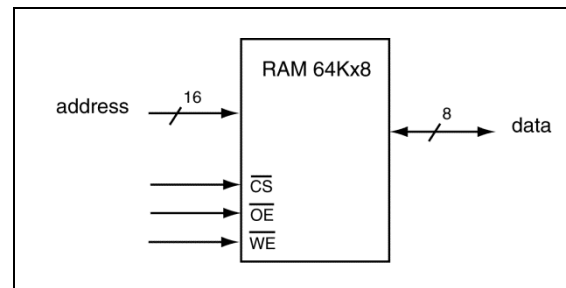
- 17) Show a circuit diagram that uses the following listed RAM to effectively create one memory that is 256x24. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



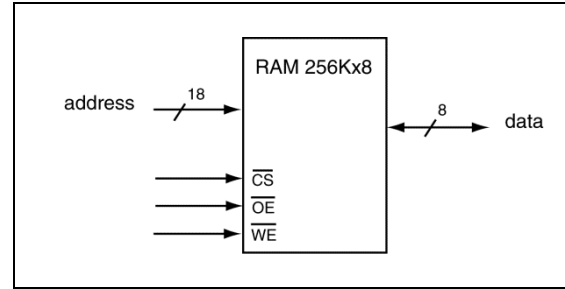
- 18) Show a circuit diagram that the following RAM to effectively create one memory that is 16Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



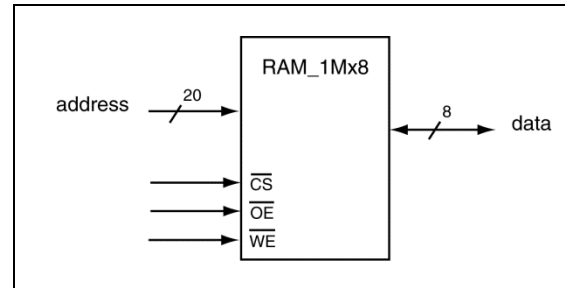
- 19) Show a circuit diagram that the following RAM to effectively create one memory that is 64Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



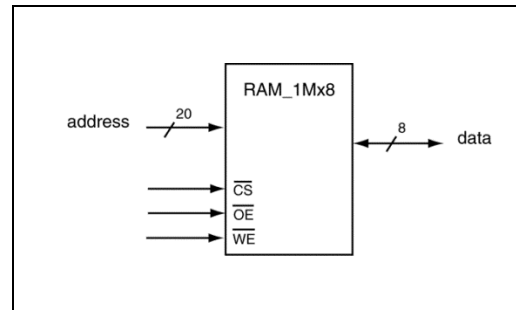
- 20) Show a circuit diagram that the following RAM to effectively create one memory that is 256Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



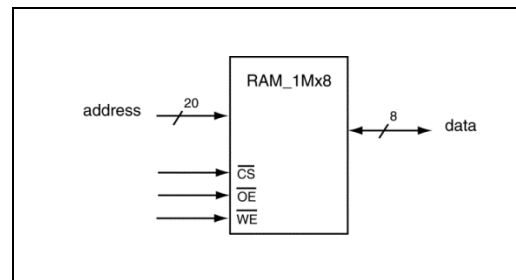
- 21) Show a circuit diagram that the following RAM to effectively create one memory that is 1Mx32. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



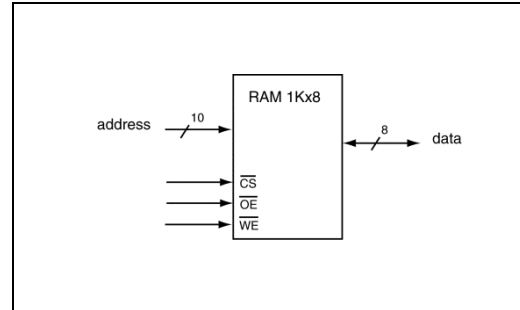
- 22) Use as many of the following RAMs as you need to create a memory with a 2Mx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



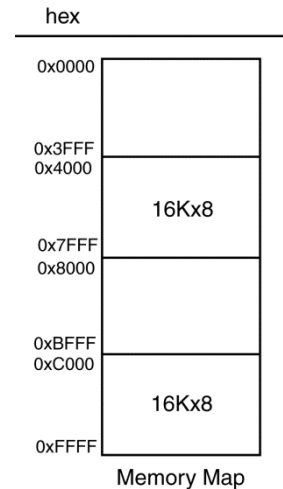
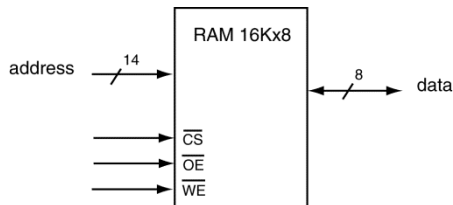
- 23) Use as many of the following RAMs as you need to create a memory with a 4Mx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



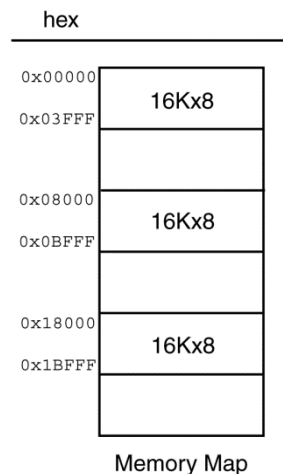
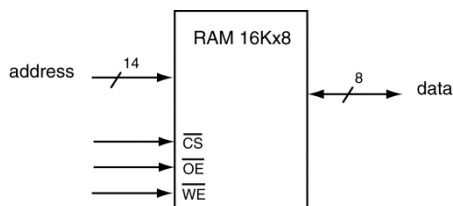
- 24) Use as many of the following RAMs as you need to create a memory with an 8Kx8 capacity. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



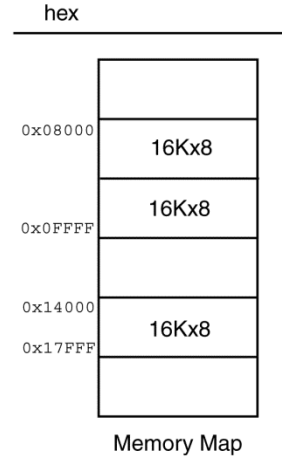
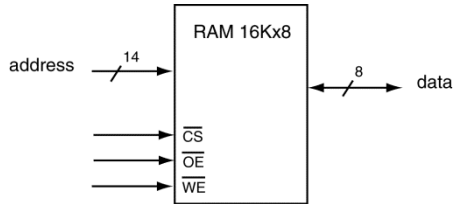
- 25) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only 2:4 standard decoder, and 2) using only one 1:2 standard decoder. Use two of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



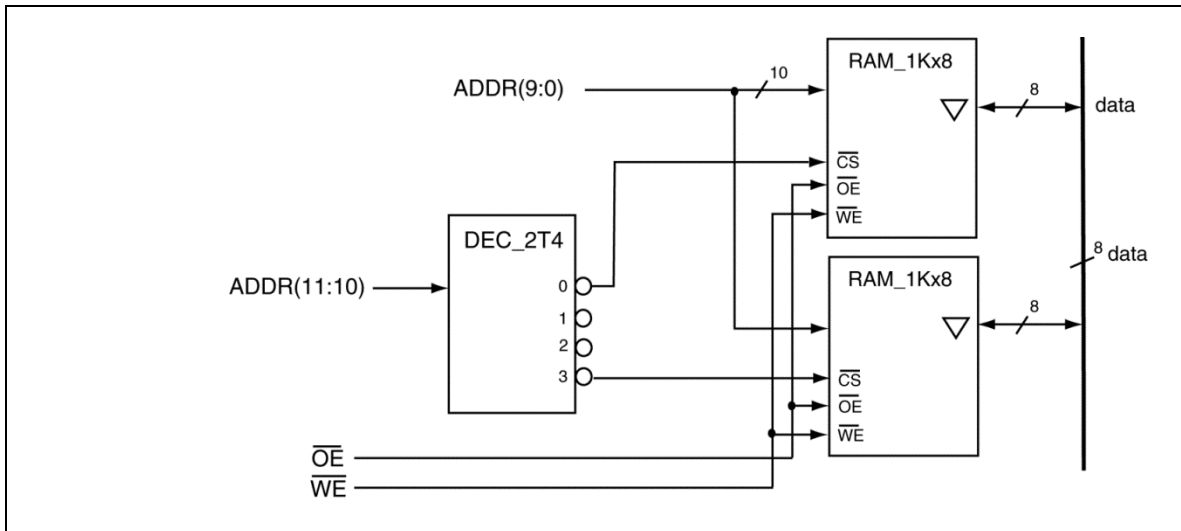
- 26) Design a circuit that implements the following memory map. Use only a 3:8 standard decoder. Use three of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



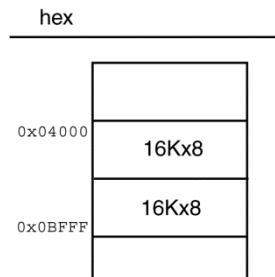
- 27) Design a circuit that implements the following memory map. Show the solution Using only one 3:8 standard decoder. Use three of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

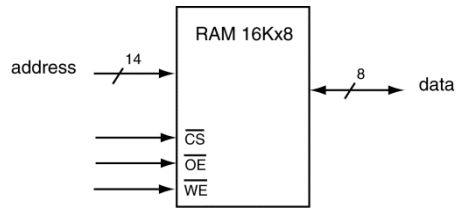


- 28) Complete the memory map below that describes the following design. Make sure to provide the starting and ending addresses (hex or binary) for used each memory device. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

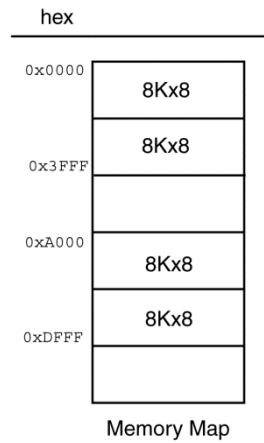
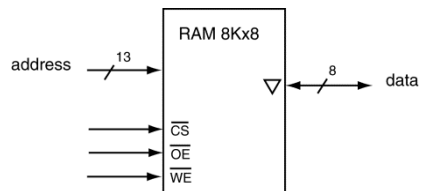


- 29) Design a circuit that implements the following memory map. Show a solution using only one 3:8 standard decoder. Use four of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

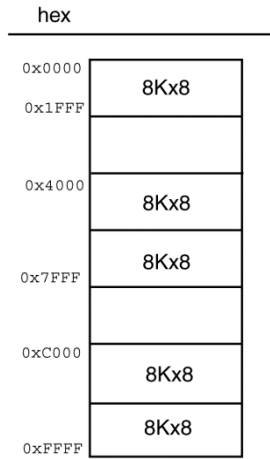
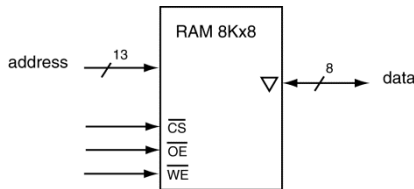




- 30) Design a circuit that implements the following memory map. Show a solution using only one 3:8 standard decoder. Use four of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

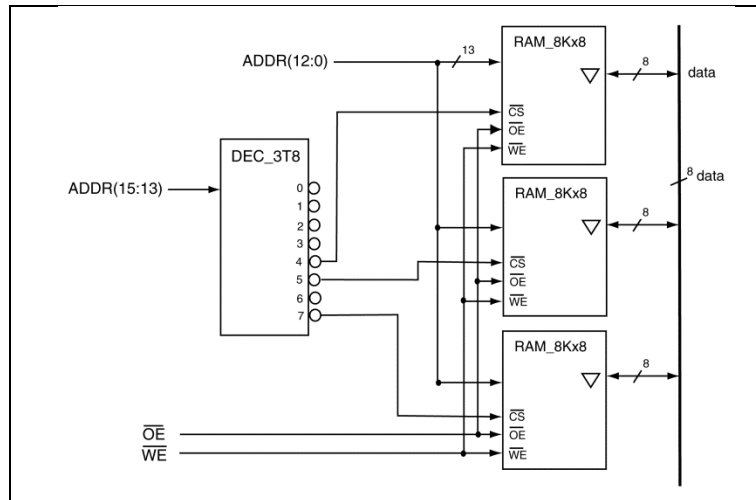


- 31) Design a circuit that implements the following memory map. Show a solution using only one 3:8 standard decoder. Use five of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

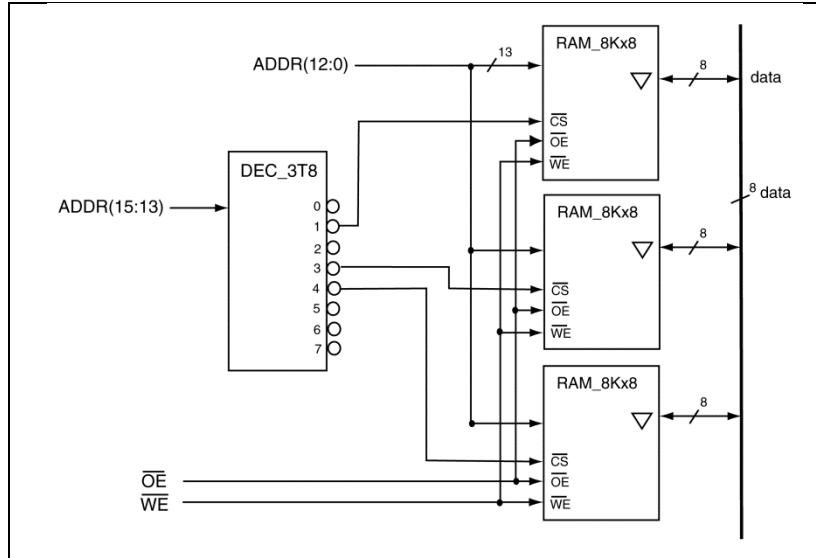


Memory Map

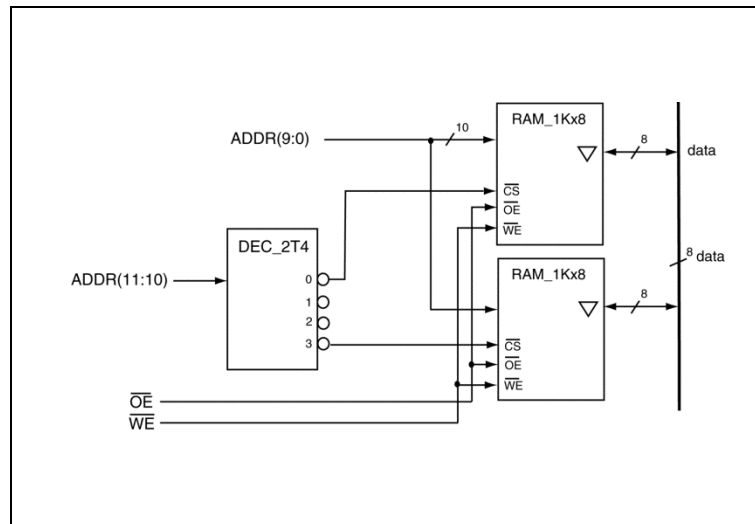
- 32) Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



- 33) Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



- 34) Complete the memory map below that describes the following design. Make sure to provide the starting and ending addresses (hex or binary) for used each memory device. Assume the CS input is an active-low chip select that “turns off” the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



5.13 Chapter Design Problems

For the following problems:

- Provide a top-level BDD and as many lower-level BDDs as necessary to describe your solution
 - Minimize the number of states in the associated state diagrams
 - Minimize the use of hardware when problem require extra hardware
 - Assume all inputs and outputs are positive logic unless stated otherwise
 - Explicitly state whether state diagrams have Mealy or Moore outputs where appropriate
 - Disregard all setup and hold-time issues
 - For sequence detector problems assume the X input is stable when each clock edge arrives and that X can change no more than once per clock period.
 - State all forms of control for your solution.
- 1) Design a circuit that upon the pressing of a button, determines how many values in a 16 RAM are negative, and displays that value until another button press. The RAM contains 8-bit signed numbers in RC format.
 - 2) Design a circuit that upon the pressing of a button, finds the maximum value in a 16x8 RAM, and displays that value until another maximum value is found after another button press. The RAM contains 8-bit unsigned numbers.
 - 3) Design a circuit that upon the pressing of a button, finds the minimum value in a 16x8 RAM, and displays that value until another minimum value is found after another button press. The RAM contains 8-bit unsigned numbers.
 - 4) Design a circuit that upon the pressing of a button, determines how many values in a 16x8 RAM are evenly divisible by eight, and displays that value a button press restarts the process. The RAM contains 8-bit unsigned numbers.
 - 5) Design a circuit that upon the pressing of a button, determines how many values in a 16x8 RAM have a value of 15 or less, and displays that value until a button press restarts the process. The RAM contains 8-bit unsigned numbers. Don't use a comparator in this problem.
 - 6) Design a circuit that upon the pressing of a button, sums all the values in a 16x8 RAM and displays that value until a button press restarts the process. The RAM contains 8-bit unsigned numbers.
 - 7) Design a circuit that upon the pressing of a button, determines if the value in a 16x8 RAM are in ascending order. If they are in ascending order, the circuit turns on an LED; otherwise it leaves the LED unlit. The circuit does this each time a button is pressed. The RAM contains 8-bit unsigned numbers.
 - 8) Design a circuit that upon the pressing of a button, determines how many bits are set in a in a 16x8 RAM and displays that number on the output. The circuit does this each time a button is pressed.
 - 9) Design a circuit that reads all the values in a 16x8 RAM. If the value is less than 26, the circuit changes that value to 0x00. The circuit does this each time the button is pressed. The RAM holds unsigned binary values.
 - 10) Design a circuit that upon the pressing of a button, determines how many values value in a 16x8 RAM are even parity and how many values are odd parity. The circuit does this each time a button is pressed.
 - 11) Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a "GO" signal, the circuit counts the number of values in each even address location in a 16x8 RAM that are evenly divisible by 8 and stores that count in a register.
 - 12) Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a "GO" signal, the circuit finds the minimum value in a 16x8 RAM. Upon completion, the circuit continually outputs both the minimum value and the RAM address of that value until another GO signal is detected. The RAM contains unsigned 8-bit values.

- 13) Provide the hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit counts how many values in each even address location in a 16x8 RAM are evenly divisible by 8. Consider address “0000” to be an even address location.
 - 14) Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit stores the largest value in a 16x8 RAM into an 8-bit register.
 - 15) Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit sums the values in each memory location of a 16x8 RAM if they are less than 63 and stores the result in a register. The final result should not be changed until another GO signal is detected. The RAM contains unsigned 8-bit values.
 - 16) Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit counts number of values in each memory location of a 64x8 RAM that are less than 32 and stores that count in a register. The final result should not be changed until another GO signal is detected. The RAM contains unsigned 8-bit values.
 - 17) Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit sums the values in two 8x8 RAMs and outputs that sum until it receives another GO signal. Design your circuit for either minimum operating time or minimum hardware; state which approach you are taking. The RAM contains unsigned 8-bit values.
 - 18) Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit finds the maximum value in a 16x8 RAM, and then clears that value in RAM. Upon completion of this operation, the circuit waits for another GO signal. The RAM contains unsigned 8-bit values.
-

PART THREE: Introduction to Computers

6 The Basic Computer in High-Level Terms

6.1 Introduction

The purpose of this chapter is to describe the notion of “computers” at a high level using terms associated with computer programming and basic digital design. We’re assuming you have experience with both basic digital design concepts and basic computer programming concepts¹. This is an important chapter as it gives you a meaningful roadmap for the stuff you’ll be learning from this text and the associated laboratories. As you may or may not know, FreeRange Digital Design Foundation Modeling is my book on digital design; it is available for free download from www.unconditionalllearning.com.

Main Chapter Topics

- **HIGH-LEVEL OVERVIEW OF COMPUTER ARCHITECTURE:** This chapter provides a high-level overview of computer architecture, which provides a context for the information in this text.
- **COMPUTER PROGRAMMING CONTEXT:** This chapter provides a context for the act of programming computers in terms of hardware, software, and the human destine to interact with them.
- **LEVELS OF PROGRAMMING:** This chapter describes the various levels possible for programming computer.
- **COMPUTER PROGRAMMING CONTEXT:** This chapter provides a context for the act of programming computers in terms of hardware, software, and the human destine to interact with them.

Why This Chapter is Important

This chapter is important because it provides a high-level overview of the computer design by placing computer design into a familiar context.

6.2 High-Level View of Learning “Digital Stuff”

Why are we doing this? Why did you bother learning about basic digital design and basic computer programming concepts? I hope that the answer is not because you want to get a job making the big bucks². The good answer is that you have a strong desire to have some external device help you solve problems. Herein lays the major difference between your first course in digital design and this text.

6.2.1 Solving Problems with Digital Circuits

Whether you know it or not, the thing you did in your first digital course was to learn how to design digital circuits that could solve problems. There are many ways to solve problems, designing digital circuits to solve

¹ FreeRange Digital Design Foundation Modeling is a viable text describing a relatively high-level approach to learning digital design. You can find this text and a complete set of learning materials at the following website: unconditionalllearning.com.

² It may actually be the case, but don’t admit it to anyone.

them is just one of those ways. Note that the advantage of designing digital circuits to solve problems was that the digital circuit forming your solution works really fast.

The general way you solved problems with digital circuits was that you received a problem you needed to solve, then you designed a digital circuit to solve that problem. Then you were given another problem to solve, and then you had to design another digital circuit to solve that problem. The point is that you designed a different circuit to solve each problem; we often refer to these specific circuits as *one-off* solution because they only solve one problem. While there is nothing inherently wrong with this approach, one could argue that there is a better approach, particularly as the number of problems you need to solve using digital circuits increases. This is because that hardware you designed to solve any particular problem has little chance of being able to solve other problems. In other words, your circuit was roughly speaking *single purpose*.

The good thing to note here is that your digital circuit was probably the “fastest acting” approach you could have taken, which means after you design the circuit (which may have taken awhile), the circuit operated relatively fast.

6.2.2 Solving Problems with Computers

Solving problems with digital circuits is great if you have the time to design a new circuit every time you need to solve a new problem. The truth is that you’ll not always have the time to design such circuit, especially as the solutions become more complex. The solution is to design a relatively generic digital circuit that you can use to solve many different problems without having to redesign the circuit each time you have a new problem. The solution is thus to design a digital circuit called a computer. Roughly speaking, the general construction of that big digital circuit (the computer) does not change. What does change for this approach is the “program” you write for that computer. In rough terms, the program provides a means of controlling the other part of the digital circuit in such a way as to solve the given problem. In this way, you don’t have to continually change the digital circuit to solve problems, you now only need to change the program.

6.2.3 Final Problem Solving Overview

You now know two ways to use digital circuits to solve problems. Either you can design a new circuit with each new problem (inherently a hardware solution) or you can design a generic digital circuit (a computer) and write a new program to solve a given problem (inherently a software solution). Each approach has its pros and cons. But wait, it gets better. If you have the good fortune of being tasked to solve a problem using a digital circuit, there is nothing stopping you from using a computer-type circuit with supporting circuitry. I believe they call this *co-design*; it’s definitely an art in itself. Think about it, if you can offload some of your processing tasks to external hardware, you’ll be able to use a more “simple” computer (which most often means less hardware, slower clock speed, etc.). It’s a long story.

6.3 What is a Computer?

In truth, if you ask a 100 people what a computer is, you’ll most likely receive 100 different replies. The working definition for a computer that we’ll go with for now is this: *a computer is a device that sequentially executes a stored program*. This so-called “device” is generally some special set of hardware that someone has configured to interact in a useful way with the stored program. The underlying factor is here is that as a result of executing a program, we’ll end up with some useful result. The hardware in the computer is generally not changeable, but we change the “result” by changing the program. We consider the program to be software; this software executes on the computer’s hardware. This definition of a computer works for now; we’ll be adding to it in later chapters.

The only thing that a computer can provide us with is data: 1’s and 0’s; it’s up to the user to interpret this data in such a way as to make the 1’s and 0’s into actual information. The real purpose of a computer is to process the data according to the directions contained in a stored program and return something useful in the form of bits. In the end, a computer may be nothing more than a device that twiddles bits, which allows us to model a computer with the standard block diagram in Figure 6.1.

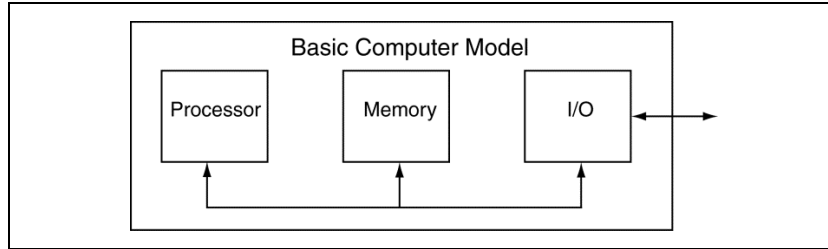


Figure 6.1: General model of a computer.

The three blocks in Figure 6.1 deserve some explanation, as these are common terms in the world of computer design. We’ll delve more into these later, but for now, here’s a short overview.

- The **processor** is a generic term for a module that inputs data, does something to it (such as processes it), and delivers the result somewhere. The processor is the “brains” of the computer, which means its main responsibility is to crunch data as required by the program stored in memory and being executed by the computer.
- The **memory** is one of the few words in computer design that is not an acronym. The memory holds data that allows the computer to operate properly. In short, the memory module, at the very least, stores the program the computer is executing. In reality, there are many other pieces of “memory” in a computer; we’ll get to those later.
- The **I/O** is short for **Input/Output**. For any computer to be useful, it needs to communicate with the outside world. In a generic sort of way, the computer receives input data from the outside world (input such as a keyboard press or sensor data) and then delivers some result back to the outside world (output such as display device or audio device).

Outside of the modules in Figure 6.1, the other important item is the directed arrows. In this overly simplified drawing, the arrows indicate that the processor connects to the memory and the I/O, which indicates that it is exchanging data with these devices. Likewise, the memory connects to the processor and the I/O. Note that only the I/O connects to the outside world.

Figure 6.1 lists a computer model that provides an opportunity to present one of the most commonly used words in the world of computer design. Namely, Figure 6.1 provides a description of a computer *architecture*. The word architecture is the commonly accepted method of describing the hardware of a computer at a relatively high level. More specifically, the computer architecture depicts the arrangement and interconnection of a computer’s functional blocks. Figure 6.1 shows the architecture at quite high level but it does actually provide much useful information.

One of the problems with the use of the words “computer architecture” is that it is not specific to any one level of describing a computer. This is the same notion as a “model” of something: there can be many ways to model something; we base the appropriateness of any model on how well it delivers the information we’re interested in. This is why when you hear someone use this term, you can never be sure exactly what level of description they are referring to.

6.4 You and the Computer

Assuming that a given computer has already been designed, you’re either a computer user or a computer programmer or both (and both at the same time). The most basic interaction with a computer is for you to “use” the computer. This roughly means that you’re interacting with a physical device that some computer is controlling.

Figure 6.2 shows a model of this simple interaction using a cheesy diagram. In Figure 6.2, “You: the user” is interacting with the computer. This means that somehow you are providing the computer with data. This data may come from typing on a keyboard or some type of sensor data such as a heart monitor. We model this interaction in Figure 6.2 with an arrow directed away from “You: the user” and going into the box label “Computer”. Note that the Figure 6.2 model of a computer only shows an interior box label “memory”, which

emphasizes the notion that we consider the computer be “running” because it is executing a set of instructions (a program) stored in its memory.

For the computer to be actually useful, it must return data to you. This data could take on any forms such as a visual display, a blinking LED, a buzzing, etc. The computer generates the data it provides you with and outputs that data by the running program; the program most likely under constant influence by your input as “You: the user”. Yes, a simple model indeed. You embody this model about a bajillion times each day, but who’s counting?

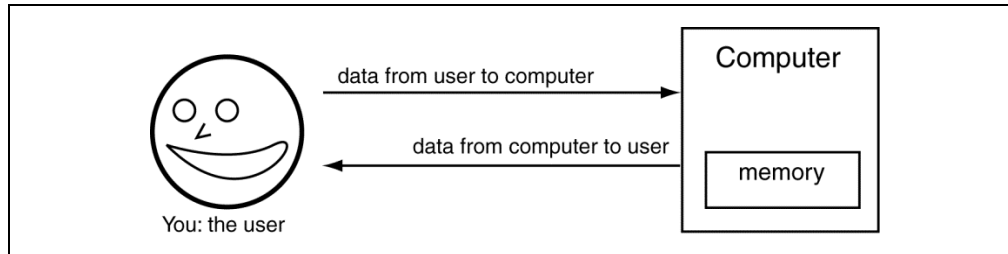


Figure 6.2: A basic model of an eerie human-like face interacting with a computer.

You don’t always have to be a simple user of a computer; you can also write computer programs. **Figure 6.3** shows a diagram that models you as the programmer (labeled “You: the programmer”). There are several steps for you to program the computer.

- The first step is that you have to write a “computer program”. The notion here is that you use some type of “computer language” and some form of software (such as a text editor) to write your computer program. The “computer language” is generally some sort of text that is syntactically structured so that the next step in the process can understand it.
- The second step translates the computer program to something that the computer (particularly the underlying hardware) can understand and use. You generally write the program in a language you can understand, and then you input it into another piece of software that translates the instructions in the computer language you’re using into a stream of 1’s and 0’s, which computer’s underlying hardware can understand. Recall that the computer itself is a digital device and thus can only understand 1’s and 0’s; software such as a compiler or an assembler translate your programs to 1’s and 0’s.
- The third step is get the 1’s and 0’s that make up your program into the memory of the computer and start the program running. We’re going to leave this step outside this conversation due to the notion that there are many ways to do this and we want to keep speaking in generalities.

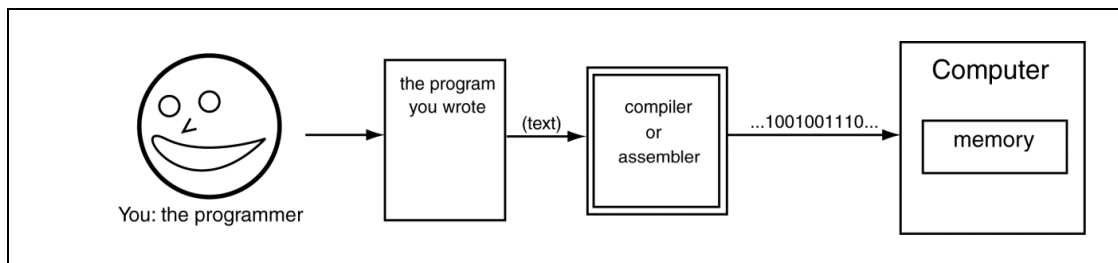


Figure 6.3: A basic model of an eerie human-like face writing a program to execute on a computer.

6.5 Computer Architecture: For the Hardware People

We’ve agreed that a computer is a piece of hardware that executes a stored program. We went over some high-level details regarding the operation of a computer in a previous section. In this section, we’ll leverage your current knowledge of digital design, which allows us to delve deeper in to the basic computer model of Figure 6.1. Figure 6.4 expands on the computer model of Figure 6.1 by listing the useful sub-modules for some of the computer’s basic blocks. The goal of this new computer architecture is to provide you with more high-level

insight into how a basic computer operates. We'll do this by describing the basic block in Figure 6.4 in more detail.

The I/O Module: This module did not change from the previous computer model. One interesting item to note is that the model in Figure 6.4 has the I/O module only connected to the processor. This is arbitrary; different architectures would have different interconnects but this model attempts to keep things generic.

The Memory module: A typical digital circuit can contain many types of memory ranging from flip-flops to large structured memory devices (such as RAM & ROM). The memory module in Figure 6.4 contains two memories: 1) instruction memory, and 2) "data" memory. As we spoke of earlier, the instruction memory stores the program that the computer executes. The data memory stores "data" that the computer requires to obtain required results. We use the term "data" memory to mean many things, all of which are outside of the context of this discussion. In short, computers generally store data in various places as a means to obtaining the required result.

The Processor module: We divide the processor module into two separate sub-modules: the *CPU* and the *Control Unit*.

- The *Control Unit* is responsible for reading an instruction and sending out the appropriate control signals to the other hardware modules in the computer that are responsible for executing that instruction. Note that the arrow points from instruction memory to the Control Unit for the instruction and from the Control Unit to the CPU with control signals. The Control Unit is typically a finite state machine (FSM), no different from the ones you studied in an introductory digital design course. The Control Unit is responsible for making sure the right things happen at the right time (and in the correct sequence) to implement the computer's instructions.
- The acronym *CPU* stands for Central Processing Unit. The CPU "processes" data under control of the Control Unit, which is in turn following orders from the instructions in instruction memory. As with many FSMs, the Control Unit receives status of various operations from the CPU as Figure 6.4 indicates with an arrow directed from the CPU to the Control Unit. One of the main sub-modules of the CPU is the ALU, which stands for Arithmetic Logic Unit. The notion of the ALU is somewhat antiquated in that a typical ALU does more than simple arithmetic and logic instructions. The term CPU is also antiquated; in days gone by, hardware was expensive and there was typically only one piece of hardware that did all the number crunching/bit manipulation and had a "central" location in the hardware. Processing in modern computers typically happens in different places, not in one place.

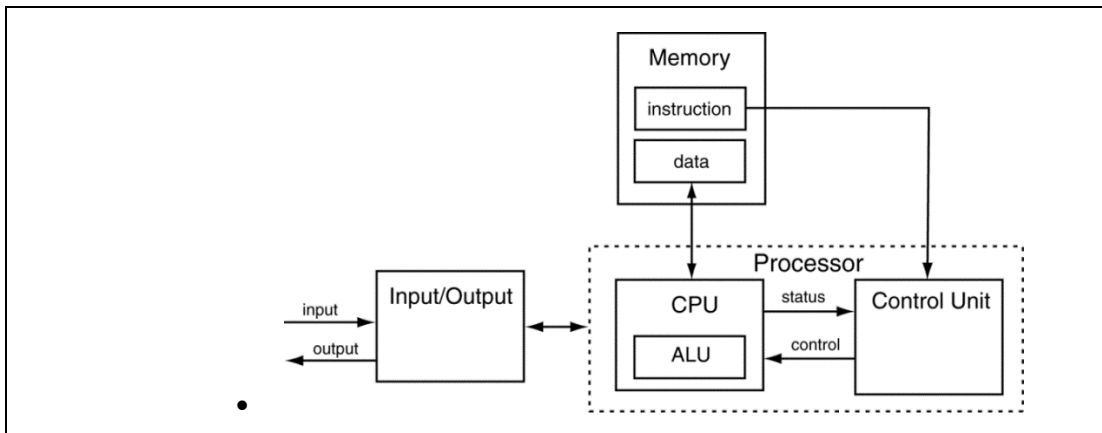


Figure 6.4: A more detailed computer basic computer architecture.

6.6 Computer Architecture: For the Programmer People

You're either a user of a computer or a programmer of a computer (you can be both at the same time). If you're a computer programmer, you'll need to understand the computer architecture you're programming as well as the

tools you have for programming and ensuring that program works properly. These two items fall under the notion of the “programmers model” (sometimes referred to as the programming model) and the “instruction set”. This section describes these two items in terms that you should be familiar with from your previous digital design and computer programming experience.

If the field you end up going into has something to do with computer design or low-level computer programming, you’ll always need to learn about new computers, namely, 1) their basic construction, and 2) how to program them. When learning about a new computer, the two items you initially look at are the programmer’s model and the instruction set. We go in-depth into these topics later, but for now, we only provide a brief overview.

6.6.1 Programmer’s Model

The Programmer’s Model is a high-level view of the hardware resources that the programmer can utilize using the various computer instructions that make up their programs. Note that while a computer is comprised of a relatively significant amount of hardware, the programmer cannot control all of that hardware. Also, note the programmer controls the hardware by writing “instructions” that the computer will execute; the instruction set lists the instructions available to the programmer.

6.6.2 Instruction Set

While the programmer’s model shows the resources available to the programmer, the instruction set allows the programmer to use those resources. In other words, the instruction set is what the programmer uses to create an actual program. Every different computer has a different instruction set because the underlying hardware is different and there is a different set of instructions that control that hardware.

6.6.3 Computer Instructions

So what exactly is an instruction? As with everything else in a computer, it is nothing more than a set of bits. These bits act as control signals that implement or allow certain data processing operations to occur in a computer. Although it is possible for us humans to write strings of 1’s and 0’s and use them to represent directives (instructions) to the hardware that makes up the computer, it’s not the most efficient approach to programming a computer. Computers are complex monsters; we must constantly do what we can to make them easier to understand, design, and eventually use; writing programs using ones and zeros does not mitigate the overall complexity of controlling the computer.

6.7 Programming Language Levels

If you’re reading this sentence, you’ve probably programmed a computer. If you’ve programmed a computer, you certainly must have some notion of the low-level details of what you were actually doing as you programmed that computer. In case you did not know what you were doing, this section aims to give you a quick overview of the big picture regarding the programming of computers.

Once again, the bit patterns that are associated with the instructions control the operation of a computer. You can write a computer program at one of three different “levels”; these levels are 1) “machine code”, 2) “assembly code”, 3) and some “higher-level language”. This section describes these levels including the information of **Figure 6.5**.

6.7.1 Machine Code

We refer to programs in the form of 1’s and 0’s as *machine code* or *machine language*; it’s the lowest level of programming. A program written in machine code is nothing more than a set of 1’s and 0’s arranged in bit-patterns that direct the operations that the underlying architecture should perform. The good part about writing programs using machine code is that there is no need to use other software (not including a text editor) as a precursor to writing a program. The downside of this approach is that programs are nearly impossible to write and completely unreadable. There probably was a day when all programmers had to use machine code to write all their programs, but that was back when dinosaurs were biting each other while they were programming their dinosaur computers. Although every program that anyone ever writes eventually ends up as machine code, the programs rarely start that way anymore.

6.7.2 Assembly Language

The next level up in the programming hierarchy from machine code is assembly language programming. In an assembly language, we replace the bit-patterns that form the instructions by mnemonics, which generally describes in shorthand notation the operation the computer should perform. Each of these mnemonics (and some other associated information) is associated with some specific set of 1's and 0's. People sometimes refer to assembly language as a "symbolic machine code", which advertises the fact that assembly language is still low level. The set of mnemonics for a given computer is generally what we consider the *instruction set* for that computer.

The upside of using assembly language programming over machine code is that mnemonics bring a level of understandability to the code as opposed to attempting to use your human brain to interpret endless strings of 1's and 0's. The downside, (if it we consider it one) is that you need another piece of software referred to as an "*assembler*" to translate the assembly language instructions to machine code. The downside of assembly language programming is that every different computer architecture (the computer hardware) necessarily has a different assembly language. Although writing code in different assembly languages is not that complicated once you know one assembly language, it does, however, have a slight learning curve.

In the end, it's still all about using the assembly language instructions to crunch bits. The reality is that the flavors of bit-twiddling are limited (in other words, you can only do so many things with bits). This means that once you know one assembly language (and have a grasp of generic programming concepts), you can relatively quickly and easily switch to another by simply learning the syntax and instruction set of the new assembly language. For example, every instruction set has an instruction that rotates a value in a register: in one assembly language, the accompanying mnemonic may be ROR and in another language, the same function would be RR. Same function, different mnemonic.

6.7.3 Higher Level Languages

The next step beyond assembly language programming is to use some type of higher-level language (HLL). Because each assembly language instruction generally performs only a basic operation, assembly language programs can quickly become long (many lines of assembly instructions) when the program is implementing a relatively complex set of operations. One possible solution to producing long programs is switching to coding the programs using a HLL. When you use a HLL, each line of code in the HLL can represent many lines of assembly code, which leads to shorter and arguably more understandable programs. When you use a HLL, you must use a *compiler* to translate the HLL code into machine code. Most likely, the compiler first converts the HLL code to assembly code before the final translation to machine code.

Using a HLL has one distinct advantage over assembly code: once you know one HLL, you can write code for any architecture without know anything about the underlying assembly language, assuming you have the correct compiler. This effectively lessens the learning curve for switching processors and generally makes your HLL code independent of the computer architecture your programming. One major downside of HLLs is that the code is not necessarily as efficient as it would be if a human generated the assembly code.

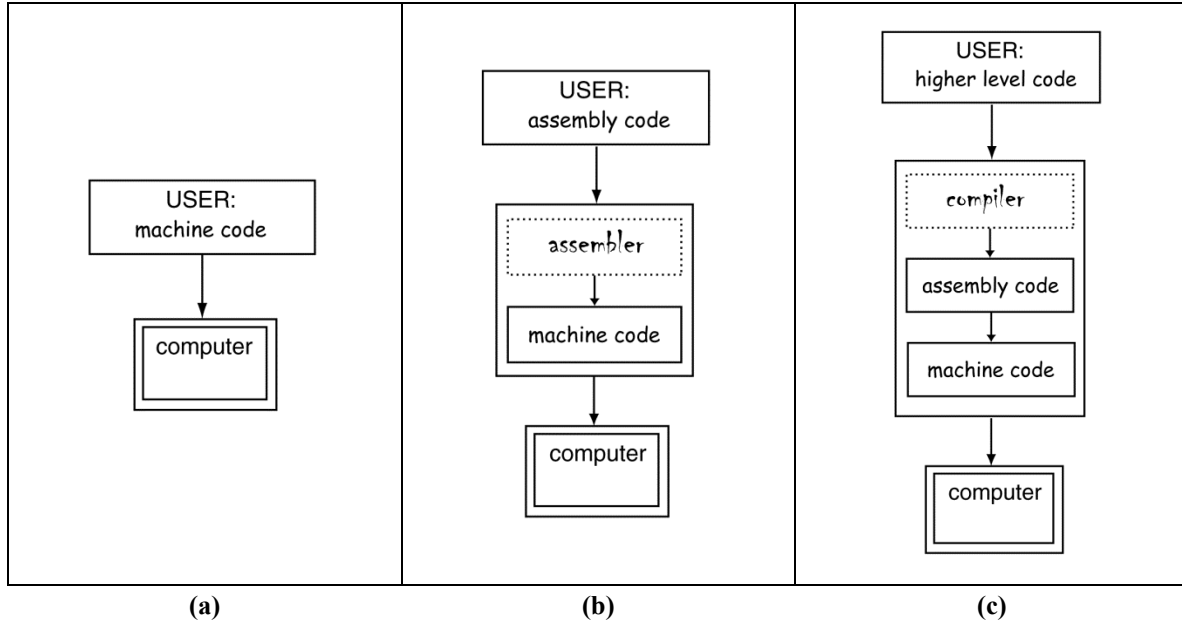


Figure 6.5: The three different levels in which you can program a computer.

6.8 The Digital Design Hierarchy

This text is about moving towards designing a computer. You’ve come a long way down the digital path to get to this point and here is a reminder of some of the more important milestones along the way. It all started with your first digital design course. Computer design represents what the next step in the natural progression of your “digital education”. Here is a brief reminder of the progression:

- The typical digital design course starts with number systems including a strong emphasis on binary number systems and various methods to represent information in binary form (binary coded decimal, 2’s complement, signed and unsigned numbers, etc.). Although this was not digital design, we’ll be using many of these concepts directly because we have a sincere interest in the ways computers store and interpret bit patterns.
- Next came the basics of digital design: AND, OR, NOT functions and gates. This quickly got into the design of basic combinatorial circuits with way too much emphasis on reducing Boolean equations before circuit implementation. The circuits that we implemented at this point were generally pointless but they provided an enjoyable academic exercise.
- Next, we placed the basic gates in certain configurations in order to obtain certain functionality. This allowed us to abstract our designs to a higher level in order to avoid talking about working with low-level things such as gates as much as possible. The resulting devices were more complex than gates but the complexity was manageable because you understood the basic functionality of the circuit from a high level. These more complex devices included such things as MUXes, decoders, adders, comparators, etc. You may have forgotten how these devices are structured, but you hopefully remember how the devices operate. For example, the mention of the word “MUX” brings to my mind a form of data selection. It probably doesn’t bring to mind a circuit any more complex than a block box.
- The concept of basic memory arrived with the introduction of sequential circuits. Our main use of sequential circuits was with registers and their various forms such as counters and shift registers. We used these memory devices to construct finite state machines (FSMs). The FSM has one primary function: it’s a circuit that controls other circuits. We later added the notion of structured memory, which was our definition for memory models designed specifically to store large amounts of data.

This progression represents part of the big picture: these were all tools we can use to design and understand a basic computer. Note that many times along this progression, we constructed circuits out of small boxes, placed the small boxes into another box, and gave it a new name. This embodies the general approach of computer design and the understanding of complex digital circuits of any type: the *hierarchical* approach. We've applied this approach from early on in your beginning digital design course in that we studied gates rather than the underlying transistors that implemented them. We extended this approach with slightly more complicated circuits which were a special assembly of gates (decoders, MUXes, etc.). And in the end, it was this hierarchical approach that allowed us to understand complex circuits by abstracting upwards. For example, the concept of a 4:16 standard decoder with a chip enable is easy to comprehend while the transistor-level circuit that implements this functionality would fill a page and would not be at all pleasurable to look at.

This hierarchical concept becomes even more important as we move into computer design. We consider a computer to be nothing more than a very complex state machine. The problem is that even the simplest computer has so many possible states that the techniques we've used to design and analyze state machines are essentially worthless if we try to apply them directly to computer design. Because of this, we necessarily need to take a different approach to designing sequential circuits such as computers, namely, the hierarchical approach. In particular, it's a top-down approach to the design of computers, which entails describing at a high-level the functional blocks in a computer.

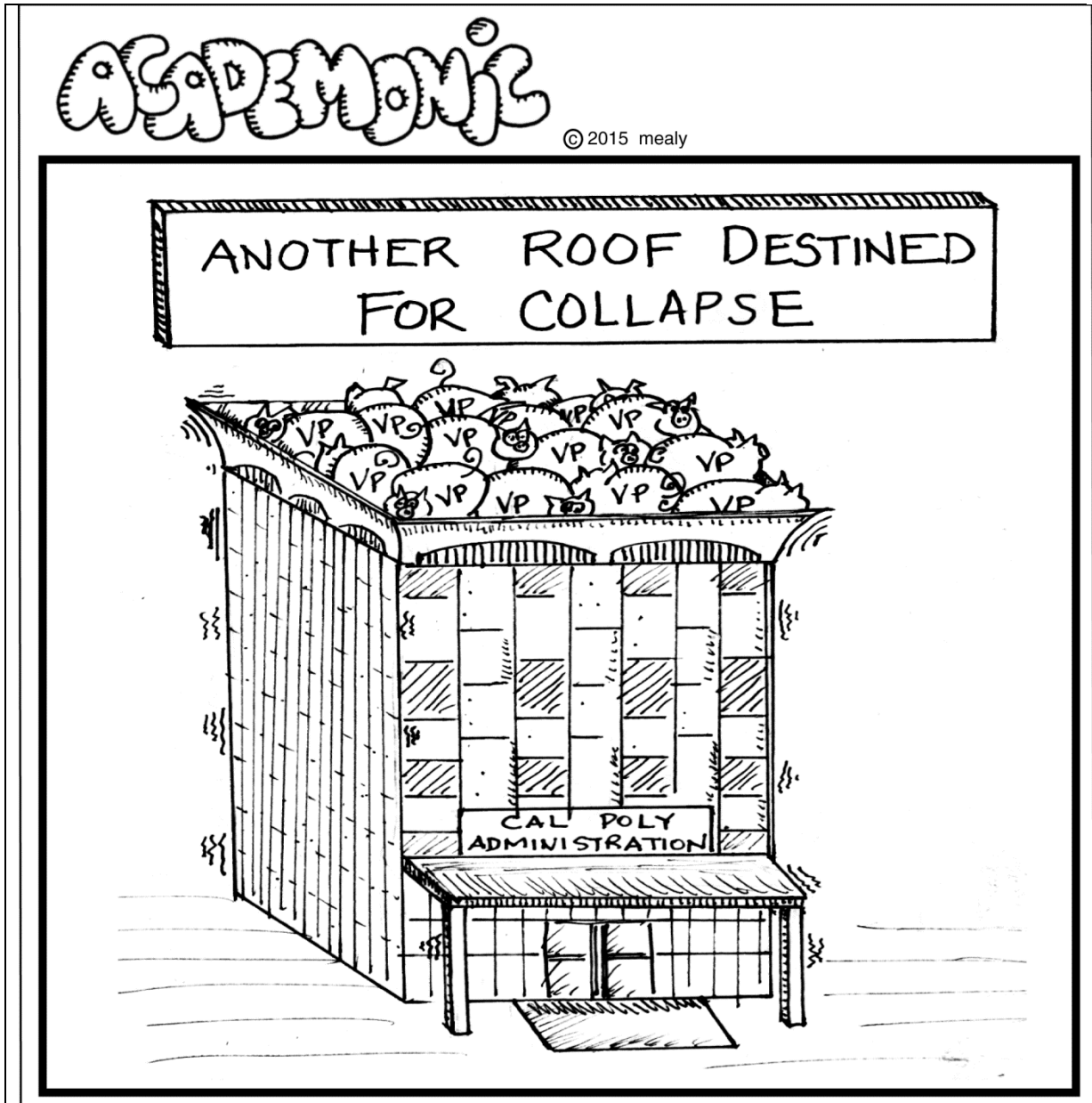
6.9 Chapter Summary

- A computer is a device that sequentially executes a stored program. Note that this is one of many definitions for a computer; this definition is quite high-level.
 - A computer is comprised of three main subsections: the memory, the input/output, and the processor. The processor crunches data based on instructions stored in memory; the input/output allows the computer to interact with the outside world.
 - You as a human interact with computer as either a user or a programmer, or both. You the programmer write programs using a text editor; some other piece of software translates your program into machine code or machine language, which is the only language a computer can actually understand.
 - The programming side of a computer is defined at a relatively high level using the Programmer's Model and the Instruction Set. The Programmer's Model shows the resources that the programmer can control using the computer's Instruction Set.
 - Programmers can program computers at three different levels: 1) machine code, 2) assembly code, and 3) a higher-level code. The application you're working on and your immediate supervisor generally dictate what level you'll be programming at.
 - The notion of "digital design" could mean many things. The idea is that you can perform digital design at one of many different levels. As you progress towards more complicated digital designs, and particularly computer designs, you'll be designing at higher levels of abstraction. Though it would be possible to design an entire computer at the transistor level, no one actually does it as design computer at the transistor level because designing at higher levels of abstraction is much more feasible and cost effective.
-

6.10 Chapter Exercises

- 1) In your own words, define the word “computer”.
 - 2) Briefly describe the main purpose of any digital circuit, including a computer.
 - 3) List the pros and cons of using a computer to solve problems as opposed to designing a dedicated digital circuit.
 - 4) List and briefly describe the function of the three main modules of a computer.
 - 5) Briefly describe what is meant by the term “computer architecture”.
 - 6) Briefly describe why there is no one absolutely correct model for any digital circuit.
 - 7) Briefly describe the main use of a computer architecture.
 - 8) Briefly describe what is meant by the term “computer instruction”.
 - 9) Briefly state the purpose and relationship between the programmers model and the instruction set.
 - 10) The programmers model does not show all of the hardware associated with a computer, only a subset of the associated computer hardware. Briefly describe why some of the computer hardware is not included in the programmers model.
 - 11) Briefly describe why there are so many different assembly languages out there.
 - 12) Briefly describe why computer instructions are represented using mnemonics.
 - 13) Briefly describe what makes it relatively easy to learn a new assembly language once you know one of them.
 - 14) Briefly describe why programming using machine code is nearly impossible.
 - 15) Briefly describe the distinct advantage that using a higher-level language has over using an assembly language.
 - 16) I claim to have designed a portable assembly language; briefly state why you would be skeptical of such a statement.
 - 17) Briefly describe a possible advantage that programming using assembly language has over programming using a higher-level language.
 - 18) Briefly describe the three levels of programming.
 - 19) Briefly describe what is meant by the notion of a “digital design hierarchy”.
 - 20) Briefly describe why it is that you can use a high-level module without understanding the low-level implementation details.
-

PART FOUR: RISC-V Assembly Language Programming



7 Assembly Language Introduction

7.1 Introduction

Assembly language programming is overwhelming once you first see it. First, it is programming, but programming that is different from languages such as C and Python. Second, you have to learn the “instruction set” and a bunch of assembly language “tricks” to be able to successfully and efficiently program in assembly language. Third, you probably need to become familiar with many hardware concepts regarding the computer associated with the assembly language you’re setting out to learn¹. There is a lot to learn, but most all of the stuff you need to learn is relatively simple (once you see what is going on).

The problem with teaching assembly language programming is that there is no good place to start. It seems everything you need to know is based on something else you need to know, but if you’re just starting out, you don’t know anything. This chapter chooses to start somewhere; the stuff you learn in this chapter helps you learn the more detailed stuff in later chapters. This chapter also reviews a standard structured approach to designing assembly language programming. As the programs you write become more complex, it becomes important for you to take a healthy and sane approach to designing and writing programs.

Main Chapter Topics

- **BEGINNERS VIEW OF ASSEMBLY LANGUAGE:** This chapter gives a generic overview of assembly languages in a context that just about anyone can understand.
- **PROGRAMMING LANGUAGE LEVELS:** This chapter put assembly language programming into a proper context of the different levels of possible for “programming a computer”.
- **ASSEMBLY LANGUAGE: GOOD OR BAD:** This chapter describes some the good and bad points of using programming at the assembly language level.
- **AN APPROACH TO WRITING ASSEMBLY LANGUAGE PROGRAMS:** This chapter provides an outline of the appropriate approach to writing assembly language programs.
- **FLOWCHARTS:** This chapter provides motivational verbage that highlights the advantages of using flowcharts and describes the basic symbols associated with flowcharting.

Why This Chapter is Important

This chapter is important because it introduces assembly languages and associated concepts as well as basic program structure concepts.

7.2 Bits to Mnemonics and Back Again

We generally model a computer as a device that sequentially executes a set of stored instructions. We use the individual instructions to control the various subsystems in the computer in such a way as to produce a meaningful result. In the end, we view computers operations as simply the pushing around of bits (1's and 0's).

¹ You actually don’t need to be familiar with the computer hardware, but it will help you create all around great assembly language programs.

Computer instructions are nothing more than bits that instruct the computer to perform predefined operations, which control the bit pushing.

We refer to the computer instructions at the bit-level as a *machine language* or *machine code*. As you could imagine, dealing with an endless stream of bits is overwhelming for the average human brain. The solution is to replace the machine language with assembly language. An assembly language is a simple upward translation of the machine language where we represent the bit patterns that form the instructions by *mnemonics*. We design the assembly mnemonics in such a way as to convey the purpose of the instruction as it relates to the function that it causes the computer hardware to perform. The upside of this translation from bits to mnemonics is that the purpose of an instruction is much easier to envision and understand for humans. The downside the bits-to-mnemonic translation is that the translation needs to be undone in order for the computer to execute the instructions. A software program known as an *assembler* translates the assembly code to machine code.

Controlling a specific computer architecture in such a way as to do something useful requires a specific machine language, and hence, an accompanying *assembly language*, for that architecture. This means there are as many different assembly languages out there, as there are different sets of computer hardware, or *computer architectures*. Computers generally differ by the number and type of operations, the “size” of data they work with, and the way and number of ways they store the data. From a high level, computers are generally able to carry out essentially the same functionality, but they must do so within the limits of their underlying computer architecture. The programmer exercises the basic functionality of a computer by using the assembly language associated with a particular computer and the assembler associated with that assembly language.

7.3 Programming Language Levels

The bit patterns that make up the instructions are what controls computer: computers understand nothing other than bits. Although it is possible to write programs using the bit-patterns directly, this approach is too tedious to make is useful and there are approaches that are more “useful” as well. The methods used to program computers are generally broken into three general levels: 1) machine code, 2) assembly code, 3) and higher-level languages. This section describes these levels; additionally, Figure 7.1 shows a graphic of these levels.

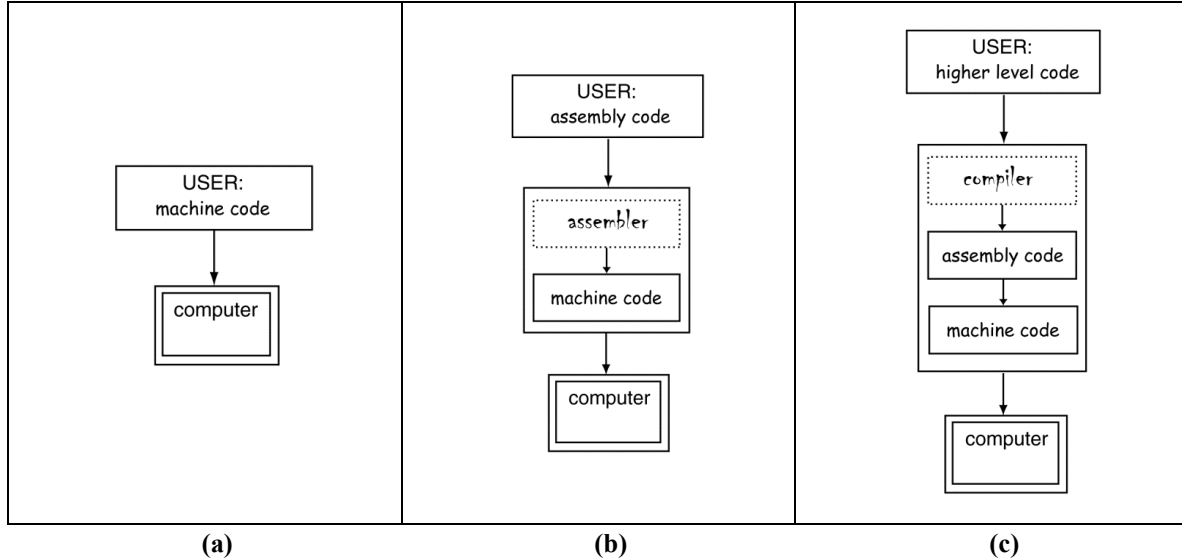


Figure 7.1: The visual choice to programming from a user's perspective.

7.3.1 Machine Code

Machine code is the lowest level of programming, meaning the level closest to the actual hardware. A program written in machine code is nothing more than a set of 1's and 0's, which we arrange in bit-patterns that control the operations that the underlying architecture can perform. The good part about writing programs using machine code is that there is no need to use other software (not including a text editor) as a precursor to writing a program. The downside of this approach is that programs are completely unreadable, as the look like a mind-

boggling stream of 1's and 0's. There probably was a day when all programs had to be written in machine code, but that was sometime in the prehistoric computer era when the earth was ruled by computersaures. Although every program that is ever written eventually ends up as machine code², programs rarely start that way.

7.3.2 Assembly Language

The next level up in the programming hierarchy from machine code is assembly language programming. In an assembly language, we replace the bit-patterns that form the instructions by mnemonics that loosely indicate the operation the instructions perform in the underlying computer hardware. The upside of using assembly language programming over machine code is that mnemonics bring a level of understandability to humans reading the code. The downside, (if you can consider this one) is that you need another piece of software referred to as an *assembler* to translate the assembly language instructions into machine code. The assembler is rarely an overly complicated piece of software based on the notion that assembly languages are generally highly constrain in their structure compared to higher-level languages. The downside of assembly language programming is that every different computer architecture (the computer hardware) necessarily has a different assembly language. Although writing code in different assembly languages is not that complicated once you know one assembly language, any new assembly language has a learning curve, with a steepness that depends on the overall complexity of the assembly language³.

7.3.3 Higher Level Languages

The next step upwards beyond assembly language programming is to use some type of higher-level language (HLL). Because each assembly language instruction generally performs only a basic operation, assembly language programs can quickly become long (many lines of assembly instructions) when the program requires a relatively complex set of operations. One possible solution to producing long programs is switching to coding the programs using an HLL.

When you use a HLL, each line of code in the HLL can represent many lines of assembly code, which leads to shorter and arguably programs that are more understandable to humans. When you use a HLL, you must use a *compiler* to translate the HLL code into machine code. Most likely, the software first converts the HLL code to assembly code before the final translation to machine code. Using a HLL has one distinct advantage over assembly code: once you know one HLL, you can write code for any computer architecture without know anything about the underlying assembly language assuming you have the correct compiler. This effectively flattens the learning curve for switching processors and makes you HLL code architecturally independent of the underlying hardware. The official technical term for this is HLLs are *portable* while assembly languages are not portable. The only downside of HLL is that the code is not necessarily as efficient as it would be if a human generated the assembly code. Compilers are good, but humans who know what they're doing (meaning they understand the assembly language and underlying hardware) are better.

7.4 Assembly Languages: The Goodness of “Low-Level”

Through the years, assembly languages have received some rather bad press. Most people who have worked with assembly languages find that assembly language programming can be tedious, primarily because the programs tend to be “long” when they are actually performing a useful task. The length of assembly programs appears to be long because they generally only have one “simple” instruction per line of text in the source code.

On the other hand, working at a low-level has several distinct advantages over using a higher-level language. Moreover, programming using a higher-level language without knowledge of the computer architecture that you intend to execute the code on can be outright inefficient in some cases. Here are some of the many benefits of programming in assembly:

- Assembly language programming inherently provides an overview of the underlying computer architecture. Therefore, writing programs using assembly language are essentially a simultaneous lesson in computer programming and computer architecture.

² More specifically, if someone executes the program on a computer.

³ The notion here is that an assembly language instruction can be very simple or very complicated. Simple instructions, such as basic bit tweaking, are not a big deal. But, more complicated hardware can be designed to do many things with a single instruction. In this case, you're going to have to spend more time reading the manual.

- Assembly language programming requires that the programmer have source code organizational techniques in order to produce viable (readable, understandable, maintainable) source code. The programmer can control the potential “length” of assembly language programs by using modular programming techniques. Learning and applying these techniques helps improve the quality of source code you write at any level.
- Assembly language programming can ensure certain portions of the code operate efficiently. Even if you are primarily writing in a higher-level language, there may be portions of the code that can be “coded by hand” to make sure the machine code is as efficient as possible⁴.
- There is a common argument that modern compilers are as efficient as a human (an intelligent one) programming in assembly language. I don’t believe this as it sounds more like a marketing ploy for a compiler company than it does true science. If you’ve ever dealt with optimizing the code generating step in a writing a compiler, you’ll understand the inherent limitations in the process. Without doubt, there are some efficient compilers out there, but it is highly unlikely that they all fall into the “good as a human” category.
- Assembly language programming helps the programmer develop a true appreciation for the higher-level languages. The more complicated the task, the more you’ll want to move to a higher-level language for bulk of your programming needs. But then again, maybe not, as you should never put 100% trust in a compiler.
- Assembly language programming builds character. Yes, recent research has proven this true in practically every known case.

There are well over 5000 different microcontrollers out there in the real world, which implies there are about the same number of different assembly languages. This number does not include the various proprietary microcontrollers and other projects that were never released to the public. The question that should be asking yourself now is: “With so many assembly languages out there, what are my odds of ever using the one we’re about to learn?” This is a good question. The answer is that you’ll probably never see the any given assembly language that you work with ever again. But here’s the truth: working with an assembly language for the first time can be challenging, but the skills and knowledge you gain in the process easily transfers to other assembly languages and in general makes you a better programmer.

What makes all assembly languages similar is that they all do the same thing: they manipulate bits. The only difference between any two assembly languages is exactly how the underlying hardware manipulates the bits and how the bits are stored, which are characteristics governed by both the instructions available to the programmer and the underlying hardware. To come up to speed quickly when learning a new assembly language, you simply need to understand the basic programming resources, which allows you to use them effectively. The quickest way to do this is by perusing the *Programmers Model* and the *Instruction Set*:

- **Programming Model**: The programming model is the programmer’s view of the computer: it shows what hardware resources are available for the programmer to use. These resources primarily include registers and other types of memory. Another useful definition for the Programming Model is the set of registers that the instructions in the associated instruction set can manipulate.
- **Instruction Set**: The instruction set lists the set of operations that the hardware can perform under control of the programmer.

One interesting point here is that there is no mention of the actual architecture of the device. There is also no mention of the external interface of the device. These are interesting points because they highlight the fact that the discussion of assembly languages generally means that we are abstracting our approach to the device to a higher level. The general thought here is that we are now going to be writing assembly language programs. We

⁴ The standard trick here is to use a “profiler” to determine the typical executional characteristics of a program. As you may find out one day, programs usually spend most of their time executing a small subset of program’s instructions. Therefore is you want to speed up your program without spending a lot of time doing so, you rewrite the higher-level code in these sections using assembly code. In this way, you get your program speed-up without having to recode your entire program.

generally assume that some other fine person implemented the device in some type of hardware setting and has setup the environment so that all we have to do is provide the working source code.

7.5 Problem Solving with Programming

In the rush to complete your assignment for your instructor or supervisor, you can easily lose track of what it is you're attempting to accomplish. In the worst cases, you find yourself mired with either the low-level details while ignoring the big picture, or you completely grasp the big picture while being unaware of the important low-level details you're probably passing over. In the end, if you want to be a successful programmer, you need to answer "yes" to the following questions:

- 1) **Did you write your program in a reasonable amount of time?** If you answer "no", you need to realize that you can't spend forever writing the program... at some point you have to call it done.
- 2) **Does your program work properly in all possible cases?** Of course if you answer "yes" to this question, it means that you've course tested the poop out of your program
- 3) **Can someone else easily understand and/or reuse your code?** If you answer "no" to this questions, then you're either a bad programmer or you're into creating job security for yourself.

The key to ensuring that you answer "yes" to all of these questions is to keep your programs as simple as possible. Why? Because complex programs, if they work at all, are well known to be fragile. A fragile program is like an academic administrator's ego: you constantly worry about breaking it if you accidentally do anything wrong. Yes, the program running on your smart phone is a result of millions of lines of code and is seemingly complex, but that's not the point. If you can decompose even the most complex program into simple building blocks, then the program is by definition simple⁵. The key to writing good programs is writing simple, well-structured code (see section 7.6).

After you've been programming in assembly for a while, you'll find that you've probably developed your own coding style and your own approach to the entire "problem solving" package. Recall that the reason you're writing any program is to solve some problem. When you first start out programming, particularly using an assembly language, you should take a nicely disciplined approach. This section describes a high-level approach to the entire problem solving process, not just the program writing part of the approach. Though this is certainly not the only approach you can take, it's the approach you should take until you have developed your own successful assembly language programming style.

There are three basic requirements you must meet before attempting to solve problems by writing assembly language programs: 1) Understand the instruction set, and, 2) understand basic programming constructs and techniques, and 3) understand the underlying hardware architecture. These two items are somewhat detailed and we provide more information in the following verbage.

- 1) **Understand the Instruction Set:** What this means exactly is that you must understand every aspect of the instruction set if you plan to write viable problems. This is totally possible because there generally a relative few number of instructions for a basic architecture and most of the instructions are relatively simple⁶. Some of the aspects you need to be familiar with are:
 - a. **Peculiar aspects of individual instruction:** Different instructions have different ways of doing things. The push for the efficiency in the underlying hardware can make some instructions very confusing and hard to understand. For example, how the hardware forms memory addresses in instructions that reference memory.
 - b. **Forms of instructions:** We generally divide the various instructions in an instruction set into a smaller subset of instructions that share the same instruction formats; each of these formats manipulates the underlying hardware in the same manner.
 - c. **How instructions use registers:** Different instructions use different number of registers and use those registers differently.

⁵ So when your phone freezes or does something stupid... probably bad code. Please don't blame the hardware.

⁶ This statement is even more true with RISC architectures.

- 2) **Understand the Underlying Hardware Architecture:** You of course must intimately familiar with the Programmers Model, but that often is not enough to write good programs. You must also understand the how the various modules in the architecture do the things they need to do to implement the instructions in the instruction set. We're trying to be good programmers here, but that can only happen if we have a basic understanding of digital the digital circuitry that forms the computer.
 - a. Space limitations on various memory elements: Computers have a finite number of memory elements, with emphasis on finite; hardware designers typically cut corners in order to save space (reduce overall hardware) and power consumption. The savvy programmer needs to understand the limitations in order to effectively solve the problem at hand.
 - b. Digital tricks: The hardware is capable of doing many things, though some of them are not obvious. Two example of this would be to use shifting for divides/multiples by two, and, the various tricks to manipulate bits: set, clear, toggle, and hold.
 - c. Input/Output architecture: Computers can handle input/output operations in a few common but different ways. Programmers need to various I/O architectures to write efficient programs.
 - d. Interrupt architecture: Computers generally interface with the outside world according to the programs they are running. If you need to interface with the computer, there are ways to have the computer stop what it's doing and deal with your requests. All computers do it, but they do it in different ways; you need to be familiar with those ways.

- 3) **Understand Basic Programming Techniques and Constructs:** You must understand basic approaches to programming in order to write code. One of the many good things about assembly language programming is that there are only a few constructs you need to know. Even though there are only a few constructs you need to know, the most complicated assembly language program (a well-written one, that is) is a conglomeration of these constructs. The basic items we're referring to are:
 - a. iterative constructs (loops): The two types of iterative constructs are loops when you know in advance how many times you'll iterate (based on a count) and when you don't know how many times you'll iterate (based on a condition). Either of type of iterative constructs can be further classified as a "while loop" or a "do-while" loop.
 - b. if/else constructs: The if/else construct is the basic decision-making program flow construct in assembly language programming.
 - c. bit manipulation and bit masking: Computers "handle" bits in only four different ways, which programmers must be well-versed with. Additionally, bit masking is one of the basic techniques to operate on bits, which is generally what you'll find yourself doing on a microcontroller⁷.

- 4) **Understand the "Toolchain":** There will be several "tools" you need to be familiar in order to run your code on actual hardware. These tools are essentially the various software packages that allow your solution go from an idea to a working computer that solves the problem. Here are a few of those items (assuming we're writing assembly language programs and having them run on a programmable logic device, or PLD):
 - a. The text editor: Knowing the features in text editors helps you write programs in an efficient manner.
 - b. The Assembler: This software translates your assembly language program code to a set of 1's and 0's that your hardware understands.

⁷ Recall that microcontrollers are generally designed to control other pieces of hardware. This means they must read individual "status" inputs and respond by sending out "control" outputs to the items the MCU is controlling.

- c. The PLD Computer Aided Design Tools: These tools allow you to model and synthesize your computer hardware such that you can execute the programs you wrote.

Once you've met the basic requirements, you're then ready to solve the problem by writing an assembly language program. The three basic steps to writing assembly language programs are:

- 1) **Understand the Problem**: This is an important step because if you don't understand the problem, there is no way you'll generate a viable solution⁸. The general idea here is that you'll get a high-level picture of the problem, which starts your brain thinking a path to the solution, which is of course the next item.
- 2) **Generate a Path to the Solution**: The notion of generating a path to the solution involves writing designing an algorithm that will solve the problem using the given parameters. In reality, there is no way you can solve the entire program with one giant plop: your brain does not work that way, and computer programming (as of this writing) does not work that way either. You'll be designing an algorithm, and there are two standard approaches to algorithm design:
 - a. Pseudo Coding: Pseudo code is an unstructured semi-written-language approach to describing a path the solution. We'll not cover this approach in this text, but it truly is helpful and something that all programmers should know.
 - b. Flowcharts: The flowchart provides a visual description of the basic flow of your program. Flowcharting describes program flow by using a few basic shapes. We dedicate the bulk of this chapter to flowcharting, so we'll opt not to say much here.
- 3) **Translate the Path to Assembly Language Code**: Once you've mapped out your algorithm, you must then translate the algorithm into the actual assembly language instructions that will implement that algorithm on a given processor. For this step, you'll need to meet the two requirements of solving problems using assembly language.

A few comments regarding all these new rules and things:

- You can't step item #1; you need to understand the problem before you can solve it.
- You can skip step #2, but you shouldn't. In addition, if your program is anything other than simple, you won't be skipping step #2 if you plan to actually generate a viable solution to the problem.
- You can't do step #3 if you don't have a solid understanding of the instruction set.

7.6 Structured Programming

The official definition of a simple program is one that we can decompose into simple parts. A consequence of this definition is that if we can't decompose our programs into simple parts, it's a complex program. There are many concerning issues with complex programs including the fact that they suck. Let's face it; if you're not a disciplined programmer, you're going to be writing spaghetti code, and you're going to hate life as much as your boss or instructor hates you. Here are some more specific issues regarding complex programs:

- They have a lower probability of working in all cases, or working at all
- They're hard to understand, maintain, and modify
- It is hard or impossible to reuse part of the programs

The approach you should take to programming is to write "structured code". The basis of writing structured code is to realize that you can categorize any code you can possibly write into one of three "structures": 1) the sequence construct, 2) the if-then-else construct, and, 3) the iterative construct. In other words, your code is

⁸ You'll generate a bunch of code, but it will generally be worthless. You may get lucky, but engineers don't rely on luck; only administrators rely on luck (and a wild show of waving hands) to solve problems.

either 1) doing something or going somewhere else to do something, or, 2) doing something conditionally, or, 3) doing something repeatedly. These constructs become easier to understand after you see them modeled with a flowchart.

If you're truly writing structured programs, your code is going to be a series of these three constructs. In other words, you should be able to decompose your program into a set of these three structures. Disclaimer: just because you're writing structured code does not guarantee that the program is going to work properly, as there are other issues regarding computer programming that you need to contend with. The payoff is that structured code is essentially the most cost effective approach to creating and maintaining a working program. Even if your program does not initially operate as expected, structured code helps you expedite debugging and testing your program. Structured programming has the added benefit of helping new programmers learn to work with the instruction set and develop their own great programming style.

7.7 Motivational Discussion of Flowcharting

You can view the writing of any useful software (or firmware) as a solution to some problem. In other words, any worthwhile program that was ever written was done in order to do something useful. We can characterize this usefulness as providing a meaningful result; we can further characterize the solution as being an *algorithm*.

An algorithm is a computational or logical method of producing a desired result. Flowcharts are useful because they facilitate the development and visual representation of algorithms. The flowchart is the software analog to the black block diagrams (BBD) we use to describe hardware subsystems. Recall that hardware block diagrams are able to quickly convey an understanding of the circuit at hand. You'll find that flowcharting an algorithm serves the same purpose: flowcharts quickly convey the basic operation of an algorithm. Another way to look at it is that BBDs model hardware while flowcharts model algorithms; programmers can then use the flowcharts as a guide to generating their programs. Keep in mind that both flowcharts and block diagrams worked well with hierarchical design to further promote understanding of the items they model.

A flowchart has two basic purposes. It is the best idea to consider it a design tool, which is how we'll be emphasizing it here. But being that flowcharts present a graphical representation of the order in which operations are carried out by programs, we can also consider them a great documentation tool that provides another description of your program in addition to the code (well commented code) itself. The use of flowcharts as a documentation tool is a by-product of proper program design. The flowchart is a great aid for anyone who needs to design a program; when the program is complete, the flowchart automatically becomes a great documentation item for anyone who later needs to understand your program. In the end, the flowchart is great design tool and documentation tool.

We can judge any piece of program code by the following qualities (with lots of overlap among these qualities): modularity, reusability, understandability, readability, modifiability, and extensibility. If you can write code that contains all of these qualities, you win the big prize of having reliable code. In the real world, you'll mostly likely be working on a team of people who all in one way or another is contributing to the production of a given product that is running some program. As you can imagine, it's a big piece of software since there are so many people are working on it. In this case, if even one small part of the code does not contain all of the above qualities, the code will spawn many problems that have a strange tendency of never going away. Problems that don't go away will create a lifetime of problems for anyone and everyone who has any dealings whatsoever with the project. The result is an unmaintainable, unmanageable, and worst of all, unreliable piece of software crap that people will continuously marvel at the fact that it ever works at all. And most likely the moment it fails will be at a customer demo.

Flowcharting supports all the qualities of good source code. So if the discussion above has not convinced you that you should use flowcharts in your program design and subsequent documentation, just do it anyway. Someday you'll thank yourself for building a sound foundation of solid programming practices.

The overall purpose of flowcharts is to quickly present information regarding a process or algorithm (particularly one code using a programming language). In addition to this goal, here are a few other fun items about flowcharts to keep in mind:

- There are many options so as how to generate flowcharts, we'll stick with the basic symbols. You can add the bells and whistles later as you see fit.

- There is no “right” method to do flowcharts. In that they are tools to help you design and/or document your work, you’ll need to provide your own definition of “right”. A good place to start, however, is with the basic concepts presented here.
- If your flowchart meets the overall goal stated above, you have a good flowchart. Part of this definition of “right” should be the level of detail that your flowcharts provide. You may need to have several flowcharts for one section of code where each of the flowcharts would present data at a different level. Flowcharts do quite well presenting various levels of detail.
- Don’t hesitate to present “flowcharts within flowcharts” because as you’ll see, they nest quite nicely. The only rule you should follow is that any single flowchart should contain about the same level of detail (note the ambiguity of the word “about”). If you need to change that level of detail, you should start a new flowchart.
- Consider keeping flowcharts as generic as possible. For example, describe an algorithm using generic programming operations that could be used for any hardware. Once your flowcharts start calling out specific items such as loop iteration counts and hardware specific items such as registers, the flowchart becomes less usable when and if the hardware changes. Additionally, if you make the flowchart generic, it can remain unchanged with minor changes in the algorithm.

7.7.1 The Basics of Flowcharting

Table 7.1 shows the a few basic symbols that we typically use in flowcharts. When you see flowcharts in various places, you’ll be seeing other symbols also, but these other symbols represent bells and whistles. As far as structured programming goes, the symbols in Table 7.1 represent the basic functionality of sequential programs, so the discussion in this chapter sticks with those symbols. We’ll start continue this discussion by looking at the flowcharts as they relate to some of the basic programs we’ve written so far. We’ll look at a few examples of flowcharts supporting basic programming constructs. We’ revisit them when we have some actual assembly language coding experience in a later chapter.



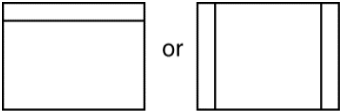
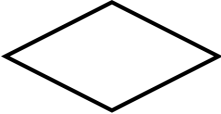

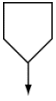
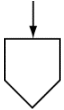
Symbol	Description
	<u>Flow lines and flow arrows:</u> the directed line segment indicates a sequence that the program follows. These lines guide the reader through the other flowcharting symbols in the correct order.
	<u>Process:</u> The rectangle symbol indicates that the algorithm performs the operation or process listed in the rectangle. All process symbols have only one exit flow line but can have multiple entry points.
	<u>Predefined Process:</u> These are a special type of process symbols that we generally use to specify a process that is predefined (such as a subroutine) or defined in some other location.
	<u>Decision:</u> The algorithm determines program flow by the condition specified inside the diamond. The decision symbol has only two exit flow lines, which are either <i>yes</i> , or <i>no</i> . Decision boxes have multiple entry points.
	<u>Terminal:</u> specifies the beginning or end of a program or subroutine.
	<u>Off-Page Connection, Entry:</u> This symbol indicates that a given flow line continues on another page. We generally fill these symbols with a short label such as “A” that matches the off-page exit connection.
	<u>Off-Page Connection, Exit:</u> This symbol indicates that a given flow line continues on another page. We generally fill these symbols with short labels that match the off-page entry connection. .

Table 7.1: The basic symbols used in flowcharting.

7.8 Structured Programming Revisited

We gave a motivation blurb regarding structured programming in an earlier section; we now need to fill in a few of the details. Recall that the notion of structured programming is that we can decompose any well-written program in to a conglomeration of three basic structures: 1) the sequence structure, 2) the if-then-else structure, and 3) the iterative structure. As you’ll see, we use two or more of the basic flowcharting main symbols to model each of these structures: the process box, the decision box, and associated flow lines. Not surprisingly, flowcharts are probably the best way to define/understand these three basic structures.

7.8.1 The *sequence* Structure

The sequence structure is a set of two or MORE process boxes placed in a series and considered as a new “higher-level” process box. **Figure 7.2(a)** shows the basic model of a sequence structure using standard flowcharting symbols. The notion of a sequence should seem familiar, as it is simply a form of abstracting to a higher level. The main characteristic of a sequence structure is that it begins at one point and ends at another, which is simply a way of stating that the sequence structure has on entry point and one exit point. If a structure has more than one entry point or more than one exit point, then it is not a sequence structure and necessarily not a part of structured programming. In this case, you can possibly model it as a sequence structure if you further

decompose the objects using standard structures. In **Figure 7.2(a)**, the solid boxes are the lower-level items and the dotted box is the higher-level item.

7.8.2 The *if-then-else* Structure

The if-then-else structure represents a decision point: the program decides to take one path or another based on some condition in the program. **Figure 7.2(b)** shows the basic model of an if-then-else structure using standard flowcharting symbols. To be a true if-then-else structure, the two paths must eventually merge after the execution of the chosen path completes. This characteristic assures that the if-then-else structure is similar to the sequence structure in that the if-then-else structure has one entry point and one exit point. Keep in mind that a variation of the if-then-else structure is the if-then structure. In this case, the structure either does something or does nothing, as compared to the if-then-else structure that does something or does something else. A specialized form of the if-then-else structure is the in-case-of structure, commonly known as a *case structure*. This is similar to the notion in higher-level languages of using if-then-else statements or case statements to implement the same functionality.

7.8.3 The *iterative* Structure

The iterative structure models a set of instructions that repeatedly performs the same process until the structure makes the decision to exit the structure. **Figure 7.2(c)** shows the basic model of an iterative structure using standard flowcharting symbols. The iterative structure is independent of the terminating condition, meaning that the terminating condition can be any condition supported by the exact form of the underlying language's flow control statement. Similar to the sequence and if-then-else construct, the iterative construct has one entry point and one exit point.

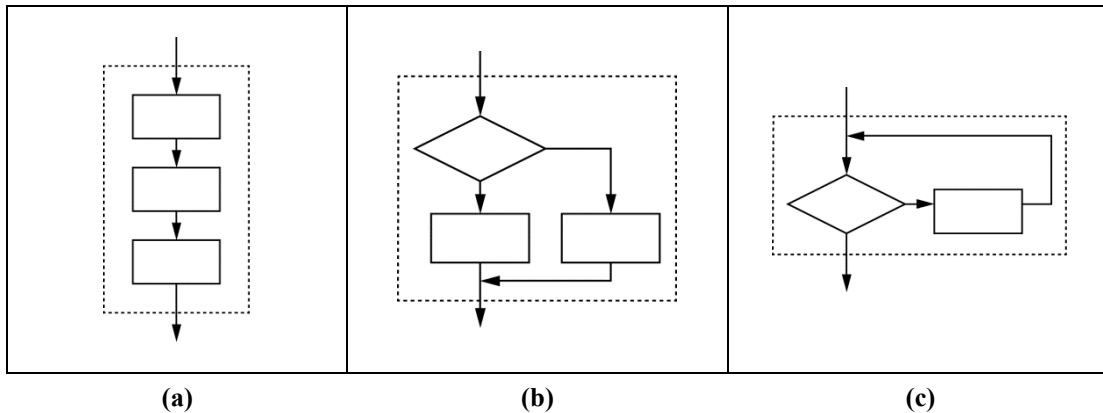


Figure 7.2: Models for (a) a sequence structure, (b) an if-then-else structure, and (c) an iterative structure.

Figure 7.3 shows flowcharts modeling the two types of iterative loops: the while loop and the do-while loop. Recall that the do-while loop always executes the associated process at least one time, which is done by executing the process before it checks the terminating condition. The while-loop checks the terminating condition before executing the associated process and thus can exit the loop before executing the process. Figure 7.3(a) shows a flowchart modeling the while loop and Figure 7.3(b) shows the flowchart modeling the do-while loop.

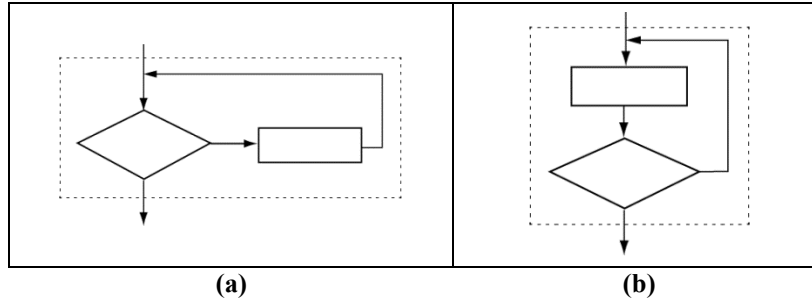


Figure 7.3: Example flowcharts modeling a while loop (a) and a do-while loop (b).

7.9 The Truth about Software

Software is definitely mysterious. Have you ever seen a program running? There's a program running the machine that's keeping your grandmother alive during her hospital stay... do you know who wrote that program? Do you know how extensively the person or people who wrote that program actually tested that program? Did the person in charge of that software think of every possible test scenario before they released the code? Should you be worried about all this stuff? I'm not sure what the answers are, but if you were worried about whether all the software that runs the world is really working correctly, you'd probably need to take lots of medication to make it through the day.

If you're reading this sentence, you're probably embarking on learning to write assembly language. Yep, it's real fun to make the LED blink or the numbers count, which sure seems trivial but is really rather important. Someday you'll graduate and find yourself on a team developing a new product. You'll be surprised how instantly that team starts depending on you to write good code for their next product; you will sort of wish you started writing good code from the get-go if you haven't already. When you see your company's product on the shelf or flying through the air, are you going to be worrying about whether your code really works or not?

7.9.1 Software Quality

Does your software work properly? How would you know if you did not extensively test it? Do you think your boss is going to ask you if you extensively tested your code? No, they will not ask you; they'll more than likely assume that you did because that is part of being a good programmer. What they're going to be asking you is if your software is completed or not. If you ask for more time to test it properly, no one will consider you a team player and you'll probably be soon laid off and then be hired as an academic administrator.

In reality, the testing and debugging of your software is most likely going to require more time than it required you to get to that point (which includes planning and writing the software). In most jobs, you'll barely have enough time to design and write the software before the release date; testing is not usually a high priority. Sad to say that the only thing that has a lower priority than testing software is actually documenting that software.

Your mission is still to write good code. Good code is going to work, and if it doesn't work, it's going to be easy to debug. Code that is easy to debug is presents a shorter path to getting the code to work. But let's be real here: all the testing in the world won't guarantee that your code will work 100% of the time. What testing will show you is that your code has bugs; testing is not going to show you that your code does not have bugs. All is not lost here; there are a few simple rules to follow to help you write good code. If you're conscientiously striving to write good code, your code will be in a constant state of improvement. If you learn from your mistakes, you won't be making those mistakes again.

7.10 Writing Good Programs

There are many great books out there describing various techniques you can use to write good programs. Because you are probably a student in an academic environment, you generally don't get a chance to experience the normal "real world" approach and accompanying expectations of writing real software. In academia, the main goal of your software is to complete the assignment at hand. In this case, you know full well that your program is probably being graded by a robot, which means most of the corners you cut attempting to submit the assignment before the due date go unnoticed by any other human.

There are several problems with writing code in an academic environment. First, courses in academia typically place way too much emphasis on completing the assignment *at any cost*. Your program does what it should in that it made the robot grader happy, but at what cost? Your code may be crappy, unreadable, unorganized, unmaintainable code—the robot doesn’t care. Because no human outside of yourself ever sees the code to inform you of your diminished code quality, you develop bad habits that you may never break. In addition, in academia, you generally have the choice of obtaining any grade outside of an F and still attain success on the assignment and pass the class. If truth, if you apply the same approach outside of academia⁹, you’d be fired rather quickly, and then later be hired as an academic administrator.

There is a right way to write code. Though you may not always have the time to take this approach, you know you should be taking this approach. We all strive to be lucky enough to have the time and/or resources to embark on writing good programs. There is much more to writing good programs than plopping down some instructions. The final word here is that writing good code is a process that extends well beyond regurgitating instructions and/or expressions; enjoy the journey.

- 1) **Know how to write proper code:** There is more to programming than simply writing code. Anyone can write good code, but it’s truly a learning process. The main problem in academia is that lazy professors don’t take the time to ensure their students are writing good code. The typical lazy professor typically verifies the code appears to be working (or has a robot check to see if it’s working) and quickly moves on to check-off the next program.
 - a) Make your code look good: Good-looking code is code that looks good standing ten feet back. In truth, most people (non-robots) who look at your code are only going to take a cursory glance it; people rarely take the time to determine if your code is actually good or not. Just like most everything else in life, people make a snap judgement based primarily on appearances: if you code looks good, it *must* be good. Therefore, if you’re not writing good code, the least you can do is make it look good.
 - b) Write structured programs: Structured programs are easier to “get working”, understand, maintain, and most importantly, debug. To be able to write structured programs, you must understand the three basic structured programming constructs: 1) sequences, 2) if-then-else, and 3) iterative constructs.
 - c) Know the entire instruction set: If you don’t know the instruction set, you’ll never be able to write good assembly language programs. If you writing in a higher-level language, knowing the underlying assembly language instruction set helps you write “better” code¹⁰.
 - d) Know the tools: There are various tools that help you write good code in an efficient manner. Assemblers and compilers have various options to help you write code that it more understandable and more portable. Simulators/debuggers often have many features that are not overly obvious to help you ensure your code is working properly.
 - e) Look for examples of good code: If you strive to write good code, you’ll become more and more aware of what good code actually is. You’ll then look at other people’s code for examples of what to do and what not to do. Learn by experience, including other people’s¹¹.

- 2) **Write simple code:** Simple code has many things going for it, though job security is not one of them. Good programmers know and understand the notion that there is a certain eloquence and beauty to good code; it’s a characteristic that defies description. If you’re trying to impress people with your code, strive to impress them with the simplicity of your code. You may not impress your butthead friends and colleagues with your code, but other good programmers will be totally impressed and adopt some of your coding practices.
 - a) Write understandable code: the assembler does not care what your program looks like, but other humans do. Understandable code is easier to get working properly, including the eventual debug part of the process. If you pass crappy code along to colleagues, they’ll quickly lose respect for you

⁹ Keep in mind this level of incompetence ensures you a promotion if you’re an academic administrator.

¹⁰ But if you’re writing in a higher-level language, often times you’re doing it for its portability characteristics, so you may not know what anything about the underlying hardware.

¹¹ As quoted to me by Keith Swanson in 1980. Thanks Keith.

programming abilities. Be sure to find an approved style file and make your code look like the code in the style file (or preferably, better).

- b) Comment your code: Use comments to primarily state “why” your code is doing something is generally more important than stating “what” your code is doing. Avoid commenting on things are obvious. Keep comments brief, but be sure to add extra comments for code that is doing something strange of patently unobvious.
 - c) Use labels in your code: Labels cost nothing but do provide a vehicle for making your code more understandable. Labels are generally short mnemonics that quickly transfer information; use labels as a special form of commenting. Don’t worry, we’ll talk about labels in an upcoming chapter.
 - d) Use white space: In fact, use a liberal amount of white space. Everything, including comments, directives, and instructions should be properly and consistently indented. Use blank lines to delineate separate ideas in the code stream. Also, use blank lines to delineate subroutines.
 - e) Write modular code: Possibly the main attribute of simple code is that it is modular. Modular code is easier to write, understand, reuse, debug, and maintain. The main vehicle for modules in assembly language programming is subroutines. Each subroutine should have a header that describes the purpose of the module, and what resources the subroutine changes.
 - f) Don’t write tricky code: Well, sometimes you have to in the name of efficiency... However, if you do write tricky code, make sure you comment the code with an excruciating amount of detail.
 - g) Write portable code: The notion of portable code means that if something in the underlying hardware changes (either the MCU or external hardware controlled by the MCU); your code will require little or no modification in order to work properly. Try not to write code that requires intimate knowledge of the hardware, or keep such knowledge to a minimum (and well commented). Use directives defined in the initial portion of your code to define constants used by the hardware.
 - h) Use look-up-tables (LUTs) when possible: You can’t say enough good things about LUTs. Always be on lookout for instances in your code where a LUT is appropriate (makes your code clearer and/or more efficient).
 - i) Write “bullet-proof code: Though it is somewhat beyond the scope of this text, write code that going to work in every possible setting, including multi-threaded environments. Don’t rely on the calling code to do the right thing; always do the right thing in each section of code you write.
 - j) Write code with testability in mind: Someone, possibly you, is going to have to debug and/or understand your code, so structure your code with testability in mind, include commented code, self-commenting labels in the code, and relatively simple code.
 - k) Write code knowing that the requirements will change: Not only will the requirements change, they will change before you’ve completed your assigned task. It’s generally fairly easy to predict such changes; you don’t need to be psychic, but it helps. They call this “feature creep”.
-

7.11 Chapter Summary

- An assembly language is a set of mnemonics that represent operations that the associated computer can perform. These mnemonics represents 1's and 0's, which are "assembled" by an assembler, which outputs machine code (the 1's and 0's). Assembly language programs are written using the instruction mnemonics.
 - We can write computer programs at three different levels 1) machine code (low-level), 2) assembly language (medium-level), or 3) a higher-level language (high-level). No intelligent person writes programs using machine code as this approach is too tedious. Assembly language programs can become long due their relative low level compared to higher-level languages. Writing programs in higher-level languages is relatively efficient as the compiler typically generates many lines of assembly code for one line of higher-level code.
 - Assembly languages are associated with specific hardware architectures. If you switch computer hardware, you necessarily need to switch assembly languages. Higher-level languages are portable in that if you switch computer hardware, you simply need to use a different compiler on the higher-level code.
 - There are many good reasons why you may want to use an assembly language over a higher-level language. Writing assembly language generally allows the knowledgeable programmer to generate code that is more efficient than a typical compiler. Assembly language programming also requires the programmer to be somewhat knowledgeable about the underlying computer architecture.
 - Assembly languages essentially tell the underlying hardware how exactly to push bits around. There are only so many things you can do with bits, so learning a new assembly language after you know one is relatively easy, as it mostly requires learning a new syntax and becoming familiar with the associated programmers model.
 - Writing programs to solve problems is an art form. However, those learning the art can get a good start by not losing sight of the problem being solved and by following this simple set of guidelines.
 - Structured programming using basic constructs assembled in a workable manner to write programs. Programs that are not properly structured often end up becoming "spaghetti code", and are essentially, giant pieces of crap.
 - Flowcharts provide a simple approach to program design and program documentation.
 - Flowcharts as a design tool give programmers a visual representation of program flow, which is important in assembly languages as they can quickly become long and complicated.
 - Flowcharts as a documentation aid will help others quickly understand the intended purpose and flow of your assembly language source code.
 - Flowcharting is based on a few simple symbols including program flow, process, predefined process, decision, and terminal.
 - The three basic structured programming structures are the sequence, if-then-else, and iterative constructs. If-then-else constructs include case-type constructs while iterative constructs include both do-while and while constructs.
 - Your software is going to have bugs; the best you can hope for is to keep the number of bugs and the damage the bugs cause to a minimum.
 - Verification and debugging of programs usually takes longer than the actual planning and writing of programs.
 - Writing good programs is an art form. If you're not an artist, you can follow a basic set of guidelines to prevent your code from becoming crappy.
-

7.12 Chapter Exercises

- 1) Briefly describe why you can model a computer as a device that “pushes bits around”.
- 2) Briefly describe how an assembly language program is converted into machine code.
- 3) Briefly describe the general purpose of instruction mnemonics.
- 4) Briefly describe why it is that every program ever written and executed on a computer ends up at the machine code level.
- 5) Briefly describe what the term “computer architecture” refers to.
- 6) Briefly describe whether it would be possible to have two different assembly languages be associated with the same computer architecture.
- 7) Briefly describe whether it would be possible to have two different computers use the same assembly language.
- 8) Briefly describe why it is that assembly language programs can quickly become long.
- 9) Briefly describe what an assembler is and what it does.
- 10) Briefly describe what a compiler is and what it does.
- 11) Briefly describe why assembly language programmers need to stay organized with their coding style.
- 12) Briefly describe why it is important for assembly language programmers to understand the hardware associated with the computer they are writing assembly language for.
- 13) Briefly describe why compiler and assemblers are good at knowing there is an error in the code but much less good at figuring out the exact error.
- 14) Briefly describe why it is that a compiler will never be as good at optimizing code as a good and knowledgeable human.
- 15) Briefly describe why it is that you must learn a new assembly language if you move to a different computer architecture.
- 16) Briefly describe what’s the best way to increase the operating speed of a large program written using a higher-level language and compiled?
- 17) Briefly describe why it is that programming in a higher-level language is more portable than programming at the assembly language level.
- 18) Briefly describe the three general differences between different computer architectures.
- 19) Briefly surmise why it is that assemblers are “free” more often than compilers.
- 20) In your own words, describe the main purpose of an algorithm.
- 21) What is the hardware analog to a programming flowchart?
- 22) What are the two main purposes of flowcharts?
- 23) Briefly describe what the notion of a *generic flowchart* refers to.
- 24) Briefly describe why it is a good idea to keep your flowcharts as generic as possible.
- 25) I suddenly got the idea to use a start symbol rather than a diamond symbol for a decision point in my program. Briefly describe why this is a bad idea.
- 26) Briefly describe why it is important to write assembly language code that not only works, but also looks good.
- 27) Briefly describe whether you know good-looking code actually works properly by just looking at the code.
- 28) Briefly describe why it is important to occasionally examine other people’s assembly language code.

- 29) Briefly describe the likelihood that you're going to need to extensively test your assembly language code.
 - 30) Briefly describe the likelihood that you'll actually have time to extensively test your assembly language code.
 - 31) Briefly describe why it's a good idea to always make your assembly language code as simple as possible.
-

8 Introduction to RISC-V Assembly Language Programming

8.1 Introduction

Assembly language programs are not complicated, but they are somewhat different from higher-level language programs you've written. There are many approaches you can take to learning to write assembly language programs; the approach we'll take in this chapter attempts to get you writing programs as quickly as possible. This chapter does not attempt to tell you everything you'll ever need to know about every RISC-V instruction in the instruction set. What we'll do instead is arbitrarily tell you only what you need to know to enable you to write and understand basic assembly language programs. Once you have a basic grasp of writing programs, we'll delve into more of the details regarding writing RISC-V assembly language programs.

The heart of assembly language programming is the instruction set associated with the computer that you're planning on programming. Each assembly language instruction comprises of a set of 1's and 0's that magically somehow control the associated computer's hardware. The notion of the precise 1's and 0's that make up the instruction is low level, so we don't cover them in this chapter. All of these issues fall under the category of "instruction set architecture", or ISA.

Main Chapter Topics

- **INSTRUCTION SET DESIGN:** This chapter covers some of the high-level details associated with designing an instruction set from scratch.
- **ISA DESIGN ISSUES:** This chapter covers discusses a few of the important design parameters associated with ISA design.
- **ASSEMBLY LANGUAGE PROGRAM STRUCTURE:** This chapter outlines the basic and preferred structure of assembly language programs including comments, assembler directives, and assembly language source code.
- **INTRODUCTION TO EMBEDDED SYSTEMS:** This chapter presents the notion of an embedded system as it relates to basic assembly language programming.
- **COMPUTER OVERVIEW:** This chapter once again describes the "big picture" in the context of the RISC-V MCU instruction set and programming model.
- **INSTRUCTIONS OVERVIEW:** This chapter presents high-level views of instructions by describing their general purpose and high-level classifications.
- **RISC-V MCU INSTRUCTION VERNACULAR:** This chapter describes some of the commonly used vernacular describing assembly the RISC-V MCU ISA and associated programming model.

Why This Chapter is Important

This chapter is important because it describes the basic structure of assembly language programs and provides several well-commented assembly language example programs.

8.2 Instruction Set Architecture Design Issues

There are people out there who spend their entire lives delving into the low-level details of instruction sets and particularly, instruction set architectures (ISAs). We're not going to go too deep into the subject in this textbook, but we're going to mention some of the most basic ISA design principles. This is one of those issues where 90% of the work in ISA design goes into the final 10% of the design. What this means is that you can generate a "good" ISA without a super-significant amount of work; most of the work (the 10% part) involves squeezing as much performance out of your ISA as possible. We won't go there.

The approach we take in this text is to allow someone else to do the thinking for us. The result is that the RISC-V is a very well thought-out instruction set, which is the result of the fact that RISC-V is an open-source architecture. The RISC-V is efficient, effective, and highly functional. Possibly the best part about RISC-V is that it is highly extensible, which means we can use it for a beginning class in computer architecture, and later use the same ISA for more advanced courses.

8.2.1 Instruction Set Design

There are most definitely some great theories on instruction set design out there in computerland. The good news is that all of the good stuff was inserted into the RISC-V ISA. The many engineering decisions made along the way add the sparkle to the RISC-V ISA.

If you had to declare the big issues in instruction set design, you would most likely find them related to the type of computer you're designing. Don't lose sight of the big picture: you're solving problems with a digital circuit. To make your solution non-generic (meaning that you can use the same circuit to solve many problems), you decided that your digital circuit will be a computer. You now have a choice: design the computer yourself or use some off-the-shelf solution.

The big issue is that if you design the computer yourself, you can design it with your specific needs in mind. Your design will thus be specific purpose: it does a great job of solving your problem, but probably a not so good job being able to solve "just any problem". On the other hand, if you use some off-the-shelf computer, that computer is most likely going to be a general-purpose computer design. It probably won't solve your problem as good as your specific computer design, but will do a good job on a wide set of problems.

General-Purpose Computer: If you're designing a general-purpose computer, then you don't really know exactly how people will use the computer. It is therefore your job as the ISA designer to provide enough instructions to do "just about anything", which means you'll be including instructions that do generic/typical operations associated with computers/computer hardware. You're essentially guessing what instructions programmers and/or compiler writers will find useful; it's an educated guess, but it's still a guess.

Specific Purpose Computer: If you're designing a specific purpose computer, you'll know exactly how people will use that computer. Designing a specific purpose computer is generally an easier task than designing a general-purpose computer because there is typically no "guessing" involved as to what the computer needs to do. In this case, you include only the instructions you know you will use, thus your computer may not be able to do everything a general purpose computer does, but it will perform your specific task better (faster, less hardware) than the general purpose computer. Keep in mind that you're going to need to write your own assembler and/or compiler to support your computer design.

8.3 ISA Driven Computer Hardware Designs

How do you go about designing a computer? Are there some rules somewhere that you follow? Is there a list of tasks somewhere that you follow and then magically have a computer once you've completed the tasks? I truly don't know the answer to these questions. What I do know is the computer design approach taken by the RISC-V. But first a story. Way back in grad school I took a course in the MPEG standard at the time¹. The general process was to encode the movie in a compressed format, then decoding it to watch the movie. The standard did

¹ MPEG is a standard used to compress and encode motion pictures.

not describe the encoder though: it only described the decoder. I found that shocking at the time. The reason it only defined the decoder was to give designers ultimate flexibility in how they designed the encoder; the only constraint was that whatever they encoded must be able to be decoded by the any MPEG compliant decoder. This story seems pertinent because it relates to the RISC-V MCU computer design.

If you think about it, you may not realize what came first: the hardware or the instruction set. The truth is that the instruction set came first. All subsequent designs are based on the ISA description. Thus, compilers, assemblers, and most importantly for us, the actual RISC-V computer hardware is based on one directive: to do whatever it takes to support the RISC-V ISA. Thus, the ISA came first. The result of this is that the RISC-V MCU hardware is unique. It was primarily an implementation of one person’s ideas to support the RISC-V ISA. The reality is that two different people would probably come up with different designs for the same problem. The only requirement here is that the hardware designs must be able to implement the instructions in the ISA.

For this text, we give you the computer hardware design, thus you don’t have to design hardware yourself. The approach taken by this text is to say: “here is the hardware that will implement the RISC-V ISA: it is your mission to completely understand the hardware, particularly how it implements the given ISA”. It’s very doable, but not trivial. On one hand, it’s only a digital circuit, comprising of standard digital modules that you’re used to working with. On the other hand, the hardware implements a modestly complex computer. The intended learning mission for you is to develop an understanding of the hardware as it relates to the ISA, which then provides you with the tools such that you can design your own ISA and supporting hardware. You’ll thus be able to provide the complete computer-based solution to any problem you face.

8.4 RISC-V MCU Assembly Language Program Structure

This may be your first experience with assembly language, so you may be totally lost at this point. This section aims to provide you with a quick overview of assembly language, programs written in assembly language, and the items provided by running the assembler on your program. This is a quick overview; we go into more detail in later chapters.

8.4.1 The Assembly Language Program

Figure 8.1 shows a simple assembly language program. The program does not do much, but it does contain all the important parts of a program written in assembly language. We follow Figure 8.1 with a description of the important parts of the program with a level of detail that supports your current knowledge of the RISC-V ISA.

```

(01) #-----
(02) # Program: example_program
(03) #
(04) # This program inputs a value, toggles all bits, and outputs the
(05) # value. This program performs these tasks repeatedly.
(06) #
(07) #-----
(08) .text                                # indicate segment with directive
(09)                                     #
(10) init:    li    x10,0xC0008000        # initialize input port address
(11)         li    x11,0xC000C000        # initialize output port address
(12)
(13) main:    lw    x15,0(x10)            # get data from input port
(14)         xori  x15,x15,-1            # toggle all bits in the data
(15)         sw    x15,0(x11)            # output data to output port
(16)         j     main                  # branch and repeat

```

Figure 8.1: A simple assembly language program.

There are three basic parts to any assembly language program: 1) comments, 2) assembler directives, and 3) the assembly language source code. The only thing you need to make your program run is assembly code, but the other parts of your program are important for writing good assembly language programs.

8.4.1.1 Comments

Comments increase the readability and understandability of your programs; you should always use comments if these two qualities are important to you. Comments represent “messages to humans”; more specifically,

comments represent messages from the human writing the code to some other human who may be reading the code. RISC-V code indicates comments with pound signs (“#”). Here are some other fun facts about comments.

- The assembler ignores all the text on a given line after the pound sign
- The pound sign can appear anywhere on a line of code
- There are no “block comments” in RISC-V; each commented line must include a pound sign.
- The code in Figure 8.1 uses comments in two different ways: to describe “big” things such as the overall functioning of the program, and to describe “little” things such as the purpose of a particular line of code.

8.4.1.2 Assembler Directives

As the name implies, assembler directives give the programmer some measure of control over the operation of the assembler. This means that assembler directives are messages from the programmer to the assembler. The code in Figure 8.1 contains only one directive, which is on line (08). Typical assemblers generally have a large set of directives available for the programmer, and the various directives come in different forms. We’ll address the topic of assembler directives in a later chapter.

8.4.1.3 Assembly Code

What makes an assembly language programs a program is the fact that it contains code. The code is the various assembly language instructions. The code in Figure 8.1 contains six assembly language instructions (no need to worry about what they do). The instructions themselves appear indented towards the right. Other things to note are as follows:

- Assembly code only contains one and only one assembly instruction per line. This imposes a fixed structure on assembly language programs that don’t exist in higher-level language programs.
- The various assembly instructions contain a different number of operands.

8.4.1.4 Labels

Labels are an important part of any program because they serve two purposes, and often serve both of these purposes at the same time. They are thus both useful to humans reading the program, and also to the assembler. These labels typically appear quite commonly in assembly language programs. They are actually somewhat like directives in that they are messages to the assembler. Labels are hard to classify what they are as they are not instructions, but they are a “part” of some instructions; but even though they are part of some instructions, they not increase program size. Labels can also can act as comments, because they don’t actually do anything other than provide information to the human reader. This will make more sense when we start writing actual programs in a later chapter.

- Lines (10) and (13) contain labels, which are on the beginning of the lines that contain assembly language programming code. Labels always appear starting in the first column of the assembly code and are always terminated with a colon.
- The label on line (10) serves purely as a comment for human readers. It differs from the label on line (13) because that label is part of the instruction on line (16). This is a common use of labels where main goal is to increase the readability of code for humans without increasing code space.
- The label on line (13) is basically required because it appears as part of the instruction on line (16), thus the instruction relies on that label.

8.4.2 Important Assembly Language Program Formatting

The assembly language programmer has the ability to make the most eloquent assembly language programs possible. These are not “rules”, but it something everyone should follow. You can argue that all you care about is the whether the program works or not, but there is much more to the story. Your mission is to both make the

program work, and to make the program readable and understandable to other humans. A few things to note about the code in Figure 8.1:

- The code contains a “file header” or “file banner” describing the contents of the file. The notion here is that the code for the program is stored in a file.
- The code has many different forms of alignment. The code aligns just about everything: instructions, the first operands in the instruction line, the comments associated with each instruction, and the comments associated with the file header.
- Lines (09) and (12) have nothing on them, but this “whitespace” effectively delineates the various parts of the program. You should always use whitespace to delineate “ideas” in your code. Whitespace does not make your programs longer in the context of computer instructions; it just makes code program text longer.

8.4.3 The Actual Program

People new to assembly language programs often have experience with writing higher-level language programs. The typical higher-level language program runs the associated code, then stops running. The programs we consider in this course are typically associated with “embedded systems”; one major characteristic of an embedded system program is that the run, and keep running, and never stop running (unless you remove power from the underlying hardware).

The program in Figure 8.1 does not have the ability to “stop running”. Note that the program in Figure 8.1 takes the form of an endless loop, the instruction on line (16) is an unconditional branch instruction which direct the program to start executing instructions appearing earlier in the program. An interesting thing to note about the RISC-V ISA is that there is no instruction that directs the computer to “stop executing instructions”. Most ISAs associated with microcontrollers likewise do not contain instructions that “halt” the processor.

8.4.4 Visual Description of Program

The assembly language code for the program can appear intimidating to people new to assembly language. To help sooth these fears and worries, we can of course describe programs with flowcharts. This is a simple program so we don’t expect the flowchart to be overly complicated. Figure 8.2 shows a flowchart the models the operation of the program in Figure 8.1. A few things to note here:

- This is “a” flowchart, and not the flowchart. There are many ways to represent this program using a flowchart, this is one of them.
- The flowchart has a start terminal symbol, but no ending symbol, which models the notion that these programs always run. You’ll later see that the instruction set does not have any instructions that stop the hardware from executing instructions.
- The flowchart shows that the initialization portion of the code only executes once because the first process symbol is outside the loop.
- We included a medium level of detail in the flowchart. For example, we stated “initialize port addresses”, but we did not say which ones and their associated addresses.

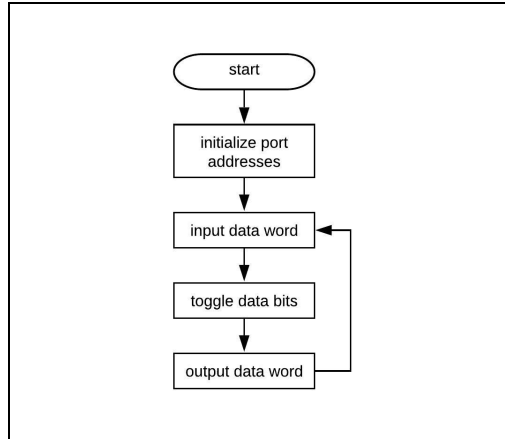


Figure 8.2: A flowchart modeling the operation of this example program.

8.5 What the ISA Really Does

We keep talking about the ISA (instruction set architecture), but what does it really do? We're setting out to use the computer to solve a problem. We'll use an off-the-shelf ISA and implement a computer that supports that ISA. The ISA is the blanket term for the set of instructions that control the underlying hardware of the computer. Figure 8.3 shows a basic high-level description of the underlying hardware, including the three accepted standard modules of a computer. We can thus classify all of our instructions according to what they do to the underlying hardware. Here is roughly what the set of instructions in the instruction set do in the context of the diagram in Figure 8.3. The bullets below roughly represent the arrows in Figure 8.3.

- Some instructions use the I/O block, which allow us to obtain data from and provide data to the outside world
- Some instructions use the microprocessor to crunch bits
- Some instructions store data in the memory block
- Some instructions get data from the memory block
- Some instructions do nothing except determine where in the program to go in order to do the next thing the program needs to do

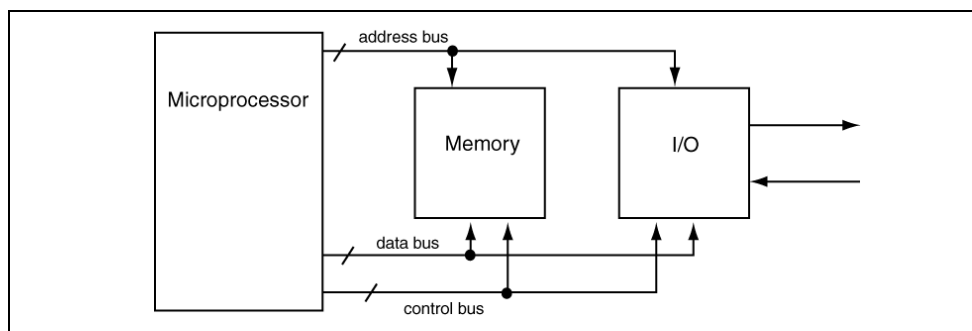


Figure 8.3: The basic computer model at a lower level.

When we describe the operations of the instructions at a high level, they seem rather simple. They are simple, but they can seem daunting because there seem like there are so many of them. Always remember that there's not that much to do in the computer hardware; it becomes a matter of understanding what your program needs to do and how to do it in the context of the underlying hardware. As you'll see as you become more familiar with the instruction set (or ISA), most of the instructions share many similarities. The devil is always in the details.

8.6 RISC-V MCU Assembly Language Basics

Before we start on assembly language, let's review the big picture. There are many details here; we need to start out on the same page.

8.6.1 The Big Picture

Recall that our aim is to use a computer to solve a problem; we'll need to program the computer in order to do that. We can program the computer at three different levels, but we'll be describing programming an assembly language level, which is one step above programming using machine code level and one step below the programming using a higher-level language.

We can model computers at many levels, but let's review the highest level in the context of assembly language programming. Figure 8.4 shows a high-level model of a computer that you've seen before and will work for us here. What we have is a model of hardware that the computer instructions will eventually control such that it will solve a problem for us. Generally speaking, our computer will read in data from the outside world (via the I/O module), that data will be crunched around in some intelligent way (via the Processor module), and then the result is output back to the outside world (via the I/O module). All this stuff happens under control of a program (set of instructions) that is stored on the computer (in the memory module). Along the way we may need to store temporary calculation values in the listed memory model, thus the memory stores both data and instructions.

The RISC-V MCU has a set of instructions that we'll use to program the computer. Relative to the description in the previous paragraph, we can divide our instructions into the following categories. This is a high-level description, but it supports our high-level architecture diagram in Figure 8.4.

- Instruction that pass data between the computer and the outside world
- Instructions that pass data between the various memory modules
- Instruction that crunch data
- Instructions that control the basic flow of the program

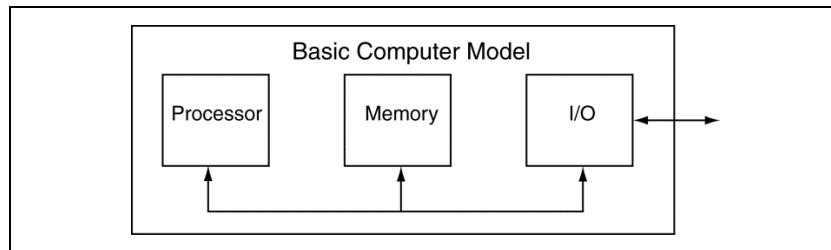


Figure 8.4: General model of a computer.

Let's drop down a level to the instruction set architecture (ISA) level and the Programming Model (or Programmers Model). Recall that the instruction set is the instructions that control the hardware listed in the Programming Model. There is much more hardware in the RISC-V MCU, but we programmers don't have direct control over that hardware via the instruction set. The only thing we're interested in at this point is the instruction set and the hardware we can control with it. The notion here is that if we can properly control the hardware (using programs, which are full of instructions from the instruction set), we'll be able to solve the problem at hand. Figure 8.5 show the RISC-V MCU ISA (a) and the programming model (b).

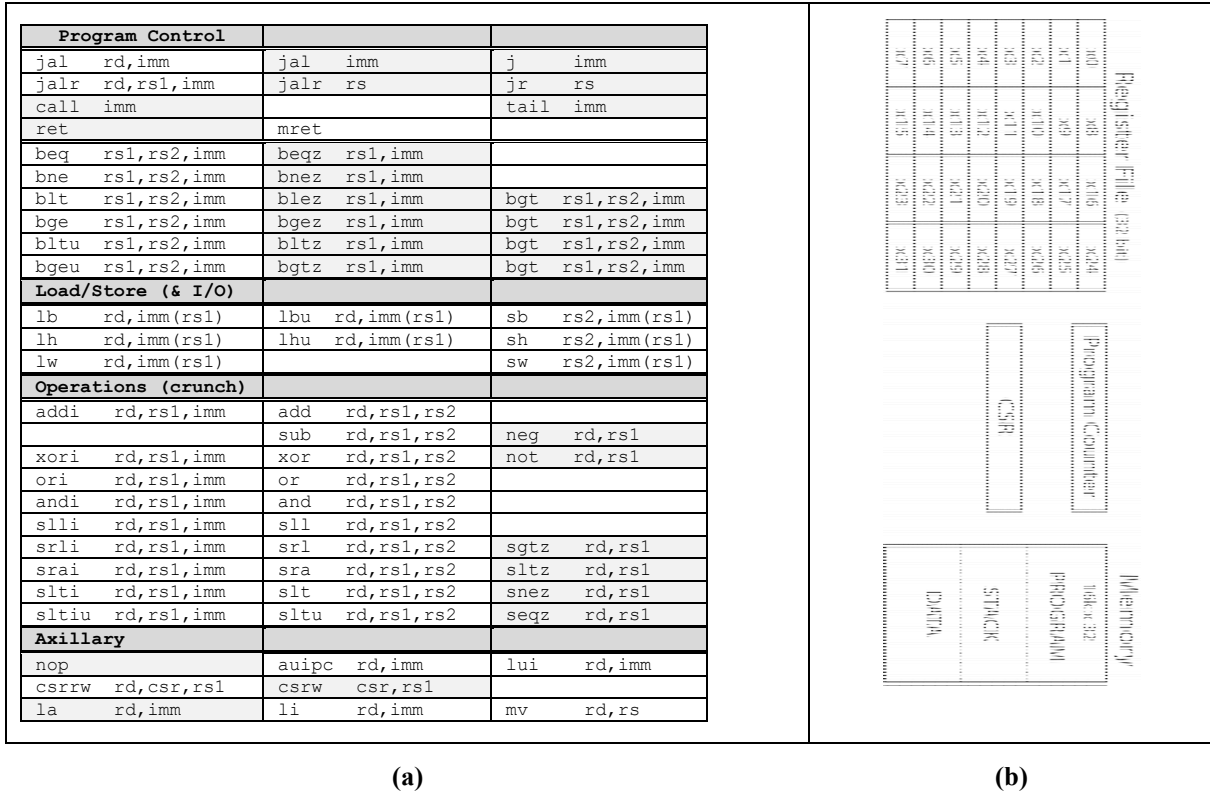


Figure 8.5: The Instruction Set (a) and the Programming Model (b).

We use the instructions in Figure 8.5(a) to control the hardware resources listed in Figure 8.5(b). Note from Figure 8.5(a) that we classified the instruction in heading that indicate what the operations the instructions perform. Also from Figure 8.5(b), we can see that we have some different hardware to control. Even if you’re only going to be a programmer, you need to have a basic understanding of the hardware listed in Figure 8.5(b)². One thing to note in Figure 8.5(b) is that all the resources the instruction set has direct control over is memory of some sort. We can see from Figure 8.5(b) that we have the following memory resources.

Register file: This is 32 32-bit general purpose registers can use to crunch and/or store numbers

Program Counter: this is a register that hold that address of the instruction in Memory that the computer is currently executing

Memory: This stores sets of bits such as computer instructions and various forms of information including data and address information. The “stack” is a special area in memory that we’ll describe later.

Now that we have a more accurate description of the actual RISC-V MCU hardware, we can provide a better description of how the instructions control the computer such that we obtain our desired result. We’ll provide this description in the context of the instruction classifications of ISA in Figure 8.5(a). Remember, this is a quick overview; we provide more details later.

8.6.1.1 Program Control

Programs don’t generally run from the “beginning” to the “end”, which is another way of saying they don’t execute the instructions from memory one after the other, then stop. Typical programs go from executing

² We’re trying to keep the hardware as separate as possible from the software; later chapters in this text deal with the more detailed hardware aspects of the RISC-V MCU. To be a good programmer, you need to have at least a basic understanding of the underlying hardware of the computer your program will execute on. If you have this understanding, you programs will be operate “more better” than if you don’t have any knowledge of the underlying hardware.

instructions from one area of memory to another, so there need to be instructions that support this operation. We refer to this type of instruction as program control instruction because these instructions alter the normal sequential execution of instructions.

We classify program control instructions as “branch” instructions, as they can cause program execution to jump from one area of program memory to another area (not sequential execution). We further classify these instructions as “conditional” and “unconditional” branch instructions. The notion of condition means that we go somewhere else if the conditions are correct; otherwise, we continue on to execute the next instruction in program memory. Unconditional branch instructions always go somewhere else. The RISC-V jump-type instructions always go somewhere else (such as in a subroutine call); the conditional branch instructions may or may not go somewhere else based on some condition of the hardware (such as in an if/else) construct.

8.6.1.2 Load & Store

This is classic computer vernacular that you need to become familiar with. Loading refers to reading something out of memory and writing the data to another memory location such as a general-purpose register (loading). Storing refers to copying something from somewhere such as a general-purpose register and writing that data to memory (storing). We have 32 registers to work with, and we try to do most of our number crunching with registers because they are “faster” than working with memory (a topic for another section). When we run out of registers but still need extra storage, we must load and store data from memory.

The RISC-V MCU of course has I/O. There are several approaches that computers use to perform I/O; the RISC-V MCU uses what we call “memory mapped” I/O. Because of this, we don’t need instructions dedicated to doing I/O. However, what we need to do is use the load instructions for inputting data from the outside world, and use the store instructions for outputting data to the outside world. This works by configuring the hardware to not do normal memory access operations when certain memory addresses are accessed (once again a topic for another chapter).

8.6.1.3 Operations

Computers, and particularly the CPUs in computers, are responsible for “crunching” numbers, or doing special “bit manipulations”. These operations include operations such as adding, subtracting, ANDing, ORing, shifting, etc. The RISC-V MCU has a set of instructions dedicated to crunching numbers. The important thing to note here is that we can only do number crunching using registers. This means if you have numbers in memory that need crunching, you first must load them from memory to the registers.

8.6.1.4 Auxillary

There are also a set of instructions that are hard to place in any of the previous classifications, so we refer to them as the auxillary instructions. Most of these instructions “set up” the hardware to do the right thing when other instructions are executed. These will make more sense when we describe them in a meaningful context.

8.7 Instruction Types

We consider the RISC-V MCU to have two types of instructions, which we refer to as “base instructions” (or just instructions) and “pseudoinstructions”. The hardware only understands the base instructions, but we can use pseudoinstructions to make our programs more understandable to the human reader. The assembler is responsible for converting pseudoinstructions into base instructions. Someone designed the RISC-V MCU instructions to be very versatile; as a result, we can use those instructions to perform special operations; we give these operations new mnemonics of their own and call them “pseudoinstructions”. Figure 8.5(a) uses shading to indicate pseudoinstructions.

The assembler is responsible for translating the pseudoinstructions into real instructions, or a “set” of real instructions. There are two types of pseudoinstructions: ones that translate into one real instruction, and ones that translate into two real instructions.

8.7.1 Instruction Formats: High Level

We generally consider instructions to “operate” one thing; we thus refer to the things the instructions operate on as the “operands”. The different instructions in the RISC-V MCU ISA require a different number of operands to

do their work, depend on what the instruction needs to do. Figure 8.6 shows examples of the various numbers of operands associated with a few example instructions. Generally speaking, the operands are data that exist somewhere, such as in a register or memory, though there is more to it than that. We once again get into the details in a later section.

# of Operands	Example	Comment
0	<code>ret</code>	Pseudoinstruction; translates one instruction
1	<code>call imm</code>	Pseudoinstruction; translates two instructions
2	<code>lui rd,imm</code>	Real instruction
3	<code>and rd,rs1,rs2</code>	Real instruction

Figure 8.6: Examples of various numbers of instruction operands.

8.7.2 Instruction Operand Addressing

This is a common term when dealing with assembly languages, so common that we often forget what it really means. ISAs have “addressing modes”, which is a technical way to state how the instruction specifies where to find the data associated with a given operand that the instruction uses. Once again, think back to the programming model for the RISC-V MCU; there are resources that instructions can manipulate, but the instructions need to be able to specify the exact location of those operands. Table 8.1 shows examples of RISC-V MCU addressing modes.

Address Mode	Instruction Form	Comment
immediate	<code>jal rd,imm</code>	Uses an immediate value as an operand
register	<code>add rd,rs1,rs2</code>	Uses register locations to specify operands
indexed	<code>lbu rd,imm(rs1)</code>	Uses in immediate and value and register contents to generate operand

Table 8.1: Various addressing modes and descriptions.

8.8 Instruction-Related Terminology

We’re almost to the point of learning some of the details associated with instructions. When you read about the instructions associated with any computer hardware, you typically run into a common “vernacular” that the documentation uses to describe that hardware. The same is true for the RISC-V MCU. This section describes that vernacular in enough detail to help you understand the lower-level details once we get there.

8.8.1 Changing Stored Values

All instructions in the ISA change the value of at least one stored item listed in the programming model. Figure 8.7 shows the RISC-V MCU programming model, which once again shows there are three main classifications of what the instructions can change:

General Purpose Registers: There are 32 general-purpose registers that instructions access to store data that is not stored in Memory.

Program Counter (PC): the program counter contains the address in memory of the instruction that the computer is currently executing. The PC advances “normally” in sequential instruction access, but can also load new values to support program control instructions such as jump and branch instructions.

Memory: Memory changes primarily when we store values into it (write operations). There are two ways to transfer non-instruction data into memory 1) copy data from a register (a store operation), and 2) input from the outside world (an input operations). Similar, there are only two

ways to copy data in memory to some other area: 1) copy data from memory to a register (a load operation), and 2) output data from memory to the outside world (an output operation).

Control and Status Register (CSR): we'll discuss this in further detail later

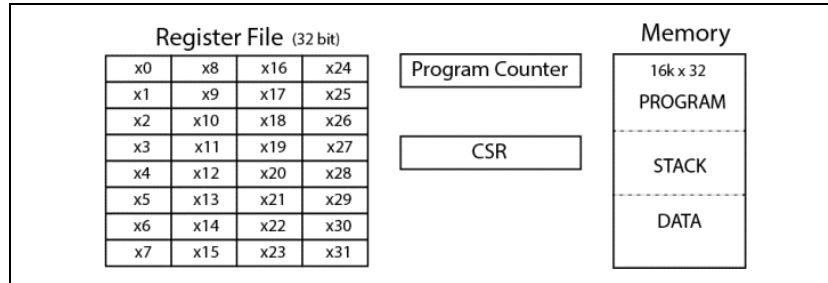


Figure 8.7: The RISC-V MCU programming model.

8.8.2 Alternate Register Names

Figure 8.7 shows the 32 general purpose registers. These registers form the basis of number crunching in the RISC-V MCU. Although we consider these registers to be general purpose, by convention in the RISC-V MCU specification, we consider some of these registers to have alternate purposes, so we give these registers alternative register names. The different register names once again make your program more understandable to humans reading your code. The assembler is responsible to interpreting register names and generating the correct machine code for all instructions. You can use either form of register names in your program, but you should use the ones that make the most sense. Table 8.2 shows the registers listed with their standard “x#” designation and their alternate definition. Here is some information to know about these alternate definitions:

- Register x0 is hardcoded to 0 (the number zero). You can read from this register, but you can't write to it.
- Several registers are special for reasons of varied importance. For this reason, x1 and x2 should not be considered general purpose (meaning, don't use them). We'll tell you the reasons later when it makes more sense.
- The acronym “ABI” stands for “Application Binary Interface” and is common in the RISC-V documentation. We use it here, but may never use it again.
- Many of the ABI registers have standard alternative uses besides x1 & x2. We'll talk about those later as well.

reg	ABI	reg	ABI	reg	ABI	reg	ABI
x0	0	x8	so/fo	x16	a6	x24	s8
x1	ra	x9	s1	x17	a7	x25	s9
x2	sp	x10	a0	x18	s2	x26	s10
x3	gp	x11	a1	x19	s3	x27	s11
x4	tp	x12	a2	x20	s4	x28	t3
x5	t0	x13	a3	x21	s5	x29	t4
x6	t1	x14	a4	x22	s6	x30	t5
x7	t2	x15	a5	x23	s7	x31	t6

Table 8.2: Official and alternate general-purpose register names.

8.8.3 Source and Destination Designations

Many of the RISC-V MCU instructions both access and change register values. The general approach is that instructions may access data in one or two registers, and use that data to alter the data in another register. We

refer to the registers that instructions access but do not change as *source* registers; we refer to the registers that instructions change as the *destination* register. There can be more than one source register but there is never more than one destination register (some instructions don't have destination registers). Instructions use special definitions when referring to source and destination operands. Table 8.3 shows examples of instructions and their operand specification and usage. Note that in Table 8.3, we use the vernacular "rd" and "rsx" to designate destination and source operands, respectively. Don't worry about what the instructions do; you only need to consider the form and names of the operands.

Instruction Form	Comment
jal rd,imm	No official source operand designation; we consider the "imm" value to be a "source"
add rd,rs1,rs2	Two source operands listed as rs1 & rs2.
lbu rd,imm(rs1)	One source operand; the "imm" value is part of the source operand calculation
addi rd,rs1,imm	One source operand; we consider the "imm" value to be a "source" operand

Table 8.3: Example instructions showing source and destination operands.

8.9 Embedded Systems Programming

As we get closer to talking about actual programming, let's describe the ultimate goal. Most assembly language programs end up in some embedded system. An embedded system is a computer-based system that requires no outside user intervention in order for it to run properly. This means the system fires up into a working state and stays working for as long as the system remains powered. Note that this is different from what you may be familiar with in your higher-level language programming courses. The programs you wrote in those courses typically did something relatively useful, and then "ended". The notion of most embedded systems is that the program they are running never ends, unless of course you remove power from the circuit. The reality of embedded systems is that they just sit there waiting to react to inputs or conditions from the outside world.

8.9.1 Software vs. Firmware

Often times when you're generating source code, a question of semantics often arises. When you are writing code, are you writing "software" or are you writing "firmware"? Regardless of the particular hardware you're writing the code for, some portion of memory in the computer you're programming is dedicated to the storage of your program. The instructions that make up your program tell the computer exactly how to process data and what to do with the data it processes. If the user can change program memory, we consider the program stored in memory as *software* and we refer to the computer system a *general-purpose* system. If the user cannot change the program in program memory, we consider the program as *firmware*, as it was written for a *single purpose* computer³.

Another way to view the software vs. firmware argument considers the target platform. In other words, if you're design code that can run on any computer, you're probably writing software. If your program only runs on a specific piece of hardware, then you're probably writing firmware. I remember this by thinking about a program that blinks an LED or writes to a display on a given board. The odds are slim that another piece of hardware will have that same display or same LED, which makes the code you wrote hardware specific, which means it's firmware. If your program runs on every PC in the world, then you've written software.

³ The reality is that most people use the term *software* in reference to true software or true firmware. In most cases, this is OK because you know what the person using this term intended because of the context it was used in. The term *firmware*, on the other hand, is never used to mean software. The biggest mistake that people generally make is that they think that firmware has a direct connection to assembly language programming. The reality is that firmware can be in the form of assembly language or a higher-level language (or both). Don't fall into this trap.

Many people mistakenly conclude that they are writing firmware if they simply writing their source code using an assembly language. According to the definitions of firmware and software, you can write firmware using either assembly language or a higher-level language. Likewise, you can also write software using either assembly language or a higher-level language.

8.10 Chapter Summary

- The act of designing an instruction sets is an independent action of designing the hardware that will be able to execute those instructions.
 - There are three main parts of an assembly language program: 1) comments, 2) assembler directives, and 3) the assembly source code. Comments are messages from the programmer to other humans attempting to understand the code. Assembler directives are message from the programmer to the assembler. The assembly source code is messages from the programmer to the underlying computer hardware.
 - Labels in assembly language programming act as both messages to the assembler and messages to other humans, depending on how the programmer decides to use them.
 - Meaningful assembly source code is neat, structured, and highly organized. It's easy to write crappy assembly language code, but a much better idea is to follow some basic formatting rules to make you source code highly readable and understandable. One the best approaches to generating good source code is to use comments to describe what you're doing and delineate different sections of the code. All languages have associated style-files that show what good assembly code looks like; be sure to access the style-file associated with any assembly code you work with.
 - The instructions in a computer control the computer hardware in meaningful ways. This roughly means that the instructions control the flow of data through the computer in order to help the computer obtain a meaningful result. A typical computer has relatively many assembly language instructions, but those many instructions can be divided into just a few groups based on what the instructions do in the hardware.
 - Assembly code is often associated with embedded systems programming. In typical embedded systems, the associated program never terminates. Likewise, the RISC-V instruction set has no instruction that stops execution of any running program.
 - We can divide the source code for any given assembly language program into various sections, which are standard in embedded systems programming. The two sections discussed in this chapter are, 1) the initialization code, and, 2) the main code. Every assembly language program should have these two sections clearly labeled.
-

8.11 Chapter Exercises

- 1) In terms of instruction set design, briefly describe the two types of computer that someone may ask you to design.
 - 2) In terms of instruction set design, briefly describe why it is “easier” to design a specific purpose computer as opposed to a general-purpose computer.
 - 3) Briefly describe why the design of an instruction set is an independent function of design hardware that could implement that instruction set.
 - 4) What is the range in the number of operands that RISC-V instructions can have?
 - 5) List and briefly describe the three parts of an assembly language program.
 - 6) An assembly language program must include assembly code; briefly describe the main purpose of the other two parts of an assembly language program.
 - 7) The three parts of an assembly language programs provide “messages” to various entities. Briefly describe those entities and the associated messages.
 - 8) What is the first comment that every assembly language source code file should contain.
 - 9) Briefly describe why it is that embedded systems program rarely terminate unless you power-down the hardware.
-

9 Assembly Language Programming Operations

9.1 Introduction

I've been at this juncture before: how am I supposed to teach assembly language programming? A few comments. First, it's hard to teach anything when not in the correct context. The correct context is that we generally write programs to solve problems. Even though we know that computers solve problems, we don't know how to write assembly language programs, so we can't solve any problems yet. We know there are bunches on instructions, and we've probably programmed using a higher-level language, but assembly language is significantly different and it's hard to make the connection between this strange new language and problem solving. Second, so much of the information you need to program in assembly language is based on other information that you don't know yet.

The solution is to start somewhere. If it seems strange at first, please know that it will seem less strange as you understand more and start writing actual assembly language programs. My feeling is that you should read a lot of stuff relatively fast in order to get a feel for the material, then go back and read it slowly so that you completely understand the material.

Main Chapter Topics

- **INPUT/OUTPUT:** This describes the various approaches in to performing input and output operations, with an emphasis on memory mapped I/O, which is the approach the RISC-V architecture uses.
- **INTRODUCTION TO INSTRUCTIONS:** This chapter introduces the first set of basic RISC-V commonly used in assembly language programs, including data transfer and bit-crunching instructions.
- **MEMORY ACCESS INSTRUCTIONS:** This chapter introduces the instructions that the RISC-V MCU uses to access main memory.
- **MICROCONTROLLER INPUT/OUTPUT:** This chapter describes the basic forms of input/output architectures, with an emphasis on memory-mapped I/O.

Why This Chapter is Important

This chapter is important because it represents an introduction to the RISC-V instruction set in such a way as to be able to write basic RISC-V programs.

9.2 Basic Instructions and Usage

Recall that a computer is roughly a device that inputs data, churns it around, and then outputs it. This being the case, the approach we'll take to introducing assembly language programming is to start with instructions that input and output data, and instructions that crunch data. Our goal here is to present some basic functionality in order to be able to present/describe the remainder of RISC-V instructions in a more meaningful context. This is going to be easier than it sounds. We'll start with a data transfer instructions and then move onto data crunching instructions.

9.2.1 The First Data Transfer Instruction

The heart of the RISC-V data crunching mechanism is the set of registers, which we refer to at the *register file* (or *reg file*). What you'll see is that all data crunching operations in the RISC-V involve the registers. For this discussion, the most basic data crunch is the transfer of data from one register to another, with some crunching in

between. For this operation, we don't actually crunch the data, but we do move the data around, which is an operation we actually do quite a bit in assembly languages.

We chose this data transfer because it illustrates two points. First, we see our first assembly language instruction, which we find out is actually a *pseudoinstruction*. Second, we examine the base instruction the assembler uses to implement the pseudoinstruction.

9.2.1.1 The `mv` Pseudoinstruction

Transferring data from one register to another is probably the most basic operation on the RISC-V MCU. We use the `mv` instruction to make register-to-register transfers, with the notion that we're "moving" data from one register to another (hence, the symbolic name "`mv`"). Table 9.1 shows most of the useful forms of information regarding the `mv` instruction. Here is some other information about Table 9.1:

- The **Instruction Form** column shows the basic form of the instruction, where `rd` is the destination register and `rs1` is the source register. Some instructions have two source registers, which is why we attach a '1' to the `rs`.
- The RTL column shows what the instruction does using register transfer language. Notice that the RTL form highlights the painful notion that the data is transferring from the right operand to the left operand¹.
- Table 9.1 also provides two examples of the instruction as it could appear in a program. You can see that we replace the `rd` and `rs1` registers from the Instruction Form column with actual RISC-V MCU register names. The second example uses alternative register names with the `mv` instruction where `t3` and `a4` are equivalent names of `x13` and `x28`, respectively.
- For the top example in Table 9.1, executing the instruction copies the data in register `x11` into register `x10`. The data in `x11`, the source register, does not change. The data in `x10` changes² because it is loaded with the data from the source register `x11`. Executing this instruction results in the loss of information in `x10`.

Instruction Form	RTL	Examples
<code>mv rd,rs1</code>	<code>rd ← rs1</code>	<pre>mv x10,x11 # copy x11 into x10 mv a4,t3 # copy t3 into a4</pre>

Table 9.1: An overview of the `mv` instruction.

Every instruction in the RISC-V MCU instruction set has an extended description in the associated assembly language instruction manual. Table 9.2 shows the entry for the `mv` instruction. The information provided is all the pertinent information regarding the `mv` instruction. Here are the important things to note about Table 9.2:

- The RTL has a different form, which uses the `rd`, and `rs1` values as indexes into what appears to be an array named "`X`". This array notation refers to the register file, thus the array in question is zero-based and has 32 elements (0→31), which is C (and thus Verilog) notation. Prepare to become accustomed to the syntax.
- There is an extended written description for the instruction, which says what we've already been saying in the previous verbage.
- The "Usage" information provides more information in the example. The example reminds us that all the registers in the reg file are 32 bits. Note that we place an underscore in the middle of the

¹ This is not overly intuitive, but is typical in assembly languages based on the notion that early MCUs first used this right-to-left notation and most everyone else followed.

² Data in `x10` will not change if the data in `x11` is equivalent to the data in `x10` before the instruction executes. I hate to be nitpicky, but I thought you'd like to know.

eight hex characters, which is an artifact from Verilog that we adopt to make the hexadecimal string more readable.

- The extended description also has a “See Also:” area, which list related instructions.
- Most importantly, we see that the **mv** instruction is actually a pseudoinstruction, which is significant for several reasons. First, the extended description provides no instruction type or instruction format that we see in base instructions. Second, we become interested in which base instruction the assembler replaced the **mv** instruction with to make this work. It so happens that the assembler replaces this instruction with an **addi** instruction, which we’ll talk about next.

mv	move	(pseudoinstruction – addi)
RTL: $X[rd] \leftarrow X[rs1]$	Form:	mv rd, rs1
Description: The mv is a pseudoinstruction based on the addi instruction. The mv instruction copies the contents of the source register rs1 into the destination register rd . The contents of the source register do not change. The mv instruction is equivalent to the following instruction: “ addi rd, rs1, 0 ”.		
Usage:	<pre>mv X10,X11 # copy the contents of source register X11 into # destination register X10 # X10=0x021F_3B0D X11=0345_668A (before exec) # X10=0x0345_668A X11=0345_668A (after exec)</pre>	
See Also: addi		

Table 9.2: The description of the mv instructions from the RISC-V MCU assembler manual.

Example 9.1: mv Code Fragment

Write a fragment of RISC-V assembly language code that does the following three operations:

- 1) Copies value in register x20 to register x21
- 2) Copies value in x31 to x2
- 3) Clears the value in x10

Solution: Figure 9.1 shows the solution to this example. There are several particularly important things to note about this solution:

- The code is only a fragment; it’s not a program or a subroutine. Additionally, the choice of registers used in this example is arbitrary, except for the x0 register (see comment below).
- The **mv** instruction on line (02) copies the value in register x20 to register x21. The previous value in register x20 is lost because the instruction overwrites it with the value in x21. The instruction does not change the value in x21. Yes, this does feel backward in that the instruction copies the operand on the right into the operand on the left. This is an artifact from early computer days but is common practice in most computer hardware documentation.
- The **mv** instruction on line (04) copies the value in x31 to x2. The previous value in x2 is lost; the value in x31 does not change.

- The instruction on line (06) clears the value in x10 because the value in register **x0** is always zero. We have a choice of instructions when clearing register values, but using the **mv** instruction is the best approach to setting any register value to zero.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)   mv   x21,x20      # copies value in x20 to x21  1)
(03)
(04)   mv   x2,x31      # copies value in x31 to x2  2)
(05)
(06)   mv   x10,x0      # clear value in x10 (make zero)  3)
(07) #
(08) #~~~~~ program fragment ~~~~~

```

Figure 9.1: Solution for this example.

9.2.2 The Second Data Transfer Instruction

The **mv** instruction provided a means to transfer data from one register to another register. While, this is useful, it's not always what programmers need to do. Another major form of data transfer is from an immediate value to a register. We perform this data transfer using the **li** instruction. Table 9.3 shows an overview of the **li** instruction. The high-level view of this instruction is relatively simple so we won't provide an in-depth description.

Instruction Form	RTL	Examples
li rd,imm	$rd \leftarrow \text{imm}$	<pre> li x10,0x23 # put 0x23 in x10 li x12,0x1100C000 # put 0x1101C000 in x12 </pre>

Table 9.3: An overview of the **li** instruction.

The **li** instruction is similar to the **mv** instruction in that it is a pseudoinstruction. There is an important difference, which is worth knowing to programmers. While the assembler instruction always translates the **mv** instruction to a single base instruction (**addi**), the assembler translates the **li** instruction to either one or two base instructions. The assembler translates the **li** instruction to an **addi** instruction if the associated immediate value can be represented using 12 bits (the width of the immediate field in the **addi** instruction). If the associated immediate value can't be represented using 12 bit, the assembler translates the **li** instruction to two base instructions (**addi** & **lui**). Table 9.4 shows the assembler manual entry for the **li** instruction. Here are a few things to note regarding the **li** instruction:

- Programmers should remain aware of the fact that the size of the immediate value in the **li** instruction determines how many base instructions the assembler uses to represent the **li** instruction. The notion here is that we always try to write programs that are both space efficient and time efficient; reducing the number of instructions in our programs generally does both.
- Yes, there are some underlying details involved with the actual encoding of the **li** instruction. Note that we opted to not describe the **lui** instruction as part of our **li** instruction overview. The good news is that we can use the **li** instruction without worry because the assembler makes the one vs. two-instruction decision for us once it determines the size of the imm value operand in the **li** instruction.

li	load immediate	(pseudoinstruction – addi)
RTL: $X[rd] \leftarrow imm$	Form:	li rd,imm
Description: The li instruction writes an immediate value to the destination register <i>rd</i> . This is an pseudoinstruction and is equivalent to the following instruction: “ addi rd,X0,imm ” if the immediate value can be represented with the 12-bit immediate field in the addi instruction, or a combination of two instructions (addi & lui) if the immediate can't be represented by a 12-bit immediate value.		
Usage:	<pre> li X9,1023 # write an immediate value into destination register X9 # X9=0x021F_3B8A (before exec) # X9=0x0000_03FF (before exec) </pre>	
See Also: addi , lui		

Table 9.4: The assembler manual entry for the **li** instruction.**Example 9.2: li Code Fragment**

Write a fragment of RISC-V assembly language code that does the following three operations:

- 1) Loads the value -1 into value in register x10
- 2) Places the value 0x134FADE8 into register x28
- 3) Copies the value -0x34 into register x17

Solution: Figure 9.2 shows the solution to this example. And yes, there are several particularly important things to note about this solution:

- This code fragment provides three examples of a **li** instruction, yet the problem states three different accepted forms of vernacular to describe the problem.
- The instruction on line (2) places 0xFFFFFFFF into x10. The RISC-V uses 2's complement notation for representing negative numbers. The assembler makes the translation from signed decimal in the code to 32-bit 2's complement in the actual hardware.
- The instruction on line (4) places the 32-bit hex value into register x28; no need for fancy translations here.
- The instruction on line (6) places the 2's complement representation of -52 (-0x34) into register x17. The actual value placed in x17 is 0xFFFFFCC, which is -52 in 2's complement notation. The assembler handles all the base conversions for us relatively mathematically challenged humans.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)   li    x10,-1          # copies value in x20 to x21
(03)
(04)   li    x28,0x134FADE8 # copies value in x31 to x2
(05)
(06)   li    x17,-0x34      # clear value in x10 (make zero)
(07) #
(08) #~~~~~ program fragment ~~~~~

```

Figure 9.2: Solution for this example.

9.2.3 The First Data Crunching Instruction

The first instruction we looked at was a data transfer instruction, which turned out to be a pseudoinstruction. When we look deeper at the **mv** instruction (or read the assembler manual description), we see that when we use a pseudoinstruction such as **mv**, the assembler replaces it with a base instruction. In this case, the assembler replaces the **mv** instruction with the **addi** base instruction.

We refer to the **addi** instruction as an “immediate” instruction because one of the operands is an immediate value and it uses that operand to calculate the value it places in the destination operand. The mnemonic for the instruction includes the word “add” which indicates to humans that this instruction is adding two values. The “i” at the end of mnemonic indicates that one of the operands appears in the instruction as an immediate value rather than being located in a register. Table 9.5 shows preliminary information about the **addi** instruction with several examples. The important issues in Table 9.5 include:

- The instruction form column shows the instruction using both immediate and register addressing. One of the source operands is **rs1**, where the “r” implies that the data is in a register. The other operand is “**imm**”, which implies the other operand is provided in the instruction as a number. The destination operand, **rd**, once again has an “r” prefix, which means the instruction places the result of the addition into the destination register.
- The RTL column shows that the instruction adds the two source operands (**rs1 & imm**) and “transfers” the result to the destination register. The **addi** instruction does not change the source operands and can only change the destination operand.
- The four examples show typical usage of this instruction as it would appear in a source code listing.
 - 1) The first example uses a “0x” prefix to indicate that we are listing the immediate value in hexadecimal. The example instruction only lists two hex characters in an effort to save space, but could have listed more characters. As you see later, there are limitations on the magnitude of the immediate value for this instruction.
 - 2) The second example uses alternate register names and specifies the immediate value in decimal. Note that the assembler interprets numbers without escape character prefixes (such as “0x”) as decimal.
 - 3) The third example shows that the destination register can also be the source register³. In this example, unlike the other examples, the value in the source register does change because the source and destination registers are the same. We use this example extensively in assembly language programming because it represents a *decrement* of register x15.
 - 4) The fourth example is of special interest to us because when we use the **mv** pseudoinstruction, the assembler translates that instruction to something like this example (only the destination register name is different). The instruction completes a “move” by using zero as an immediate value; when we add zero to the source register, it does not change the value in the source register and the result of the addition is stored in the destination register. In other words, the instruction copies the value in the source register to the destination register.

³ Use of a source register as a destination register is very common in assembly languages. It’s easily described in hardware, but we’ll save that description for another chapter.

Instruction Form	RTL	Examples
addi rd,rs1,imm	$rd \leftarrow rs1 + \text{imm}$	<pre>addi x12,x11,0x75 # add 0x75 to x11; # store result in x12 addi a2,t1,34 # add 34 to t1; store # result in a2 addi x15,x15,-1 # add -1 to x15; store # result in x15 addi x20,x25,0 # transfer x25 value to x20</pre>

Table 9.5: An overview of the `addi` instruction.

The `addi` instruction also has a complete description in the assembler language instruction manual. We’ve included the entry for the `addi` instruction in Table 9.6. There is some very important information in the `addi` instruction description that was no in the `mv` description based on the fact that `addi` is a base instruction while `mv` is a pseudoinstruction. Here are the important differences:

- We know that the instruction adds a register value to an immediate value, but there is more to it than that. The registers are 32 bits wide, but the magnitude of the immediate value is limited to 12 bits, which we can see from the image in the “instruction format” row. Additionally, the assembler interprets the 12-bit immediate value as a signed value, which means the assembler is going to interpret the left-most bit of the 12-bit value as a sign bit. Because the instruction is doing 32-bit arithmetic, the hardware sign extends the 12-bit value to create a 32-bit value before it does the addition, which the RTL description states with the “`sext(imm)`” notation⁴.
- The Instruction Format row contains two types of information worth noting. First, the `addi` instruction is an “I-type” instruction, which is one of the six instruction formats used in the RISC-V ISA. The row also shows an image of the underlying bit values for the instruction. Recall that the assembler translates each mnemonic such as “`addi`” and associated operands into machine code. The image in this row shows the associated machine code for this instruction. From that machine code, you can see that there is only 12 bits available to encode the immediate value associated with the instruction. Also, some areas have numbers in them (opcodes) and some areas have labels in them (field codes); these are hardware issues that we don’t need to be aware of as programmers and we’ll thus discuss them in the hardware portion of this text.

⁴ The use of “`sext`” is common in the RISC-V documentation and we always interpret this as sign extension. Recall there is also a notion of zero-extension.

addi	<i>addition with immediate</i>		
RTL: $X[rd] \leftarrow X[rs1] + sext(imm)$	Forms:	addi <code>rd,rs1,imm</code>	
<p>Description: The add instruction performs an addition operation on the operand rs1 and the immediate value and stores the result in the destination operand rd. The instruction overwrites value in the destination operand; the source operand is not affected unless it specifies same register as the destination. The 12-bit immediate value is sign-extended before addition. Both source operands are treated as signed values in 2's complement format. The addi instruction ignores any arithmetic overflow resulting from the operation.</p>			
Instruction Format (I-type)			
Usage:	<pre>addi X10,X11,0x0DC # addition of values in X11 to 0xDC # result stored in X10; X11 is not affected. # X10 = 0x0000_0045 X11 = 0x0000_0024 (before exec) # X10 = 0x0000_0100 X11 = 0x0000_0024 (after exec)</pre>		
See Also:	<code>add</code> , <code>sub</code>		

Table 9.6: The description of the **addi** instruction from the RISC-V MCU assembler manual.

Example 9.3: **addi** Code Fragment

Write a fragment of RISC-V assembly language code that does the following three operations:

- 1) Add the value 0x12345678 to register x29 and store the result in x11
- 2) Increment the value in register x15
- 3) Decrement the value in register x22
- 4) Subtract the value of 30 from register x12 and store the result in x8

Solution: Figure 9.3 shows the solution to this example. And yes, of course, there are several particularly important things to note about this solution:

- This code fragment provides four examples of typical **addi** instruction usage but uses some new and interesting wording.
- The instruction on line (2) adds 0x12345678 to the value in register x29 and stores the result in register x11. The instruction does not alter the value in x29 but does alter the value in x11.
- The instruction on line (4) adds 1 to the value in x15, which is a classic “increment” operation. This instruction uses x15 as both a source operand and destination operand, which is typical in assembly language programming.
- The instruction on line (6) subtracts 1 from the value in x17 (same as adding -1), which is a classic “decrement” operation. The assembler handles all the knarly 2’s complement details.
- The instruction on line (8) subtracts 30 from the value in x12 and stores the result in x8. The instruction does not alter the value in x12. The assembler does the all the 2’s complement conversions for us wimpy human programmers.


```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)     addi   x11,x29,0x12345678    # adds 0x12345678 to value in x11
(03)
(04)     addi   x15,x15,1            # increments value in x15
(05)
(06)     addi   x17,x17,-1           # decrement value in x17
(07)
(08)     addi   x8,x12,-30           # subtract 30 from x12; store result in x8
(09) #
(10) #~~~~~ program fragment ~~~~~

```

Figure 9.3: Solution for this example.

9.2.4 Memory Related Data Transfer Instructions

One of the three computer subsystems is “the memory”. When we refer to memory in the RISC-V MCU architecture, we refer to the larger structured memory, or the “main memory”. There are other memory resources such as the program counter, the register file, the control unit, etc., but when we say “memory” in the context of the RISC-V MCU, we are generally referring to the large memory.

Since this section of the text deals with programming, we prefer to deal with the RISC-V MCU programming model, which we show in Figure 9.4. The programming model shows the resources available to the programmer, which the programmer can control using instructions in the instruction set. As you can see in Figure 9.4, all of the instruction-controllable features in the RISC-V MCU are sequential elements. Additionally, Figure 9.4 also shows that main memory serves two purposes: stores the program (PROGRAM) and stores data (STACK & DATA). The diagram lists the overall size of main memory as 16k x 32, but there is more to that number, which we’ll discuss shortly.

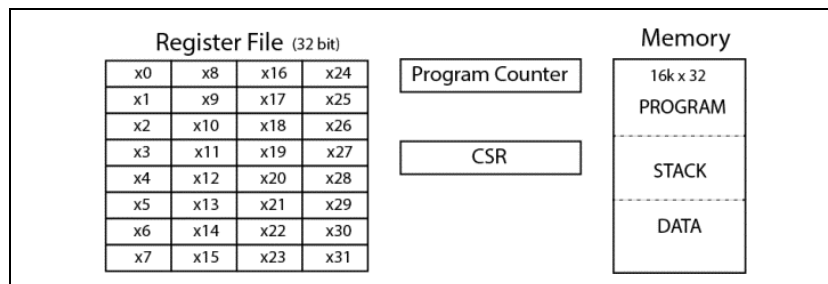


Figure 9.4: The RISC-V MCU programming model.

9.2.4.1 RISC-V Main Memory

The main memory in the RISC-V OTTER serves two purposes: part of it stores the program, and other parts of it store data. The non-program storage part of memory also supports hardware related items such as the stack and heap, which are items we deal with in a later section. Memory is memory; there is nothing inherently special about it other than the fact that it stores data. The implication here is that we can use instructions to write to that memory and read from that memory, which are actions that fall under the category of data transfers. Recall that storing data is a write operation that changes a value in the memory, while loading data is a read operation that does not change any data in the memory.

The RISC-V main memory is quite specialized. It’s a memory in the sense that you can read from it and write to it, but it’s special in the way it stores/accesses program memory and data memory. Most of these details are out of the scope of the programming section of this text, so we save them for another chapter. The overall size and accessibility of memory are two issues that we need to mention here, as programmers need to be aware of the details.

The main memory stores both instructions and data. When we state the capacity of main memory, we generally do so using two different metrics. All RISC-V MCU instructions are 32-bits wide, so all memory accesses associated

with instruction memory output are 32-bits. This being the case, we often refer to main memory as being 16k x 32; this means the memory can hold 16k instructions with each instruction being 32 bits wide. On the other hand, the main memory also stores data. The “data” portion of the memory is *byte addressable*, which means we can access (read and write) individual bytes. Because of this, it sometimes makes more sense to speak of the memory capacity in terms of bytes, or 64k x 8. Note that the overall bit capacity is the same, but we refer to it using two different metrics depending on context.

9.2.4.2 Accessing Main Memory Data

There are two things you can do with memory, read it and write it. In computer terms, we refer to reading data as “loading” data, which generally means we read data from memory and copy it somewhere without changing the data in memory. We refer to writing data as “storing” data, which generally means we copy data from somewhere and overwrite some data currently in memory. The notion of instructions that access data include two types of instructions: *load* instructions that read data from memory, and *store* instructions that write data to memory. Here are the two most important things to note about load and store instructions in the RISC-V ISA:

- 1) Load and store instructions always involve registers. More specifically, load instructions read data from memory (a read operation) and copy that data into a register (the value in memory does not change, the value in the register does). Store instructions copy data from a register into memory (a write operation), which necessarily changes that value in that specific memory location but does not change the source register.
- 2) Once again, memory is memory, so any read or write operation (load or store) means we’ll need to provide an address to read or write from. Additionally, if we’re writing to memory (store), we need to also provide the data. For read operations (loading), we need to provide a destination to copy the read data to. Yes, memory has control signals also, but the underlying hardware takes care of those details for us programmers.

Loading and storing information from/to memory on the programming level is not much different from the same actions on the hardware level. While we don’t have to worry about the underlying control signals, we are responsible for specifying an address of the data in memory we’re accessing, and a place to put that data (loading) or a place to get the data from (storing). The RISC-V ISA uses special notation for accessing memory; this notation is similar for both load and store instructions. The similarity is that the instructions specify the memory address in the same way; the differences are that the load and store instructions specify a destination or source register, respectively. Table 9.7 provides an overview of the load and store-type instructions. Here are the important things to note in Table 9.7:

- Both instructions use the same specific syntax to allow the programmer to specify the memory address for the memory access instruction. The memory address is a summation of the “imm” value (immediate) and the source register (rs1). In the case of memory access instructions, the *base register* is a source register. The base register holds a 32-bit value, but the assembler encodes the imm value as a signed 12-bit value, so the imm value is sign-extended before the hardware adds it to the base address. This of course means that the immediate value can be a negative number, which we’ll see later in some actual examples.
- The left-most operand in the load-type instruction is the destination register, which is the register where the instruction copies the data from memory into. The left-most operand in the store-type instruction is a second source register, which holds that data that will be copied into the memory at the address specified by the right-most operands.
- Both forms of memory access instructions use what we refer to as “indexed addressing”. The notion here is that the imm value is an index, which uses the rs1 source operand as an “initial address”. We typically refer to the imm value as an *offset* and address source operand as the *base address*. This type of addressing mode is common in assembly languages.

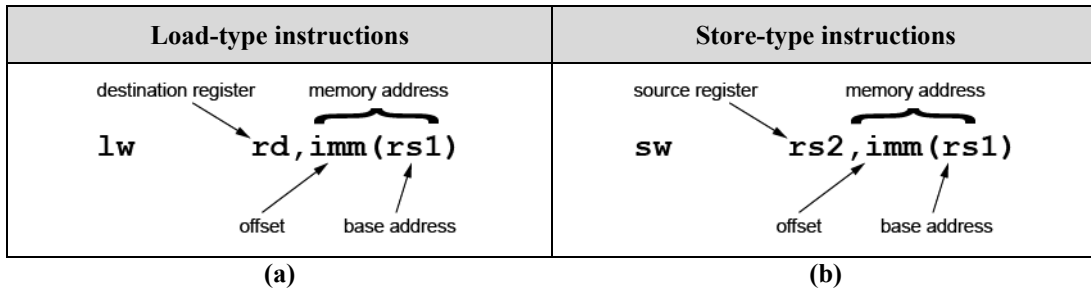


Table 9.7: Overview of the load and store-type instructions.

There are five types of load instructions and three types of store instructions in the RISC-V ISA. In order to keep this discussion simple and uncluttered, we'll only be discussing the **lw** (load word) and **sw** (store word) instructions in this section. Table 9.8 shows an overview of these two instructions including examples.

Load		
Instruction	Example	Comment
lw rd, imm(rs1)	lw x7, -4(x22)	Loads word from memory address = (-4 + value in x22) into x7
Store		
sw rs2, imm(rs1)	sw x5, 0x34(x8)	Stores contents of x5 into memory address = 52 + value in x8)

Table 9.8: An overview of **lw** and **sw** instructions.

Example 9.4: Code Fragment Using **lw** & **sw** Instructions

Write a fragment of RISC-V assembly language code that does the following three operations:

- 1) Copies the data in main memory address 0x0000F004 to x13
- 2) Copies the value in register x16 to the main memory address stored in x18
- 3) Copies the word two words past the main memory address in x20 to x31
- 4) Stores the value in x18 to main memory address 0x24

Solution: Figure 9.5 shows the solution to this example. And yes, there are several particularly important things to note about this solution:

- This code fragment provides four examples of a **sw** & **lw** instructions, yet three of the problems do not use the words “load” or “store”. Typical MCU vernacular.
- The instructions on lines (02-03) takes care the of part 1). For this problem, we need to first get the address value into a register, which we do on line (02); using x10 was an arbitrary choice. The instruction on line (03) then copies the word at the address in x10 into x14, there is a zero offset value provided, so the effective address is what is in x10. The instruction does not change the value in x10 after the instruction on line (02). Note that this operation required two instructions based on the magnitude of the address: 0x0000F004 can't fit into the 12-bit immediate value associated with the **lw** instruction.
- The instruction on line (05) store the word in x16 into memory at the address in x18. Note there is a zero offset value, which means the value in x18 is the effective address.
- The instruction on line (07) copies the value from eight greater than the address in x20 into x31. In this case, the instruction uses the offset to advance the value 8 past the address in x20; the instruction uses

“8” because that represents two “words” worth of data based on the number of bytes in two words (which is 8). The underlying RISC-V hardware generates the effective address by adding the value of 8 to the value in x20.

- The instruction on line (09) stores the value in x18 into address 0x24. In this case, the immediate value, 0x24, is the effective address because the instruction uses x0 as the base register.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)     li    x10,0x0000F004    # copies value in x20 to x21
(03)     lw    x13,0(x10)       # load data at address x10 into x13
(04)
(05)     sw    x16,0(x18)
(06)
(07)     lw    x31,8(x20)       # copies value in x31 to x2
(08)
(09)     sw    x18,0x24(x0)     # clear value in x10 (make zero)
(10) #
(11) #~~~~~ program fragment ~~~~~

```

Figure 9.5: Solution for this example.

9.3 Input/Output (I/O)

The architectural diagram Figure 9.6 represents another model of a basic computer system. You can see that the microprocessor is able to communicate with the other blocks in the computer system. For this section, we are mainly interested in how the microprocessor communicates with the outside world in the context of the RISC-V MCU instruction set. Keep in mind that the only reason that computers are useful is because they are able to communicate with the outside world. It’s true that computers crunch data really fast, but this speed would be useless if it were not able to transfer data such as results to and from the external environment. Lastly, also keep in mind that there are many hardware aspects to this communication that we’re omitting from this section and leaving for our hardware-related discussion of the I/O.

Communications with the outside world occur through the I/O block as Figure 9.6 indicates. The microprocessor is responsible for crunching data and the memory is responsible for storing the program and intermediate results. The I/O block is typically a placeholder of sorts; the block does not necessarily imply external hardware is involved. Once again, there is a lot to this story; but limit the discussion to what we need to know for this section and cover the full details later.

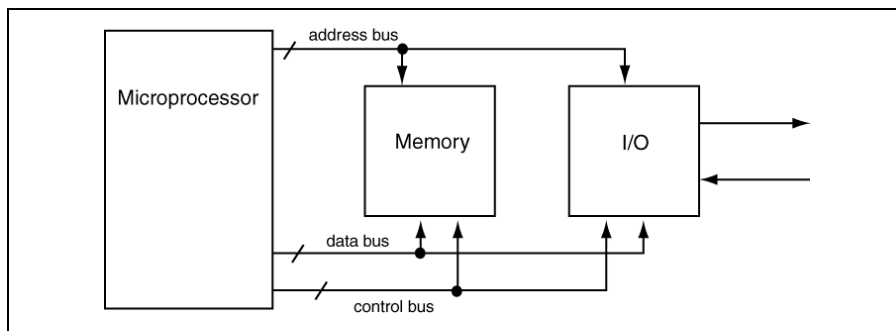


Figure 9.6: Generic computer architecture diagram.

There are actually several main types of standard approaches for computers to communications with the outside world, including, 1) Programmed I/O, 2) Interrupt Driven I/O, and 3) Direct Memory Access (DMA). Below is a brief description of each.

- 1) Programmed I/O: Programmed I/O falls into one of two main categories: *Port mapped* and *memory mapped*. These two categories are similar from a programmer’s perspective; their main

differences lie in the underlying hardware implementation. We refer to this approach as “programmed” I/O because true input or output happens as a result of the program issuing a dedicated input or output instruction. The main thing to keep in mind about performing I/O is that the MCU is getting (input) from something or giving (output) to something; therefore, the program must both state what it is you’re getting or giving and which external devices you’re getting it from (input) or giving it to (output). An output-type instruction provides both a source of data internal from the computer to output to the outside world, and some type of specification as to which external device to output that data to. Similarly, an input instruction provides a destination within the microprocessor to receive data from the outside world as well as a specification of which external device to receive that data from.

Each external device (both input and output devices) has a unique value, which we typically refer to as an “address” or “port address” that the programmer uses to specify which external device the I/O instruction is intending on communicating with. The differences between port mapped and memory mapped I/O lies in how exactly you specify the external device you’re performing the I/O with (the source for inputting and the destination for outputting). The next two items describe those differences in more detail.

- a) **Port Mapped I/O:** Port mapped I/O uses a “port number”, or “port ID”, or “port_id” to specify the external device associated with the given I/O instruction. The port_id is simply a number; roughly speaking, the external hardware uses (or, “decodes” maybe be a better word) the port_id to “activate” a given I/O device. In this way, every I/O device necessarily has a unique port_id number. The port_ids are a function of the hardware; if you’re writing assembly code for a given piece of hardware, someone must tell you, the programmer, the specific port_ids for the I/O devices associated with your given system. We consider architectures that use port mapping as having *separate address space* for I/O, which may seem strange, but makes sense when after you read about memory mapped I/O. Finally, typical port mapped architectures have dedicated instructions for I/O, such as IN and OUT instructions for input and output operations, respectively.
 - b) **Memory Mapped I/O:** In contrast to port mapped I/O, memory mapped I/O does not have dedicated IN-type and OUT-type instructions. The memory-mapped approach uses memory access instructions to handle I/O. As you may guess, memory access instructions must provide an address in memory of the item you’re trying to write or store (output) or trying to read or load (input). In a memory-mapped architecture, the hardware designer configured the hardware such that if you read or write from memory using a “special address” associated with an external device, you’re not really reading or writing memory; you’re actually inputting data from or outputting data to a particular external device, respectively. Each I/O device has its own unique address similar to port address in port mapped I/O. In memory-mapped systems, we consider the I/O to be sharing the address space with the data memory (recall that port mapped systems have a separate I/O space). Once again, the hardware designer must provide the programmer with the address values associated with various I/O devices; you would not know the addresses otherwise.
- 2) **Interrupt I/O:** There are some special issues associated with programmed I/O. In rough terms, the MCU is not always inputting or outputting data: it only does so when it needs to. The problem is figuring out when it needs to or not. If you don’t use interrupt-driven I/O, the MCU needs to expend clock cycles to determine when it needs to do I/O. The problem arises when dealing with input. Many peripheral devices require that you “get data from them” when they’re ready to give the data to you. The problem is that you generally do not know when such devices are ready to give you data, so the only solution is to constantly ask these devices if they’re ready to give you data; we refer to this process as *polling*. The reality is that if your processor is stuck polling something waiting for a response, it means your processor is not available to do other things, possibly other really important things⁵. Another way to look at this is that it is a waste of processing power. Wasting processor power is actually not a big deal unless there is some other important task that needs doing while you’re polling.

⁵ Such as restart some dude’s heart...

Instead of the processor constantly asking if an input device is ready to provide data, or an output device is ready to receive data, it's better (in terms of processing efficiency) to have the devices tell the processor when they're ready to act. Having devices communicate directly with the processor happens via the interrupt mechanism on a given MCU. When a device is ready to communicate with the MCU, we generally refer to this as the external device is "requesting service" from the MCU. We refer to this mechanism as an "interrupt" because the processor stops what it is currently doing (stops executing the code it is currently executing) in order to take care of the device requesting service. This mechanism actually switches processing to a different set of instructions when the MCU receives an interrupt. When the external device, or "peripheral"⁶ is satisfied, the MCU returns to the code it was executing before it received the interrupt. This is a topic for another chapter; we mention it here for clarity.

- 3) **Direct Memory Access (DMA):** The final type of I/O is another form of I/O that does not require an excessive amount of processing power from the processor. This type of I/O is generally associated with large data transfers between memory and peripherals (as you may have gathered from the name). The idea here is that the processor limits its involvement with transfers. The concept of DMA is relatively simple but is more complex in cases where you need to actually design the system that implements it or program the device that controls it.

We can characterize the three types of I/O by what device is in control of handling I/O. For programmed I/O, the MCU is in charge. With interrupt driven I/O, some external device is in control. With DMA, some device external to the MCU is also in control with "help" from the MCU. Which device is in control of the MCU's resources is a hot issue in the wonderful world of embedded systems.

9.3.1 RISC-V Memory Mapped I/O

The RISC-V MCU uses memory mapped I/O. This means that the RISC-V uses memory access instructions to perform I/O. More specifically, the RISC-V MCU uses load instructions to perform input and store instructions to perform output. What makes a load or store instruction into a memory access instruction is the value of the address associated with the instruction. When working with MCUs, there is always a notion of memory address space, which MCUs typically defines by providing a memory map.

Figure 9.7 shows the memory map for the RISC-V MCU. This memory map is important for both programmers and hardware people as it shows how the MCU uses the memory spaces associated with the 32-bit addresses or "address space", used by the RISC-V MCU. Figure 9.7 shows that memory addresses 0x11000000 or above are associated with the I/O. What you can also see from Figure 9.7 is the notion that the address ranges from 0x00000000 to 0x0000FFFF refer to actual memory (we'll discuss the stack, data, and code segments in another chapter). It is thus up to the programmer to utilize the address space via the load and store instructions to properly access I/O or memory. Keep in mind, the assembler acts on the memory address as written, and does not know hardware details such as the difference between memory access and I/O.

The port addresses for I/O devices have generally been setup by some hardware designer. In other words, they are a function of the underlying hardware. For any given piece of hardware that the RISC-V may be running on, the hardware designer (or someone with a similar title) needs to provide the programmers with the port addresses such that programmers can properly access external hardware, which means properly perform I/O.

⁶ Devices in digital systems are often referred to as *peripherals*. This is nothing more than saying that there is a module there that is communicating in some way with the processor.

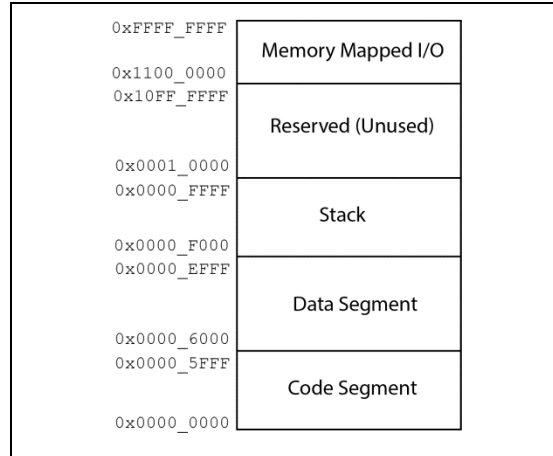


Figure 9.7: The RISC-V MCU memory map.

9.3.2 RISC-V Input & Output Instructions

This is where it starts to get really confusing. The problem is that we run into vernacular issues once we start reusing instructions for more than one purpose. What we have now is input that uses a load instruction, which is associated with memory read. Then we have output that uses a store instruction, where the store instruction is associated with a memory write. I still toil with this when I use this vernacular. Table 9.9 provides the big overview of this vernacular.

Instruction	Operation	Comment
lw (load)	Memory read	Copies data from memory to register
	input	Copies data from outside world to register
sw (store)	Memory write	Copies data from register to memory
	output	Copies data from register to outside world

Table 9.9: Overview of dual purpose load and store instructions.

We're ready to look at the actual input and output instructions. Table 9.10 shows the **lw** instruction used as an input and the **sw** instruction used as an output. Here is some more pertinent information regarding Table 9.10:

- The instructions can specify any register for source and/or destination. The registers in the example are arbitrary.
- The immediate field associated with both the **lw** and **sw** instruction limits the size of the offset. The immediate value for both instructions is limited to a 12-bit value, which the assembler interprets as a signed value.
- The base register is a 32-bit value used in the address calculation. The examples in Table 9.10 assume that the proper data is currently in the base register before hardware executes the **lw** or **sw** instructions.

Instruction	Operation	Comment
lw x10, 0(x23)	input	Inputs data from the port address calculated by adding the offset (0) plus base register (x23) into register x10.
sw x11, 0(x24)	output	Outputs data in register x24 to the port address calculated by adding the offset (0) plus the base register (x24)

Table 9.10: Overview input (load) & output (store) instructions

We're ready to show most of the instructions we've introduced into actual code. We're not yet to the point of writing actual programs, but we can write "fragments" of programs to illustrate a few points. We'll do this by way of example problems. Here is our first example program having to do with programming.

Example 9.5: Load & Input Code Fragment

Write a fragment of RISC-V assembly language code that loads a word from memory address 0x3F4 to register x21, and also inputs a word from port address 0x11008000 to register x16.

Solution: Figure 9.8 shows the solution to this example. There are several particularly important things to note about this solution:

- RISC-V assembly language uses the “#” symbol for comments; the assembler ignores everything following this symbol. There are currently no block-type comments in RISC-V.
- The code appears strikingly nice. We've aligned the instructions themselves. We've aligned the first operand for each instruction. We've aligned the comments. We included white space (blank lines) between what we feel are different types of instructions on line (4) and line (6). Everything we've done with the code takes advantage of the fact that the assembler ignores white space. You can't see it, but we also wrote this code without using tabs, using spaces to indent various items⁷.
- When we think “input”, which means input from devices external to the RISC-V MCU, we immediately think “load”, which means we need to issue some type of load instruction. Because it's input, some hardware person (or the problem specification) needs to provide us the programmer with a port address that we can use to access the input data. The problem description did in fact provide us with a port address.
- The fragment uses two **li** instructions on lines (02-03) to load values (considered immediate values) into two registers. The assembler translates the first **li** instruction on line (02) into a **lui & addi** instruction because the immediate value can't be represented using 12-bits. Because the second **li** instruction has an immediate value that we can represent using 12-bits, the assembler translates that instruction into an **addi** instruction.
- For this problem, we don't care about the values in x21 & x16 because the two **lw** instructions overwrite them.
- The form of the two **lw** instructions on line (05) and line (07) are identical not including the different register definitions. Despite looking the same, they are distinctively different. The first **lw** instruction on line (05) performs a memory read while the second **lw** instruction on line (07) performs an input operation. The difference between these two instructions is based solely on the value of the address. In the case of the first **lw** instruction, the address is 0xFFFF or less, which makes it a true memory access. In the case of the second **lw** instruction, the address is 0x11008000,

⁷ It's a bad idea to use the tab key when writing code. Different editors (such as those of someone else working on the code) and different printers interpret tabs differently. Use the spacebar for indentation and make sure your editor does not automatically insert the tabs instead of spaces.

so the instruction performs an input operation. Note that the assembler does not know the difference; the differences are actually only known to the hardware.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)     li    x15,0x11008000    # put 0x11008000 value into x15
(03)     li    x20,0x3F4        # put 0x3F4 into x20
(04)
(05)     lw    x21,0(x20)       # copy value from mem address 0x3F4 to x21
(06)
(07)     lw    x16,0(x15)       # input value from port address 0x11008000 to x16
(08) #
(09) #~~~~~ program fragment ~~~~~

```

Figure 9.8: Solution for this example.

Example 9.6: Store & Output Code Fragment

Write a fragment of RISC-V assembly language code that stores a word in register x29 into memory address 0x774; the fragment should also output the data in register x18 to port address 0x1100C000, and also inputs a word from port address 0x11008000 to register x16.

Solution: Figure 9.9 shows the solution to this example. There are several particularly important and informative things to note about this solution:

- The code is similar to the previous set of code in appearance: everything looks great standing two meters away. You can't say enough about having good-looking code, particularly when people automatically think good looking code works good⁸. This is also a fragment of code and not a complete program.
- The code first loads address values into registers using `li` instructions on line (02-03). One of the address values is greater than 0xFFFF so it is necessarily a port address (line 02). The other address is a valid memory address because the address is less than 0x00010000.
- The code on lines (02-03) falls into the category of "initialization code"; we do our best to put values into registers that we use often and leave them there without changing them. We try to do it this way because these two `li` instructions don't really do anything useful, so we want to execute them as little as possible.
- The `sw` instruction on line (05) performs a memory write operation because the address that is being "written to" is less than 0x00010000. We also use a new notation in the comment, which is an array notation of sorts, to indicate that the instruction uses the value in x11 as an index into memory.
- The `sw` instruction on line (07) performs an output operation, which means it sends the value x18 to the outside world (meaning some external device not part of the RISC-V MCU. The RISC-V hardware interprets the fact that the address is greater than 0xFFFF and essentially implements an output operation as opposed to a memory write operation.

⁸ The also automatically think that bad looking code works poorly. Write good looking code.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)     li    x10,0x1100C000    # put 0x11008000 value into x15
(03)     li    x11,0x774        # put 0x3F4 into x20
(04)
(05)     sw    x29,0(x11)       # write value x29 to mem[0x0774]
(06)
(07)     sw    x18,0(x10)       # output value in x18 to port address 0x1100C000
(08) #
(09) #~~~~~ program fragment ~~~~~

```

Figure 9.9: Solution for this example.

9.3.3 Load and Store: The Complete Story

Our main mission in this chapter is to give you a general feel for the RISC-V ISA and how “things are done” in RISC-V. If you continue in your career with computer-type stuff, you’ll probably find that although computers all roughly do the same stuff, they have a different “feel” for how they do things. When you’re learning your way around a new architecture, you definitely need to learn both items.

In order to keep things as simple as possible, we’ve up to this point only presented a subset of the load and store instructions in the RISC-V instruction set. The idea behind the other versions of the instructions is the same, so we’ll quickly describe these new instructions in this section. Keep in mind that load and store instructions comprise of eight of the 40 or so instructions in the RISC-V ISA, which underscores the notion that there are many similarities between instructions in the ISA.

Table 9.11 shows the complete set of load and store instructions in the RISC-V ISA. These instructions differ in several different ways, which we of course list below. The main “idea” behind the load and store instructions and their relation to memory and I/O remains the same as our previous discussion.

- 1) The load and store instructions operate on three different sizes of data. The main memory in the RISC-V MCU is byte addressable, which means a byte is the smallest size of data we can access in main memory. Although the main memory is byte addressable, memory access instructions can also access halfwords (two bytes) or words (four bytes) with a single instruction⁹.
- 2) When we issue a store instruction, the instruction causes the main memory to deal with the required width of the instruction based on the exact instruction (**sw** vs. **sh** vs. **sb**). In other words, when you store a value from a register, the main memory in hardware only store the proper amount of data from the register, which is the lower byte for the **sb** instructions, the lower two bytes for the **sh** instruction, or the entire register contents for the **sw** instruction.
- 3) Loading words into register from memory is different from storing words. When you store a word, you always for from a register (or a known part of a register) to memory, so there are no “extra bytes” to worry but. When you load a halfword or a byte from memory into a register, there is a question of what to do with the extra bytes. For example, when you load a byte (8 bits) in to a 32-bit register, where do not place that byte and what do you do with the bytes in the register that you don’t have data for?

First, the data from memory always fills the right-most bytes in the register, which means when you load a byte, the hardware places it into the right-most of the four bytes in the register. Second, what happens to the extra bits when you load a byte or halfword? The answer depends on which instruction you use. There are two types of load instructions for loading data lengths other than words, which are **lb** & **lbu**, and the **lh** & **lhu** pairs. The difference, for example, between the **lb** and **lbu** is what the hardware does with the unused bytes. For the **lb** instruction, the hardware considers the byte a signed value and then sign extends the three unspecified bytes, which means copying the sign bit of the byte into all the other 24 bits in the register. For the **lbu** instruction, the

⁹ We use the word access to mean both reading and writing to memory.

hardware considers the byte to be an unsigned value and zero-extends the value to fill the register, which means it placed 24 0's into the unused bits¹⁰.

Load			
Instruction		Example	Comment
lb	rd, imm(rs1)	lb x8, 0(x11)	Loads byte into x8; upper 3 bytes are sign extension of byte; Memory address = 0 + value in x11
lbu	rd, imm(rs1)	lbu x7, 14(x23)	Loads byte into x7; upper 3 bytes are zero extension of byte; Memory address = 14 + value in x23
lh	rd, imm(rs1)	lh x8, 4(x21)	Loads 2-bytes into x8; upper 2 bytes are sign extension of half word; Memory address = 4 + value in x21
lhu	rd, imm(rs1)	lhu x6, 2(x23)	Loads 2-bytes into x6; upper 2 bytes are zero extension of half word; Memory address = 2 + value in x23
lw	rd, imm(rs1)	lw x7, -4(x22)	Loads 4-bytes into x7; Memory address = -4 + value in x22
Store			
sb	rs2, imm(rs1)	sb x5, 3(x6)	Stores right-most byte of x5 into memory address = 3 + value in x6
sh	rs2, imm(rs1)	sh x4, 34(x7)	Stores lower 2 bytes of x4 into memory address = 34 + value in x7
sw	rs2, imm(rs1)	sw x5, 0(x8)	Stores contents of x5 into memory address = 0 + value in x8

Table 9.11: Overview of the complete set of RISC-V load & store instruction.

Example 9.7: Loading and Storing with Different Data Sizes

Write a fragment of RISC-V assembly language copies the word value at the address given in x10 to four registers starting at x20. Each register should receive one byte of the word data at address x10. Consider the bytes of the word value to be unsigned values. Don't use any shift-type instructions in your solution.

Solution: Figure 9.10 shows the solution to this example. Yet another expressive example; here's the stuff that allow you to impress your friends at parties:

- The problem states that you need to take the individual bytes from memory and store them as bytes in consecutive registers starting at x20. There are several ways to do this but we'll use the approach that leverages the different versions of memory access instructions.
- The problem states to divide up a word in memory into byte in registers. The problem also states that we want the bytes in the registers to be unsigned. We have five different flavors of load instructions, including one for words (**lw**), halfwords (**lh** & **lhu**), and bytes (**lb** & **lbu**).
- Our approach for this solution is to issue four **lbu** instructions, which we do on lines (02-05). Note that we use the same base address for each of the **lbu** instructions (x10), but we increment the offset portion of the instruction by '1' with each instruction. This advances the address one byte greater than the base address in x10. Recall that the memory is byte addressable, which is why we increment the offset by '1' with each instruction.

¹⁰ The same stuff happens for the **lhu** and **lh** instructions, but we won't bore you with the details again.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02)     lbu   x20,0(x10)   # left-most byte of word goes into x20
(03)     lbu   x21,1(x10)   #
(04)     lbu   x22,2(x10)   #
(05)     lbu   x23,3(x10)   # right-most byte of word goes into x23
(06) #
(07) #~~~~~ program fragment ~~~~~

```

Figure 9.10: Solution for this example.

9.3.3.1 Load & Store Instructions Relation to I/O Data Widths

When dealing with I/O, the widths of the input and output data become somewhat of an issue. The issue is the fact that the RISC-V load and store instructions handle only a limited range of data widths, namely words, halfwords, and bytes. The notion of I/O is that we need to communicate with peripherals outside of the RISC-V MCU, which generally means we need to deal with the peripherals on their terms. For example, if you're RISC-V MCU is driving 12 LEDs (thus an output), we need 12 bits to control those LEDs. Note that 12 bits is bigger than a byte and smaller than a halfword.

The first thing to note is that you need to deal with I/O according to the configuration of the hardware you are working with. For example, if you had to drive 12 LEDs, designer can configure the hardware many different ways, though two of the ways represent the most common approach. The following is a description of these two approaches; you'll want to compare and contrast these to get a feel for how to properly utilize such outputs.

- 1) Configure the hardware to associate the 12 signals driving the LEDs with a single port address. In this way, driving the LEDs with an output instruction (a store) would require that you use a **sb** instruction at the very least. In this way, you could drive all 12 LEDs with one output instruction.
- 2) Configure the hardware to associate eight LEDs with one output port address and four LEDs with a different output port address. In this way, driving all 12 LEDs would require issuing at least two output instructions. The best approach in this case would be to issue two **sb** instructions.

One issue you that you may need to consider with the second option is what happens to the entire unused bit when where there is a size mismatch between the data width offered by the instruction (byte, halfword, word) and the width of the actual output. In this case, you can probably assume that someone has configured the hardware such that you can issue any instruction with a larger width and it would work. For example, when you issue a **sb** (store byte) instruction to drive four LEDs, what happens to the missing LEDs? This is actually more of an issue when inputting data of widths that don't match the instruction widths. In these cases, you definitely need to be aware of how the associated hardware configuration. For example, when you input four bits using an **lbu** instruction, you'll get a register filled with 32 bits, and you can probably be sure the lower four bits are the data you're trying to input, but what is the other data.

In the end, you hope someone has both configured the hardware in an intelligent manner, and that they let you the programmer know how they configured that hardware. As a programmer, you should try to match data widths with your I/O instructions the best you can even though it may not matter. For example, if you're inputting five bit, issue a **lb** or **lbu** instruction, even though the hardware may do the same thing using an **lh**, **lhu**, or **lw** instruction.

9.4 The First Program Flow Control Instruction

Our working definition of a computer was a digital device that sequentially executes a programmed stored in memory. We'll get into more details later, but what this generally means is that the hardware executes an instruction stored in memory, then executes the next instruction stored in memory, etc. Note the "sequentialness" of instruction execution in this definition. Programs would quickly run out of instructions if sequential execution of instructions were all that the computer could do. In reality, computer hardware "can be directed" to execute any

instruction in memory. The truth is that some instructions have the ability to direct the computer hardware to execute an instruction that is not necessarily the next instruction in program memory.

The notion of the next instruction that the computer executes falls under the topic of program flow control. We refer to instructions that have the ability to direct instruction execution to an instruction other than the next instruction as program flow control instruction. In RISC-V computer lingo, there are two types program flow control instructions: jumps and branches¹¹. Both of these instruction have the ability to send *program flow* to somewhere other than the next instructions; the difference between these instructions is that jump instructions always cause a change in program flow while branch instructions can cause a change in program flow, but only under certain conditions. The RISC-V vernacular here is that jump-type instructions cause a change in program flow control unconditionally while branch instruction conditionally cause a change in program flow control.

This section briefly introduces a jump-type instruction, which is the final instruction we need to start writing actual RISC-V assembly language programs. A full description of RISC-V program flow control instructions appears in Section 10.3.

9.4.1 Introduction to Program Flow Control

The only way to stop a RISC-V assembly language program from running (once you start it) is to turn off the power. You'll find that there is not "stop" or "halt" instruction in the RISC-V instruction set. The intent of many computer programs is to always run, which generally means to keep monitoring input and waiting for an indication that the computer needs to do some task based on that input¹². To successfully keep a given program running (or executing in a meaningful way), the program must somehow direct program flow from the last instruction in the program to some other instruction in the program. We use program flow control instructions to accomplish this redirection.

The most simple program flow control instruction is the `j` pseudoinstruction. This instruction translates to a `jal` instruction, but we'll save the underlying details for another section. What we're interested in at this time is a simple unconditional branch instruction so we can start writing complete programs. Table 9.12 shows the details of the `j` pseudoinstruction; we'll quickly use this in a simple program to explain its actual usage.

Instruction Form	Equivalent Base Instruction(s)	Example Usage	Comment
<code>j label</code>	<code>jal x0, label</code>	<code>j label</code>	Jump to instruction associated with label

Table 9.12: The basic unconditional branch pseudoinstruction.

Example 9.8: Our First RISC-V Assembly Language Program

Write a RISC-V assembly language program that continuously reads data from port address 0x11003300 and outputs the data to port address 0x11005500.

Solution: Figure 9.11 shows the solution to this example. Since this is our first complete program, we'll describe it in a painful amount of detail:

- This problem is nice in that it contains the three main parts of an assembly language program (four parts if you include labels) 1) comments, 2) directives, and 3) instructions.
- There is nothing special about the port addresses called out by the problem other than the fact that they are 32-bits values. The RISC-V address space is 32 bits with requires we always provide port addresses as 32-bit values.

¹¹ As you'll see later, subroutine calls and return from subroutines in RISC-V are both a type of jump instruction.

¹² This is the classic embedded systems model.

- The choice of registers x10, x11, and x20 in the solution was arbitrary; we could have used other registers instead.
- The program starts with an informative file header (or *file banner*) that describes the purpose of the program. Always include file headers; we sometimes don't include them in this text as a space-saving manner. The comments on lines (00-03) represent the file banner.
- An assembler directive appears on line (04). This is the “.text” assembler directive that roughly indicates the text that follows are all instructions. We'll deal more with assembler directives and memory segmentation in a later chapter.
- All assembly language programs require some of initialization code at the start of the program. Line (06) represents the start of the initialization code. Note that we use an “init” label to indicate that the code that follows is some type of initialization code. The program does not use the “init” label in any way; it serves only to indicate to human readers of the code the general purpose of that section of code. This label, as with all labels, does not increase the size of the program eventually stored in program memory.
- The purpose of the initialization code is to place the port addresses into a register, which we must do because the I/O instructions in RISC-V use registers to generate absolute memory address. We use the `li` (load immediate) instruction on lines (06 & 07) to put the I/O port addresses into memory.
- The `lw` instruction on line (09) inputs the data from the input port to register x20. The `sw` instruction on line (10) outputs the data in register x20 to the output port.
- The `j` instruction on line (11) is the program flow instruction. This instruction directs program execution to some other executable instruction in the program. Note that no instructions follow the `j` instruction, so there is no “following” instruction program execution can sequentially flow to. The only possibility is to direct program flow to some other instruction in the program. The fact that the argument of the `j` instruction is “main” directs the program flow back to the instruction following the “main” label. We cover the details of exactly how the computer does this in a later chapter. Thus, the `j` instruction essentially ensures that the program never runs out of instructions to execute by directing program flow to some other valid instruction in the program. Wildly exciting!

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x11003300 and output that data to port address 0x11005500.
(03) #-----
(04) .text
(05)
(06) init:  li    x10,0x11003300    # input port address
(07)         li    x11,0x11005500    # output port address
(08)
(09) main:  lw     x20,0(x10)        # input data
(10)         sw     x20,0(x11)        # output data
(11)         j     main              # repeat I/O sequence

```

Figure 9.11: Solution for this example.

Figure 9.12 shows a flowchart that models the operation of this program; here are a few things to note about this amazing flowchart:

- The flowchart shows the basic flow of the program without providing any assembly language specific details. Making flowcharts generic in this way makes them arguably more maintainable. For example, if the flowchart was specific to port addresses and registers associated with the RISC-V OTTER MCU, the flowchart would be harder to read and thus less usable if the hardware changed.
- The program contains no conditional branch instructions so the flowchart does not contain any decision symbols.

- The flowchart almost has a process block for each instruction, but note there are no process block for the unconditional jump instruction on line (11). Because the branch instruction (jump) in unconditional, we represent it with flow lines.

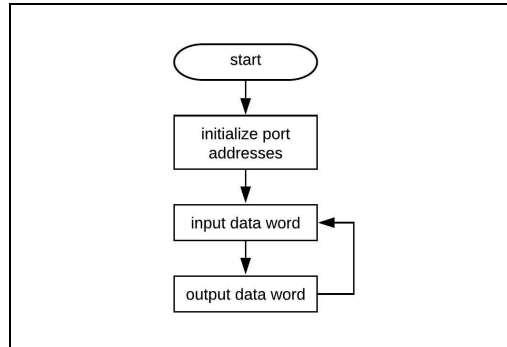


Figure 9.12: A flowchart modeling the operation of this example program.

Example 9.9: Input, Modify, & Output Data

Write a RISC-V assembly language program that continuously reads data from port address 0x1100DD00, adds 47 to that data, then outputs the data to port address 0x1100DF00. Don't worry about overflow in the addition instruction.

Solution: Figure 9.13 shows the solution to this example. Since this is our second complete program, we'll opt not to repeat the level of detail from the first program; here are the main differences to make yourself aware of:

- What we need to do in this program is modify each piece of input data before we output it. We add 47 to the input data on line (11) as the problems requests. We use the `addi` instruction to do this because we are adding a constant value to the input value. The instruction adds 47 to the value in `x20` and then stores the result in `x20`. In this case, the value in `x20` is always modified.

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x1100DD00, adds the value 47 to that data, and then outputs
(03) # that data to port address 0x1100DF00.
(04) #-----
(05) .text
(06)
(07) init:  li    x10,0x1100DD00    # input port address
(08)      li    x11,0x1100DF00    # output port address
(09)
(10) main:  lw     x20,0(x10)       # input data
(11)      addi  x20,x20,47         # add 47 (an immediate value) to the data
(12)      sw     x20,0(x11)       # output data
(13)
(14)      j     main              # repeat I/O sequence
  
```

Figure 9.13: Solution for this example.

There are a few ways to do this problem in real life, but to do so, we must know more instructions. The `addi` in this chapter added an immediate value to a register and stored the sum in another register. There is also an `add` instruction in the RISC-V instruction set that adds values from two registers and stores that value in another register. Figure 9.14 shows an alternative solution to this example that uses an `add` instruction; here are some worthy comments regarding that solution, noting that most of this solution is similar to the previous solution.

- Since we'll be using an **add** instruction to add 47 to the input value, we first must put 47 into a register; we do this with an **li** instruction on line (08). We need to do this because the **add** instruction is a register/register instruction; the **addi** instruction was a register/immediate instruction.
- We use the **add** instruction rather than the **addi** instruction on line (12). The **addi** instruction required us to always add a constant value to the input data; using the **add** instruction with the extra register operand allows us to effectively add a variable value to the input data, which we could do by changing the value in x15 somewhere in the program after it has been initialized.

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x1100DD00, adds the value 47 to that data, and then outputs
(03) # that data to port address 0x1100DF00.
(04) #-----
(05) .text
(06)
(07) init:  li    x10,0x1100DD00    # input port address
(08)      li    x11,0x1100DF00    # output port address
(09)      li    x15,47            # place the value 47 in a register
(10)
(11) main:  lw    x20,0(x10)       # input data
(12)      add   x20,x20,x15       # add 47 (from register) to the data
(13)      sw    x20,0(x11)       # output data
(14)
(15)      j     main              # repeat I/O sequence

```

Figure 9.14: An alternative solution for this example.

Example 9.10: Input Multiple Data, Modify, & Output Data

Write a RISC-V assembly language program that continuously does the following: reads data from port address 0x1100CC00 two times (two different pieces of data), adds that data from those two inputs together, then outputs the data to port address 0x1100EE00. Use an **add** instruction rather than an **addi** instruction in your solution. Don't worry about overflow in the addition operation.

Solution: Figure 9.15 shows the solution to this example. Since this solution is similar to previous solutions, we'll only describe the significant differences:

- The main difference in this problem is that we need to read two pieces of data and sum that data before we output it. We read the data from the same input port, but we need to place it in two different registers, which we do on lines (10-11).
- We add the data using a register/register-type **add** instruction on line (12). This instruction adds the values in x10 and x11 and then stores the sum back into x10. Though it may seem like we're reusing registers in this instruction, this is typically the way we do it. We could have stored the sum in a different register, but the way we did it in this examples saves us from reusing another register. Having 32 registers sounds like a lot, but unused registers become scarce when coding complex algorithms.


```

(00) #-----
(01) # Program Description: The program continuously does the following: reads
(02) # data from an input two times, sums the two input values, then outputs the
(03) # data to an output port.
(04) #-----
(05) .text
(06)
(07) init:  li    x20,0x1100CC00    # input port address
(08)        li    x21,0x1100EE00    # output port address
(09)
(10) main:  lw    x10,0(x20)         # input first piece data
(11)        lw    x11,0(x20)         # input second piece of data
(12)        add   x10,x10,x11        # sum two input value to register x10
(13)        sw    x10,0(x21)         # output data
(14)
(15)        j     main              # rinse, repeat

```

Figure 9.15: Solution for this example.

Figure 9.16 shows a flowchart modeling the operation of this program. This flowchart resembles previous flowcharts so we'll omit any extra verbage for this problem.

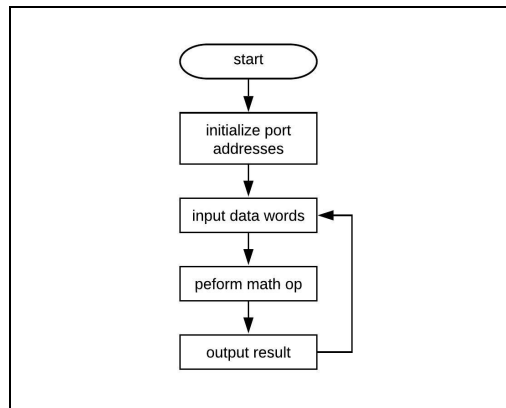


Figure 9.16: A flowchart modeling the operation of this example program.

9.5 Chapter Summary

- Transferring data between storage elements is probably the most common operation in microcontrollers. We can group the RISC-V instructions based on where they transfer data to and from.
 - **mv**: transfers data from register to register
 - **li**: transfers data from an immediate value to a register
 - Load-type instructions (**lw**, **lh**, **lhu**, **lb**, **lbu**): transfer data from memory to register (memory access) or from external devices to register
 - Store-type instructions (**sw**, **sh**, **sb**): transfers data from register to memory (memory access) or from register to external devices.
 - Input/Output operations are what make computers useful. The three main type of I/O are programmed I/O, Direct Memory Access (DMA), and interrupt driven I/O. The two main types of programmed I/O are *port mapped* and *memory mapped I/O* (MMIO)
 - The RISC-V MCU uses memory-mapped I/O, which means that load and store instructions (memory access) also perform I/O operations. The RISC-V assembler does not know whether a particular load or store instruction will perform a memory access or an I/O operation; the underlying RISC-V hardware implements the correct instructions based on the effective address of the load and store instruction. If the effective address is within a range specified by the hardware designer, the load/store operation is an I/O; otherwise, the operation is a memory access.
 - Load-type instructions: used for memory reads and input
 - Store-type instructions: used for memory writes and output
-

9.6 Chapter Exercises

- 1) What is the difference between pseudoinstructions and base instructions?
 - 2) All data crunching instructions involve which particular module of the RISC-V architecture. Hint: this module is part of the programmers model.
 - 3) Briefly describe the primary difference between the `mv` and `li` instructions.
 - 4) Name and briefly describe the three types of I/O used by computers.
 - 5) `li` is a pseudoinstruction that the assembler translates to either one or two base instruction. Briefly describe what determines how many base instructions the compiler will use.
 - 6) Briefly describe why there are signed and unsigned load-type instructions but not signed and unsigned store-type instructions.
 - 7) Briefly explain how pseudoinstructions are converted to base instructions.
 - 8) Name and briefly describe the two types of programmed I/O.
 - 9) Who or what decides whether a particular computer architecture will use memory mapped I/O or port mapped I/O.
 - 10) For any given MCU-based circuit, who is responsible for “setting up” the port addresses?
 - 11) Briefly explain if there is any way for a programmer who knows nothing about hardware to discern port addresses without being told directly?
 - 12) Briefly explain why instruction sets such as the RISC-V instruction set have no need for “halt” or “stop” instructions.
-

9.7 Chapter Programming Problems

For the following problems:

- Minimize the amount of instructions in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code
-

- 1) Write a RISC-V assembly language program that continuously reads a word of data from port address 0x1100A000 and outputs that data to port address 0x1100B000.
 - 2) Write a RISC-V assembly language program that continuously reads a word of data from port address 0x1100C000, adds the value 0x434 to the input value, and then outputs the data to port address 0x1100D000.
 - 3) Write a RISC-V assembly language program that continuously reads a word of data from port address 0x11000020, adds the value -45 to the input value, and then outputs the result to port address 0x11000030. Don't worry about overflow (or underflow) from the addition operation.
 - 4) Write a RISC-V assembly language program that continuously reads signed halfword data from port address 0x1100C000, doubles that input value, and then outputs the result to port address 0x1100D000. Don't worry about overflow from the mathematical operation.
 - 5) Write a RISC-V assembly language program that continuously reads unsigned halfword data from port address 0x1100FF00 two times, doubles each of the two input values, sums the results of the doubling operation, and then outputs the result as a word to port address 0x1100EE00. Don't worry about overflow from the addition operation.
 - 6) Write a RISC-V assembly language program that continuously reads unsigned byte data from port address 0x1100AA00 three times, sums those input values, then outputs the result as an unsigned word to port address 0x1100AB00. Don't worry about overflow from the addition operation.
 - 7) Write a RISC-V assembly language program that continuously reads word data from port address 0x11001000 two times, multiplies each of those values by three, sums the results, and then outputs the final result to port address 0x11002000. Don't worry about overflow from the addition operation.
 - 8) Write a RISC-V assembly language program that continuously reads word data from port address 0x1100DDD0 eight times, sums the inputs and then outputs the result to port address 0x1100EEE0. Don't worry about overflow from the addition operation.
 - 9) Write a RISC-V assembly language program that continuously reads signed byte data from port address 0x11001111 four times; the program subtracts one from the first input value, two from the second input value, three from the third input value, and four from the fourth input value, sums the results, and outputs the result as a signed halfword to port address 0x11002222. Don't worry about overflow (or underflow) from the addition operations.
 - 10) Write a RISC-V assembly language program that continuously reads signed halfword data from port address 0x11001000 two times, multiplies each of those values by two, and outputs the two results as words to port addresses 0x11002000 and 0x11002002, respectively. Don't worry about overflow from the addition operation.
-

10 Instructions, Constructs, and Bit-Level Manipulations

10.1 Introduction

We can model assembly language programming as an exercise in pulling “things” out of an assembly language “bag of tricks” in a structured manner in able to solve our given problem. The notion here is that there is generally not that much you can do with assembly languages compared to higher-level languages. My feeling is that the most complicated part of learning to program in an assembly language is learning and keeping track of the various “tricks”. These so-called tricks, are not really tricks; they’re actually simple operations that would take you extra time to be aware of if someone did not point them out to you.

Assembly language programs are simple because they are inherently limited in their ability to do things. The result of this simplicity is that programs use the same programming constructs and instructions to do the same type of operations repeatedly. The good news is that there only a relative few constructs and they’re all relatively simple. Everything about assembly language is simple; assembly language programming only seems hard because there are initially so many new things to learn. This chapter describes some of the more important considerations programmers should be aware of when writing robust assembly language code.

Main Chapter Topics

- **BIT CRUNCHING INSTRUCTIONS:** This chapter describes the remaining instructions that “crunch” bits including logic, arithmetic, and shift-type instructions.
- **PROGRAM FLOW INSTRUCTIONS:** This chapter describes program flow instructions including conditional and unconditional branch instructions.
- **ITERATIVE CONSTRUCT ISSUES:** This chapter describes some of the important underlying issues regarding iterative constructs.
- **MANIPULATING BITS:** This chapter describes the common bit manipulations found in assembly language programming known as bit masking.
- **AUXILIARY INSTRUCTIONS:** This chapter describes a few other useful instructions and pseudoinstructions that are hard to easily classify with other instruction types.

Why This Chapter is Important

This chapter is important because it describes some of the basic programming concepts and approaches beyond simple description of individual instructions.

10.2 Bit Crunching Instructions

The RISC-V ISA contains a set of instructions that we can describe as bit crunching. These instructions primarily change the value in the destination operand based on a given operation between source operands. There is always one destination operand and that operand is always a register. There are always two source operands, one of them is always a register, and the other operand can either be a register or an immediate value. Many of the bit crunching operands are similar in that the instruction set uses two different instructions to perform the same operation but on two sources register operands or one source register operand and one immediate value operand.

Important to note here is that the RISC-V MCU instruction set does not directly provide the ability to test the validity of bit-crunching operations. This is because the underlying hardware does not provide any type of status signal regarding the result of any given operation. Because of this, programmers are required to use the flexibility of the instruction set in order to determine items such as when an ALU operation overflows the 32-bit register width.

10.2.1 Logic Instructions: AND, OR, & XOR

We grouped these instructions together because they are all logic-based instructions. All of these instructions perform what we refer to as *bit-wise* logic operations on their operands. This is a common notion in computerland; it simply means that given logic operator performs the logic operation on the corresponding individual bits of the two operands.

Table 10.1 shows the two forms of the instructions implementing AND, OR, and EXOR operations. Their instruction type differentiates the two forms, where the two-register operand form is an “R-Type” instruction and the one register one immediate value form is an “I-Type” instruction. These are differentiated by the “i” postfix on the instruction mnemonic; “R” roughly stand for register and “I” roughly stands for immediate. Both instruction types perform the given operation on the individual bits in the two operands and store the results in the designated destination register. These instructions do not alter either source operand.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
R-Type	<code>and rd,rs1,rs2</code>	$rd \leftarrow rs1 \cdot rs2$	<code>and x10,x20,x21</code>	32-bit operation
I-Type	<code>andi rd,rs1,imm</code>	$rd \leftarrow rs1 \cdot imm$	<code>andi x8,x8,0xF0</code>	imm max: 12-bits
R-Type	<code>or rd,rs1,rs2</code>	$rd \leftarrow rs1 + rs2$	<code>or x15,x15,x16</code>	32-bit operation
I-Type	<code>ori rd,rs1,imm</code>	$rd \leftarrow rs1 + imm$	<code>ori x7,x8,255</code>	imm max: 12-bits
R-Type	<code>xor rd,rs1,rs2</code>	$rd \leftarrow rs1 \wedge rs2$	<code>xor x30,x28,x8</code>	32-bit operation
I-Type	<code>xori rd,rs1,imm</code>	$rd \leftarrow rs1 \wedge imm$	<code>xori x10,x9,0x44</code>	imm max: 12-bits

Table 10.1: The two forms associated with the four logic instructions.

Figure 10.1 shows a well-commented code fragment that demonstrates the use of both forms of the logic-based instructions. While most of these instructions are straightforward in the sense that they are performing bitwise logic operations that should be familiar to you, you should take special note of a few items:

- We initialize few registers with known values on lines (01-03) so we can use them in the instruction examples that follow. We also included a label on line (01) which acts as a comment to indicate that we’re performing some type of initialization, in this case, of a few register values.
- The immediate version of each instruction has limitations on the size of the immediate value. For these instructions, that limit is 12 bits. The RISC-V hardware is responsible for sign-extending each 12-bit immediate value to form a 32-bit value so that the operation becomes a true bitwise operation (meaning both source operands are 32 bits when the hardware does the logic operation). The register-register version of these operations does not require any modification of the source operands.
- As implied by the previous bullet, the transformation of an immediate value to a 32-bit value is typically a two-step process. The assembler does the first step by converting the immediate operand appearing as part of the instruction in to a 12-bit signed number; this value is subsequently stored as part of the instruction. The RISC-V hardware performs the second step in this conversion by sign-extending the 12-bit immediate value stored as part of the instruction to

¹ Could not find proper XOR symbol in my editor, so I’m opting to use the “^”, which is the XOR operator in the C programming language.

form a 32-bit value. At this point, the RISC-V hardware can now implement a true bitwise operation.

(00)	#~~~~~ program fragment ~~~~~
(01)	init: li x10,0xFFFF0A8B # Initialize three registers
(02)	li x11,0xF000000F #
(03)	li x12,0x0F0000FF #
(04)	
(05)	and x20,x10,x11 # Op: x10 AND x11 (0xFFFF0A8B AND 0xF000000F)
(06)	# Result: x20=F000000B; x10 & x20: no change
(07)	
(08)	andi x21,x10,0xFF # Op: x10 AND 0xFF (0xFFFF0A8B AND 0x000000FF)
(09)	# Result: x21=F000008B; x10: no change
(10)	
(11)	or x22,x10,x12 # Op: x10 OR x12 (0xFFFF0A8B AND 0x0F0000FF)
(12)	# Result: x22=FFFF0AFF; x10 & x12: no change
(13)	
(14)	ori x23,x10,1023 # Op: x10 OR 0xFF (0xFFFF0A8B AND 0x000003FF)
(15)	# Result: x23=FFFF0BFF; x10: no change
(16)	
(17)	xor x24,x10,x12 # Op: x10 XOR x12 (0xFFFF0A8B AND 0x0F0000FF)
(18)	# Result: x24=FFFF0AFF; x10 & x12: no change
(19)	
(20)	xori x25,x10,0x3FF # Op: x10 XOR 0xF (0xFFFF0A8B AND 0x000003FF)
(21)	# Result: x25=FFFF0574; x10: no change
(22)	
(23)	#~~~~~ program fragment ~~~~~

Figure 10.1: Usage examples for register and immediate forms of the logic instructions.

Example 10.1: Crunching Input Data

Write a RISC-V assembly language program that continuously inputs data from port address 0x1100CC00, performs a 1's complement on that data, then outputs the result to port address 0x1100EE00.

Solution: Figure 10.2 shows the solution to this example. Since this solution is similar to previous solutions, we'll only describe the significant differences:

- As with previous problems, the program has a great file banner (lines (00-04)) and a declaration of the text segment using the “.text” directive on line (05). Also similar to previous example, we must place the stated input and output port addresses into register, which we do on lines (07-08).
- We know we need to complement all the bits in the input value, so we look for a “complement” instruction in the RISC-V instruction set. We of course don't find one. Our approach is using what we have in the instruction set, which is the xor instruction. Recall that if we XOR a bit with a 1, the result is to toggle (or complement) that bit. Therefore, to perform a bit-wise 1's complement, we must use the xor instruction, which we do on line (12).
- You may be wondering why we did not simply use the `xori` instruction, such as something like this: `“xori x10,x10,0xFFFFFFFF”`. The problem is that the size of the immediate value is limited to 12 bits in immediate-type instructions. Had we tried to use the `xori` instruction in this form, the assembler would have declared that an error. The only solution we had is to put the 32-bit value into a register prior to the `xor` instruction, which we do on line (09). We effectively pre-loaded the register with the value we needed for the 32-bit complement. This works nicely, but it does have the drawback of “reserving” a register, which means we can't use that register for anything else in a given program.

```

(00) #-----
(01) # Program Description: The program continuously does the following: reads
(02) # data from an input port, does a 1's complement on that data, then outputs
(03) # the result to the output port.
(04) #-----
(05) .text # declare text segment
(06)
(07) init:  li    x20,0x1100CC00 # input port address
(08)         li    x21,0x1100EE00 # output port address
(09)         li    x22,0xFFFFFFFF # set all bits in x22
(10)
(11) main:  lw    x10,0(x20)      # input data
(12)         xor   x10,x10,x22   # complement all bits in input
(13)         sw    x10,0(x21)    # output result data
(14)
(15)         j     main          # rinse, repeat

```

Figure 10.2: Solution for this example.

Example 10.2: Crunching Input Data

Write a RISC-V assembly language program that continuously inputs data from port address 0x1100CC00, performs a 1's complement on the lower eight bits of the input data, then outputs the result to port address 0x1100EE00.

Solution: Figure 10.4 shows a possible solution for this example. We purposely used the previous solution as a starting point for this solution to show that you can solve this problem two ways. You'll see that one way is more efficient than the other way.

- This solution is the same as the previous solution; the only different is that we modified the li instruction on line (09) to load the register with 8 1's rather than 32 1's. The xor instruction on line (12) uses this new data in x22. This works, but savvy RISC-V assembly language programmers know they can write the program to perform the same operation but use less instructions.

```

(00) #-----
(01) # Program Description: The program continuously does the following: reads
(02) # data from an input port, does a 1's complement on the lower 8 bits of the
(03) # input data, then outputs the result to the output port.
(04) #-----
(05) .text # declare text segment
(06)
(07) init:  li    x20,0x1100CC00 # input port address
(08)         li    x21,0x1100EE00 # output port address
(09)         li    x22,0xFF      # set lower eight bits in x22
(10)
(11) main:  lw    x10,0(x20)      # input data
(12)         xor   x10,x10,x22   # complement the lower eight of input
(13)         sw    x10,0(x21)    # output result data
(14)
(15)         j     main          # rinse, repeat

```

Figure 10.3: A less efficient solution for this example.

Figure 10.4 shows the better and preferred solution for this example. This solution is better because it uses less instruction than the previous solution. It also officially uses one less register; using as few registers as possible is generally good practice in assembly language programming.

- Because we only need to toggle the lower eight bits in the input value, we can use an **xori** instruction rather than an **xor** instruction. Recall that immediate-type instructions limit the size of the immediate value to 12 bits. This means that we don't need to preload a register with a 32 value, which makes the program functionally equivalent to the previous program but more space efficient (uses one less instruction).

```

(00) #-----
(01) # Program Description: The program continuously does the following: reads
(02) # data from an input port, does a 1's complement on the lower 8 bits of the
(03) # input data, then outputs the result to the output port.
(04) #-----
(05) .text # declare text segment
(06)
(07) init:  li    x20,0x1100CC00 # input port address
(08)        li    x21,0x1100EE00 # output port address
(09)
(10) main:  lw    x10,0(x20)      # input data
(11)        xori  x10,x10,0xFF  # complement the lower 8 bit of input
(12)        sw    x10,0(x21)    # output result data
(13)
(14)        j     main          # rinse, repeat

```

Figure 10.4: The preferred solution for this example.

10.2.2 Arithmetic Instructions: Addition & Subtraction

The arithmetic-type instructions perform the basic mathematical operations of addition and subtraction. Like many simple MCUs, the RISC-V MCU only has a bare minimum of arithmetic instructions, namely addition and subtraction. If you need to do more complex math such as multiplication and division, you need to use the addition and subtraction instructions to do so². Recall that we are using the RISC-V OTTER as a microcontroller (MCU), meaning that the main purpose of the RISC-V is to “control” things. The RISC-V OTTER can do some math, but complex mathematical operations are something it does not do efficiently³. The RISC-V OTTER MCU contains three mathematical instructions: **add**, **addi**, and **sub**.

Before we continue, we must mention an underlying characteristic of the RISC-V MCU. When our intention is to crunch bits, we need to sometimes consider the “meaning” of the bits we’re crunching. Some RISC-V instructions perform operations that treat the source operands as unsigned numbers; other instructions treat operands as signed numbers. When working with any RISC-V instruction, particularly the arithmetic instructions, the programmer needs to be aware of the default operation of the instruction based on its treatment of signed and unsigned numbers. Sometimes the operations are obvious, and sometimes not. The savvy programmer always checks the details in the assembly language manual to ensure they are using instructions properly.

The **add**, **addi**, and **sub** instructions perform operations as if the values of the operands are unsigned. If you need to work with any other form of numbers, such as radix complement (2’s complement), you need to work out the details in your program’s code. Keep in mind that each MCU does things differently, so you’ll always want to check the spec before you start programming. Note that the RISC-V instruction set contains register and immediate forms of the addition instruction, but only a register form of the subtraction instruction.

² The notion here is that multiplication is repeated addition and division is repeated subtraction.

³ Keep in mind we’re using a specific version of the RISC-V MCU; other versions include some complex instructions, such as instructions that deal with floating point numbers as well as other instructions that perform division and multiplication.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
R-Type	add rd,rs1,rs2	$rd \leftarrow rs1 + rs2$	add x11,x21,x31	32-bit operation
I-Type	addi rd,rs1,imm	$rd \leftarrow rs1 + imm$	addi x7,x8,0x0F	imm max: 12-bits
R-Type	sub rd,rs1,rs2	$rd \leftarrow rs1 - rs2$	sub x15,x14,x17	32-bit operation

Table 10.2: The two forms associated with the four logic instructions.

Figure 10.5 shows yet another well-commented code fragment that demonstrates the use of both forms of the arithmetic-type instructions. While these instructions do what the mnemonic implies, there are some underlying details that programmers must be aware of so they can write working programs. Here are a few things to note about the code in Figure 10.5.

- We initialize few registers with known values on lines (01-03) so we can use them in the examples that follow.
- The first example on line (06) is a reg-reg addition, with no surprise in the results.
- The second example on line (09) is the reg-imm version of the addition instruction. Here we list the immediate value as -0xFF. The assembler converts this value to a two's complement format and then sign-extends it to 0xF01. It does this to fit in the 12-bit immediate field associated with this type of immediate instruction. Before the instruction executes in hardware, the hardware sign extends 0xF01 again to become a 32-bit value: 0xFFFFF01.
- The third example on line (12) shows a subtraction operation. The second operand is all 1's, which is either a big number or a -1 depending on how you interpret the format. The hardware does no interpretation of the format and does the subtraction as listed. The instruction subtracts a negative number, which is effectively addition; the result reflects this notion. In this case, the hardware performs the required 2's complement operation before it adds the second operand.
- The fourth example on line (15) is another subtraction. This time we are subtracting a positive number, so the operation requires no 2's complement conversion of the second operand.

```

(00) #~~~~~ program fragment ~~~~~
(01) init:  li    x10,0x0000FFFF    # Initialize several registers
(02)      li    x11,0x00000001    #
(03)      li    x12,0xFFFFFFFF    #
(04)      li    x13,0x00000003
(05)
(06)      add   x20,x10,x11      # Op: x10 + x11 (0x0000FFFF + 0x00000001)
(07)                                # Result: x20=00010000; x10 & x11: no change
(08)
(09)      addi  x21,x10,-0xFF    # Op: x10 + -0xFF (0x0000FFFF + 0xFFFFF01)
(10)                                # Result: x21=0000FF00; x10: no change
(11)
(12)      sub   x22,x13,x12      # Op: x13 - x12 (0x00000003 - 0xFFFFFFFF)
(13)                                # Result: x22=0x00000004; x12 & x13: no change
(14)
(15)      sub   x23,x13,x11      # Op: x13 - x11 (0x00000003 - 0x00000001)
(16)                                # Result: x22=0x00000002; x13 & x1: no change
(17) #~~~~~ program fragment ~~~~~

```

Figure 10.5: Usage examples for register and immediate forms of the arithmetic instructions.

10.2.3 Shift Instructions

The RISC-V MCU supports integer-based math with a modest but useful set of shift instructions. Recall that a single bit shift to the right or left is a fast way of dividing and multiplying by two. The left shift is true multiplication by two so long as the MSBs are not lost from the left end of the result. The right shift is a true divide by two except in the case that the LSB of the value being shifted is '1'; truncation occurs when we right-shift a set LSB value. We typically use arithmetic shifts for signed values and logical shifts for unsigned values.

RISC-V MCU has three types of shift instructions including a shift left, shift right, and an arithmetic shift right. Either each of these shifts can be simple shifts (shift one bit positions) or barrel shifts (shift multiple bit positions). For each of the shift instructions, one of the source operands provides the number of bits to shift. For reg-type instructions, the lower five bits in the second source register provides the number of bit positions to shift. For immed-type instruction, the lower five bits of the immediate operand provides the number of bit positions to shift. While we can consider these shift-type operations as a form of mathematical operations, we've grouped them separately from the arithmetic instructions⁴.

The first thing to know about the shift instructions is their most basic difference. The logic shifts (the non-arithmetic shifts) automatically replaced bit positions made vacant by the shift with zeros (the same for shifts in either direction). The arithmetic shift (which only shifts right) fills in vacated bit positions with copies of the sign-bit associated with the source operand that the instruction shifts. We sometimes refer to these operations as zero-filling or sign-filling. Table 10.3 shows some example of shift instruction usage including other information as well.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
R-Type	sll rd,rs1,rs2	$rd \leftarrow rs1 \ll rs2[4:0]$	sll x10,x20,x21	5 rs2 LSBs only zero-filled
I-Type	slli rd,rs1,imm	$rd \leftarrow rs1 \ll imm[4:0]$	slli x8,x8,0xF0	imm: 5 LSB only zero-filled
R-Type	srl rd,rs1,rs2	$rd \leftarrow rs1 \gg rs2[4:0]$	srl x5,x5,x16	5 rs2 LSBs only zero-filled
I-Type	srl i rd,rs1,imm	$rd \leftarrow rs1 \gg imm[4:0]$	srl i x7,x8,15	imm: 5 LSB only zero-filled
R-Type	sra rd,rs1,rs2	$rd \leftarrow rs1 \gg rs2[4:0]$	sra x30,x28,x8	5 rs2 LSBs only sign-filled
I-Type	srai rd,rs1,imm	$rd \leftarrow rs1 \gg imm[4:0]$	srai x10,x9,0x12	imm: 5 LSB only sign-filled

Table 10.3: The two forms associated with the four logic instructions.

Figure 10.6 show some examples of shift-type instructions. The instructions follow the basic rules of logical and arithmetic shifts, but there are a few things to note here and there, which we list here:

- The number of bits to shift for any shift instruction is always a positive number. It is therefore not possible to shift in a negative direction.
- The first example on line (05) performs a shift left operation base on the value in x12. The value is x12 is much larger than 32; the underlying RISC-V hardware only considers the lower five bits of the x12 for the shifting operation. The programmer should strive to not rely on the hardware to do the “right thing” in such cases.
- The arithmetic shift propagates the sign bit in the operation. To be clear, the sign-bit is the left-most bit in the first (left-most) source operand.

⁴ Recall that a single bit-level shift-left and shift-right operations are clever ways to perform a multiply by 2 or divide by 2, respectively.

(00)	#~~~~~ program fragment ~~~~~
(01)	init: li x10,0xFFFF0A8B # Initialize three registers
(02)	li x11,0xF000000F #
(03)	li x12,0xFF000010 #
(04)	li x13,0x80007000 #
(05)	li x14,0x00007000 #
(06)	
(07)	sll x20,x10,x12 # Op: x10 sll x11 (0xFFFF0A8B sll 0xFF000010)
(08)	# Result: x20=0x0A8B0000; x10 & x12: no change
(09)	
(10)	slli x21,x10,0x04 # Op: x10 sll 0xFF (0xFFFF0A8B sll 0x000000FF)
(11)	# Result: x21=0xFFFF0A8B0; x10: no change
(12)	
(13)	srl x22,x10,x12 # Op: x10 srl x12 (0xFFFF0A8B srl 0xFF000010)
(14)	# Result: x22=0x0000FFFF; x10 & x12: no change
(15)	
(16)	srlui x23,x10,0x8 # Op: x10 srl 0x8 (0xFFFF0A8B srl 0x00000008)
(17)	# Result: x23=0x00FFFF0A; x10: no change
(18)	
(19)	sra x24,x13,x11 # Op: x13 sra x12 (0x8000F000 sra 0xF000000F)
(20)	# Result: x24=FFFF0000; x13 & x11: no change
(21)	
(22)	srai x25,x14,0x3 # Op: x14 sra 0x3 (0x8000F000 sra 0x00000003)
(23)	# Result: x25=0000E00; x14: no change
(24)	#~~~~~ program fragment ~~~~~

Figure 10.6: Usage examples for register and immediate forms of the shift-type instructions.

10.3 Auxiliary Instructions

The RISC-V instruction set has several other instructions that we'll mention in this section.

10.3.1 Various Simple Pseudoinstructions Operation: the nop Instruction

There are a few simple pseudoinstructions used by the RISC-V ISA. Because of their simplicity, we're grouping them together in this section.

10.3.1.1 Pseudoinstruction: nop

First, the mnemonic “nop” stands for “no operation”. While it does not make sense to have an instruction that does nothing, there are times assembly language programming land that we want to do nothing. The only reason we ever want to do nothing is to wait for some other event to happen. Thus, the nop instruction serves only to create a delay but changes nothing else in the MCU. Most assembly languages include a nop in their instruction sets.

- For the RISC-V MCU, there is a **nop** pseudoinstruction available. The assembler then translates this instruction of an addi base instruction. Table 10.4 provides the details; we provide the following details.
- The **nop** pseudoinstruction has no operands.
- When the assembly encounters the **nop** pseudoinstruction in the source code, it replaces it with a base instruction. The assembler can actually use one of many different base instructions to “perform” a no operation, but the assembly chooses an **addi** instruction. For example, another way to perform a **nop** would be this instruction: “**ori x1,x1,0**”. Not overly exciting.
- As with many pseudoinstructions, they primarily exist for two reasons: to make assembly language source code easier for humans to write⁵ and understand code. For example, when you see a **nop** instruction in code, you know immediately what the instruction is doing. If you were to see an equivalent base instruction, such as the **addi** in Table 10.4, you might wonder for a moment what the instruction was actually doing.

⁵ As opposed to a computer writing the assembly language code, as is what the typical compiler is responsible for doing.

Instruction Form	Example Usage	Equivalent Base Instruction	Comment
<code>nop</code>	<code>nop</code>	<code>addi x0,x0,0</code>	Do nothing

Table 10.4: On overview of the `nop` pseudoinstruction.

Figure 10.7 and Figure 10.8 show examples of using `nop` instructions in code. The code in both of these figures creates a delay by first initializing a loop count (iteration variable) on line (02). The code in Figure 10.7 uses a do-while loop construct to create the delay while the code in Figure 10.8 uses a while loop construct to create a delay.

```
(00) #~~~~~ code fragment ~~~~~
(01)
(02) init:    li    x10,0xFFF    # input port address
(03)
(04) loop:    addi   x10,x10,-1   # decrement loop count
(05)         nop                    # insert delay
(06)         beq    x10,x10,done  # check condition
(07)         j     loop          # branch to loop
(08) done:                    # done with loop
(09)
(10) #~~~~~ code fragment ~~~~~
```

Figure 10.7: A code fragment implementing a delay using a do-while loop.

```
(00) #~~~~~ code fragment ~~~~~
(01)
(02) init:    li    x10,0xFFF    # input port address
(03)
(04) loop:    beq    x10,x10,done  # check condition
(05)         nop                    # insert delay
(06)         addi   x10,x10,-1   # decrement loop count
(07)         j     loop          # branch to loop
(08) done:                    # done with loop
(09)
(10) #~~~~~ code fragment ~~~~~
```

Figure 10.8: A code fragment implementing a delay using a do-while loop.

10.3.1.2 Pseudoinstruction: `not`

The `not` is a pseudoinstruction is once again quite useful. Its official purpose is to perform a 1's complement (toggles all the bits, or does a bitwise inversion) on the source operand and store the result in the destination operand. We of course recall that toggling a bit is always done with an XOR function, so it's no surprise that the assembler replaces the `not` pseudoinstruction with some type of XOR instruction. Table 10.5 shows the details, and here is some extra explanation.

- The assembler replaces the `not` instruction with a `xori` instruction. While we need to do a bitwise XOR operation on the full 32 bits, it appears the immediate field in the `xori` instruction is limited to a 12-bit value. The hardware is responsible for sign-extending the 12-bit immediate value to a 32-bit value so the instruction does the XOR operation in a true bitwise manner.

Instruction Form	Example Usage	Equivalent Base Instruction	Comment
<code>not rd,rs2</code>	<code>not x8,x9</code>	<code>xori x8,x9,-1</code>	Do 1's complement

Table 10.5: On overview of the `not` pseudoinstruction.

Table 10.6 shows a program fragment that uses the `not` pseudoinstruction. The fragment initializes two registers on line (02-03); the code one lines (05-06) performs a bitwise inversion (1's complement) the values in the initialized registers. Note that on line (06) the source and destination registers are the same.

(00)	#~~~~~ code fragment ~~~~~
(01)	#
(02)	<code>init: mv x10,x0 # clear x10</code>
(03)	<code>li x20,0xAFAFAFAF # load x20 with pointless value</code>
(04)	#
(05)	<code>ex1: not x11,x10 # x11=0xFFFFFFFF after execution (x10 => no change)</code>
(06)	<code>ex2: not x20,x20 # x20=0x50505050 after execution</code>
(07)	#
(08)	#~~~~~ code fragment ~~~~~

Table 10.6: Code fragment example using the `not` pseudoinstruction.

10.3.1.3 Pseudoinstruction: `neg`

The `neg` is a pseudoinstruction is also quite useful. Its official purpose is to perform a 2's complement (toggles all the bits then add 1) on the register specified by the source operand. Programmers typically use the `neg` instruction when working with signed numbers. Be sure to note the difference between the `neg` instruction (2's complement) and the `not` instruction (1's complement). Table 10.7 shows the details, and here is some extra explanation.

- The assembler replaces the `neg` instruction with a `sub` instruction. The instruction then subtracts the source operand from zero and stores the result in the destination register. This is equivalent to toggling all bits and adding one; this is one of the standard tricks associated with representing numbers 2's complement format.
- The implication of using the `neg` pseudoinstruction is that the instruction is assuming the value in the source register is in 2's complement format; otherwise, the instruction would make no sense.

Instruction Form	Example Usage	Equivalent Base Instruction	Comment
<code>neg rd,rs2</code>	<code>neg x8,x9</code>	<code>sub x8,x0,x9</code>	Do 2's complement

Table 10.7: On overview of the `neg` pseudoinstruction.

Table 10.8 shows an example using the `neg` pseudoinstruction. The code first initializes two registers on line (02-03). The code then uses the `neg` pseudoinstruction to perform a 2's complement on the initialized values. We included a `not` pseudoinstruction in this example on line (08) to highlight the difference between the `neg` and `not` pseudoinstruction; note that the value in `x20` and `x20` are not equivalent, but only differ by the 1, which underscores the difference in the definitions of the 1's and 2's complement.

(00)	#~~~~~ code fragment ~~~~~
(01)	#
(02)	init: li x10,-1 # load x10 with -1 (x10=0xFFFFFFFF)
(03)	li x20,0x000000FF # load x20 with pointless value
(04)	#
(05)	ex1: neg x11,x10 # x11=0x00000001 after execution (x10 => no change)
(06)	ex2: neg x21,x20 # x21=0xFFFFFFFF01 after execution (x20 => no change)
(07)	#
(08)	ex3: not x20,x20 # x20=0xFFFFFFFF00 after execution
(09)	#
(10)	#~~~~~ code fragment ~~~~~

Table 10.8: Code fragment example using the not pseudoinstruction.

10.3.2 xxxxSet If Less Than: slt, slti, sltu, sltiu

The RISC-V ISA include six base conditional branch instructions and ten other conditional branch pseudoinstructions, all of which are program flow control related instructions. These instructions allow the program to make conditional branch decisions based on the values in two registers. There are actually two drawbacks with the branch-type instructions. First, the instruction “branches” if the condition is met; but it may be the case where you don’t want to branch. You may have other tasks to complete before you actually need to branch. Second, the branch-type instructions base their inherent comparison on two registers only. It sometimes can become inconvenient to not be able to base program flow control decisions based on immediate values rather than register values.

The RISC-V ISA also contains a group of compare-oriented instructions that provide programmers with expanded flexibility in performing branches. The group of “set if less than” instruction partially solves the issue of being constrained to basing branches on register comparisons by provided the ability to establish “less than” relationships based on immediate values.

Table 10.9 provides an overview of the set if less than (slt) instructions. There are two main types of slt-type instructions. All slt-type instructions set the destination register (writes ‘1’ to the register) if the result of the comparison is true; the differences lie in the comparisons made by the instructions. The immediate forms of the instructions (**slti** & **sltiu**) compare a register to an immediate value, while the register-immediate forms of the instructions compare two register values. Additionally, the instructions either interpret the two operands differently, as both unsigned values (**sltu** & **sltiu**) or signed values (**slt** & **slti**).

Table 10.9 uses some special vernacular in the associated RTL to describe the instructions. First, it uses “<” and “<_s” for unsigned and signed comparisons, respectively. Second, it uses a C programming language type operator to describe the result of the comparison. The “?:” is an arithmetic if operator. This operator includes an expression on the left of the question mark, and a value on each side of the colon; the RTL statements in Table 10.9 use a comparison in place of the expression. If the comparison evaluates are true, the operator assigns the value on the left side of the colon (‘1’) to the destination register; otherwise, the operation assigns the value on the right side of the operator (‘0’). Thus, the destination register is either set or cleared as a result of executing any one of these slt-type instructions.

The immediate forms of the slt-type instructions represent the immediate operand in a 12-bit field in the instruction format. The hardware interprets these values as signed values, which gives the immediate value an effective range of [-2048,2047]. The RISC-V MCU hardware sign-extends these values prior to the comparison. Table 10.9 indicates sign-extension of the immediate value with using the “sext(imm)” notation in the RTL statement.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
R-Type	<code>slt rd,rs1,rs2</code>	$rd \leftarrow (rs1 <_s rs2) ? 1 : 0$	<code>slt x10,x5,x21</code>	signed compare
I-Type	<code>slti rd,rs1,imm</code>	$rd \leftarrow (rs1 <_s sext(imm)) ? 1 : 0$	<code>slti x8,x9,0xF0</code>	signed compare 12-bit signed imm
R-Type	<code>sltu rd,rs1,rs2</code>	$rd \leftarrow (rs1 <_u rs2) ? 1 : 0$	<code>sltu x5,x6,x16</code>	unsigned compare
I-Type	<code>sltiu rd,rs1,imm</code>	$rd \leftarrow (rs1 <_u sext(imm)) ? 1 : 0$	<code>sltiu x7,x8,25</code>	unsigned compare 12-bit signed imm

Table 10.9: The two forms associated with the four logic instructions.

Table 10.10 shows a code fragment that uses the slt-type instructions. The code fragment groups the two-register compare slt-types (`slt` & `sltu`) and the register-immediate compare types (`slti` & `sltiu`) in the code for easy comparisons for humans who may actually be reading this. Here is some extra explanation regarding the Table 10.10 code fragment.

The code initializes from generic values on lines (02-03) to use in the code below it. We load x11 with “-1”, which the assembler represents in x11 in 2’s complement format (0xFFFFFFFF).

- The `slt` instruction on line (05) compares the values in register x11 & x12. This instruction causes the hardware to interpret the values in x11 & x12 as signed value. The RISC-V hardware interprets the value in x11 (0xFFFFFFFF) as -1, which is less than the value in x12, which is zero. As a result the hardware writes a ‘1’ (0x00000001) to x10. The hardware overwrites whatever value is in x10; the values in x11 & x12 do not change.
- The `sltu` instruction on line (06) uses the same two source operands as the slt instruction on line (05). The value written to x20 is different, however, because the sltu instruction directs the hardware to interpret the values in x11 and x12 as unsigned values. This means that the value in x11 (0xFFFFFFFF) is now a large positive number instead of -1 as it was in the previous instruction. The result is that the value in x11 is no longer less than the value in x12, so the hardware clears the x20 register.
- Lines (09-10) show the register-immed version of the slt-type instructions. Line (09) compares x11 to 1; the instruction directs the hardware to treat the value in x11 as a signed number. The result is that x10 is written with ‘1’ because -1 is less than the immediate operand of ‘2’. The sltiu instruction on line (10) is similar but the instruction directs the hardware to interpret the value in x11 as an unsigned number (0xFFFFFFFF). The hardware clears the value in x20 because the value in x11 is not less than the immediate value of “2”.

(00)	#~~~~~ code fragment ~~~~~
(01)	#
(02)	<code>init: li x11,-1 # load x10 with -1 (x10=0xFFFFFFFF)</code>
(03)	<code>mv x12,x0 # clear x12</code>
(04)	#
(05)	<code>ex1: slt x10,x11,x12 # x10 = 1 after execution (x11,x12 => no change)</code>
(06)	<code>ex2: sltu x20,x11,x12 # x20 = 0 after execution (x11,x12 => no change)</code>
(07)	#
(08)	#
(09)	<code>ex3: slti x10,x11,2 # x10 = 1 after execution (x11 => no change)</code>
(10)	<code>ex4: sltiu x20,x11,2 # x20 = 0 after execution (x11 => no change)</code>
(11)	#
(12)	#~~~~~ code fragment ~~~~~

Table 10.10: Code fragment example using the slt-type instructions.

10.3.3 The Load Address Instruction: `la`

Assembly code uses labels as a way to make it easier for humans to write and understand code without needing to know about the underlying mechanics of the instruction. Labels serve two main purposes in programs: 1) as “no-cost” comments, and, 2) to locate specific portions of the code required by other instructions (including pseudoinstructions). Labels used as comments make code more readable for humans without increasing the program length, and without using actual comments (using the “#” symbol). The other use of labels is to locate sections of code that the program requires for program flow control issues such as conditional and unconditional branches (jumps), which includes subroutine calls.

Labels represent addresses of either data or instructions in memory. The good news is that the assembler handles most of the underlying details on this, so we won’t go into too much detail in this section. However, for some program flow control issues, we need to be able to manipulate the address value associated with a label. In those situations, we use the **la** instruction.

The **la** instruction is a pseudoinstruction that the assembler translates to an **auipc** and **lui** instruction. The mnemonic stands for “load address”, which means the instruction places the value associated with a label into a register. The value it places into the register is an address of data in the main memory module; this data can either be the address of true data or the address of an instruction as the same memory module holds both types of data. Table 10.11 provides an overview of the **la** instruction; astute programmers can find a full description in the associated assembly language manual.

Instruction Form	Instruction RTL	Example Usage	Comment
la rd, label	$rd \leftarrow \&label$	la x8, my_label	Numerical value of my_label copied to x8

Table 10.11: The overview of the **la pseudoinstruction.**

The best way to understand the operation of the **la** instruction is to see it in code. Figure 10.9 shows a code fragment that uses two **la** instructions. This is a simple example; we’ll later use these instructions in the proper context when we discuss look up tables (LUTs) in Section 14.5 and interrupts in Chapter 13. Here are the important points to notice in Figure 10.9:

- The code is a fragment, which means it’s a part of some larger program. It doesn’t do anything meaningful without seeing the other parts of the program.
- The code uses a bunch of **nop** instructions as placeholders for more meaningful instruction. Recall that a **nop** is a pseudoinstruction that does nothing.
- The code uses five labels: each of the labels represent the numerical value of the first instruction following it. The code provides the address of the **emu** label in the comment, thus the address of the **emu** label in instruction memory is 0x00000040 as the comment indicates.
- The **ex0** label does not have an instruction on the same line as do the other labels. This is common in programs and is completely legal. The **ex0** label takes on the address of the next instruction in the code, which is the instruction on line (05). Thus, the numerical values associated with the **ex0** and **ex1** labels are equivalent.
- The **la** instruction on line 06 copies the value of the **emu** label (0x00000040) into register x20, which is an arbitrary register.
- The **la** instruction on line (05) copies the value of the **cow** label into register x10. For this instruction, we need to count forward in the code from address 0x00000040 to determine the address of the **cow** label, which is 0x0000005C. Recall that each instruction in program memory requires four bytes of space and main memory is byte-addressable.

```

(00) #~~~~~ code fragment ~~~~~
(01) emu:  nop          # placeholder instruction: addr=0x00000040
(02)      nop          # placeholder instructions
(03)      nop
(04) ex0:
(05) ex1:  la    x10,cow # place associated value of cow (0x0000005C) into x10
(06) ex2:  la    x20,emu # place associated value of emu (0x00000040) into x20
(07)
(08)      nop          # placeholder instructions
(09)      nop          #
(10)      nop          #
(11) cow:  nop          #
(12) #~~~~~ code fragment ~~~~~

```

Figure 10.9: Code fragment example using the `la` instruction.

10.3.4 Other Loading-Type Instructions: `auipc` & `lui`

The RISC-V ISA contains many generic base instructions that provide the ISA with significant flexibility without having to add more instructions. What this means is that the usefulness of some base instructions is not readily apparent because we typically don't use these instructions directly. Because these instructions are part of other useful pseudoinstructions, the assembler converts those pseudoinstructions to the base instructions as part of the assembly process.

This section describes the `auipc` and `lui` base instructions. The assembler translates the call pseudoinstruction to `auipc` & `jalr` base instructions; the assembler also translates the `la` pseudoinstruction into an `auipc` & `addi` base instructions. The assembler translates the `li` pseudoinstruction into a `lui` instruction (and possibly an `addi` instruction).

10.3.4.1 Add Upper Immediate to PC Instruction: `auipc`

The primary purpose of the `auipc` instruction is to load a copy of the current program counter to a register. The `auipc` instruction is primarily used indirectly by programmers because it is part of the `call` pseudoinstruction (the other part of the call pseudoinstruction is a `jalr` instruction). Table 10.12 shows a description of the `auipc` instruction along with some usage details. Here are some other fun facts about this instruction.

- The `auipc` instruction loads the sum of the current PC and a modified immediate value into the destination register. The instruction sign-extends the immediate value and shifts it left 12 bit locations before being adding it to the destination register. The instruction clears the lower 12-bits in the destination register.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
U-Type	<code>auipc rd,imm</code>	$rd \leftarrow PC + (\text{sext}(\text{imm}) \ll 12)$	<code>auipc x7,25</code>	Lower 12 bits are cleared

Table 10.12: An overview of the `auipc` instruction.

10.3.4.2 Load Upper Immediate Instruction: `lui`

The `lui` instruction is similar to the `auipc` instruction. It's once again one of those instructions that programmers don't use often in a direct manner, but use often in an indirect manner. The assembler translates the `li` pseudoinstruction into `lui` instruction.

Similar to the `auipc` instruction, the `lui` instruction loads a modified immediate value into the destination register. The difference from the `auipc` instruction is that the PC value is not included in value loaded into the register. The instruction sign-extends the immediate value and shifts it left 12 bit locations before loading it into destination register. Table 10.13 provides an RTL description of the `lui` instruction.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
U-Type	<code>lui rd,imm</code>	$rd \leftarrow (\text{sext}(\text{imm}) \ll 12)$	<code>lui x18,47</code>	Lower 12 bits are cleared

Table 10.13: An overview of the `lui` instruction.

10.3.5 Loading Immediate Values: `li`

The `li` pseudoinstruction provides a way to load an immediate value into a register. The assembler translates the `li` pseudoinstruction to a `lui` instruction and possibly an `addi` instruction, depending on the magnitude of the immediate value. Programmers can use either negative or positive values for the operand of this instruction. The assembler handles the details of where the `li` instruction translates to one or two base instructions. Table 10.14 provides an overview of the `li` pseudoinstruction.

Instruction Form	Instruction RTL	Example Usage	Comment
<code>li rd,imm</code>	$rd \leftarrow \text{imm}$	<code>li x8,20</code>	Immediate value loaded into destination register

Table 10.14: The overview of the `li` pseudoinstruction.

10.4 Program Flow Control

Our original definition of a computer was a “piece of hardware that sequentially executes a stored program”. The key word here is *sequentially*. Computer programs typically execute programs in sequence: one instruction after another. Recall that program memory stores the instructions, so sequential program execution essentially means that computers execute the instruction at one memory address location, and then the computer executes the instruction at the next contiguous memory address, etc. Programs do not always simply execute instructions sequentially; if they did so, the program would quickly run out of instructions to execute.

The notion of *program flow control* deals with instructions that have the ability to cause the computer to execute instructions in some order other than strictly sequentially. In other words, some instructions instruct the MCU to “jump” somewhere in program memory other than to the instruction following the current instruction being executed. We consider sequential instruction execution as “normal operation”, while everything else falls into the category of non-normal operation, or more aptly put, program flow control. This section covers the main type of flow control instructions: branch instructions. Additionally, there are two types of branch instructions: 1) unconditional branches, or jumps, and 2) conditional branches.

As you will see, there are other program flow control issues in the RISC-V MCU including subroutines and interrupts. As it turns out, the RISC-V ISA does not have base instructions dedicated to working with subroutines; the RISC-V MCU deals with subroutines using the two available unconditional branch instructions. Interrupts are also part of program flow control; we cover the RISC-V interrupt architecture in another chapter.

10.4.1 Labels Revisited

We first mentioned labels in Section 8.4.1.4, but we purposely skipped over an important detail regarding the true non-commenting aspect of labels. We revisit the notion of labels in this chapter because of labels are critically important to the notion of jumping around in programs. In other words, it’s not a big deal to execute the instructions stored in program memory in a sequential manner; it does, however, require a more complicated mechanism to have the MCU execute instruction in a non-sequential manner, which is where the notion of branching comes in.

The truth is that labels represent numbers, and more specifically addresses of instructions in program memory. We can omit most of the underlying details regarding labels in this programming section of the text because we can more appropriately discuss them when describing the RISC-V MCU hardware. However, we mention a few details here. Labels represent addresses; when you branch to some other section of instructions in your program, you do so by loading the address value associated with a given label into a register that the hardware uses as an

index into program memory. We can mostly skip over those details for this section and deal 100% with labels without knowing anything of the underlying details. The notion here is that all the branch instructions generally include an operand that is an address value. We work to avoid using address values directly when dealing with labels; we instead use only labels and allow the assembler to convert those labels into actual numbers as the assembler translates the assembly language source code to machine code. So once again, the pure programmer gets away without knowing the full story embedded in the underlying hardware. I'm sure glad I understand hardware.

Example 10.3: Label Values

For the following RISC-V assembly language code fragment, if the label **init** has a value of 0x00000E00, provide the following information:

- What are the numeric values (in hex) with all the labels in the fragment
- What is the address in program memory of the **j** instruction?
- What is the relative address of **t_100** relative to **init**, **c22** relative to **loop1**, **loop1** relative to **t_100**, and **t_100** relative to **loop1**.

```

init:    mv    x15,x10      # save a copy
         li    x21,0x00000F00 # 100's bit mask
         li    x22,0x000000F0 # 10's bit mask
c22:    li    x23,0x0000000F # 1's bit mask
         mv    x20,x0      # zero accumulator

t_100:  and   x15,x15,x21  # mask 100's nibble
         srl  x15,x15,8    # shift to lowest position
loop1:  beqz  x15,t_10     # go to tens if zero
         addi x20,x20,100  # accumulate 100s
         addi x15,x15,-1   # decrement loop count

junk:   j     loop1       # do it again

```

Solution: The key thing to recall doing problems such as this is that each RISC-V instruction requires four bytes of program memory. The code in this example is only instructions, so the entire problem becomes an exercise in doing math.

- This part of the problem is simply a matter of doing the math, where each instruction advances the address by four.

label	Value	comment
init	0x00000E00	Given by problem
c22	0x00000E0C	3 instrs past init
t_100	0x00000E14	5 instrs past init , 2 past c22
loop1	0x00000E1C	7 instrs past init , 2 past t_100
junk	0x00000E28	10 instrs past init , 3 past loop1

Table 10.15: Solutions to part a)

- We already did the math for the address of the **j** instruction; it's the same value that is associated with the **junk** label, which is 0x00000E28.
- The solutions with explanation to part c). Note that because we are working with instructions, all the answer are divisible by four. Also, negative values indicate going backwards in the code (to lower memory addresses) while positive values represent going forward in the code (to higher memory addresses).

label	Value	comment
t_100 relative to <code>init</code>	20	Five instructions forward
c22 relative to <code>loop1</code>	-16	Four instructions backwards
<code>loop1</code> relative to t_100	8	Two instructions forward
t_100 relative to <code>loop1</code>	-8	Two instructions backwards

Table 10.16: Solutions to part c)

10.4.2 Branch Instructions

Branch instructions *can*⁶ cause the MCU to execute an instruction that is not the next instruction in program memory. There are two types of branch instructions: unconditional branches and conditional branches. The RISC-V refers to unconditional branches as “jumps” and conditional branches as “branches”. Both types of branch instructions potentially alter the sequence in which the MCU executes instructions from program memory. The difference between these two types of instructions is that unconditional branches always change the program execution sequence while conditional branches may or may not change the instruction execution sequence depending on certain conditions in the MCU. The main thing to keep in mind is that branch instructions have the ability to transfer program control from one instruction to another instruction that is not necessarily the next instruction in the sequence.

10.4.2.1 Unconditional Branch Instructions

As the name implies, when the MCU executes an unconditional branch instruction, the MCU always takes the branch. In other words, program control always transfers to another instruction in program memory that is not the instruction following the instruction just executed⁷. The RISC-V instruction set contains two unconditional branch instructions: `jal` & `jalr`. The mnemonics for these instructions roughly mean “jump and link” for `jal` and “jump and link register” for `jalr`. Table 10.17 shows the two of the unconditional branch instructions. Here are a few things to notice about Table 10.17:

- These instructions officially have different types, where the `jal` instruction is a “J-type” and the `jalr` instruction is an “I-type”. The difference in the number of operands is what requires these two instructions to have different types. The pure programmer does not need to be aware of this level of detail; we include it here for completeness.
- There are two forms of each instruction. This means that you can use either form of each instruction in your source code. The number of operands forms the difference in the forms, where the forms with the most operands is the full instruction. If you don’t include all the operands, the assembler makes some assumptions. The difference in both instructions is the inclusion of the `rd` operand. In both cases, the assembler uses `x1` for the `rd` operand if you use a form that omits it.
- From the Instruction RTL column, you can see that both forms store the current PC (program counter) and modify the current PC. Note that the PC holds the addresses of the current instruction that the hardware is executing, which makes it an index into program memory. The current is (adjusted to point at the next instruction) is stored in the `rd`, the destination register. Both instructions also change the PC, which underscores the major difference between these two instructions. The new PC for the `jal` instruction is a function of the immediate value; the new PC for the `jalr` instruction is a function of the source register (`rs1`) plus the immediate value.
- The difference the new PC value listed in the previous bullet is significant, sort of. As you’ll soon see, the pure programmer can get away without knowing these details, but everyone else needs to at one point understand these details. The functional difference is that you can use either the `jal`

⁶ We use the word “can” here because some types of branch instructions don’t always branch.

⁷ You can branch to the instruction following the branch instruction, but that generally means you’re not understanding the point of the branch instruction.

or **jalr** instruction to call a subroutine, but you can only use the **jalr** instruction to return from a subroutine. We'll discuss these details further in a later chapter. One last thing to note here is that the "r" in the **jalr** mnemonic stands for register, which refers to the new PC value is also a function of some register.

- We opted to present these instructions using a "lab" expression, which is short for "label", even though the official documentation uses an "imm" expression for the thing. The expression is eventually a value, as the "imm" implies, but it only becomes a value after the assembler makes it into one. This means that you can't use a raw number for the "lab" value when you use these instructions in your source code; you have to use a label. Even if you could use an actual value, you would not want to because it would make your code unmanageable and hard to modify.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
J-Type	jal rd,lab	$X[rd] \leftarrow PC + 4$ $PC \leftarrow PC + lab$	jal x8,label	Return address stored in X[rd]; lab is signed
	jal lab	$x1 \leftarrow PC + 4$ $PC \leftarrow PC + lab$	jal label	Return address stored in x1 lab is signed
I-Type	jalr rd,rs1,lab	$X[rd] \leftarrow PC + 4$ $PC \leftarrow rs1 + lab$	jalr x5,x6,label	Return address stored in X[rd]; lab is signed
	jalr rs,lab	$x1 \leftarrow rs1 + lab$	jalr x7,label	Return address stored in x1; lab is signed

Table 10.17: Two forms of the two unconditional branch instructions.

The creators of the RISC-V instruction set designed the **jal** and **jalr** instructions for genericity and efficiency; the designers didn't design them for easy use or quick understanding. The two instruction also serve to handle to well-known areas where executing the instruction other than the next instruction (jumping) is required: *calling* subroutines and *returning* from subroutines. The other area we use unconditional branches is to continue processing in an iterative loop. We cover iterative loops later in this chapter, and everything you want to know about subroutines in another chapter. This stuff makes more sense when you see it used in actual assembly language programs; we're almost ready to do that.

Although the application of **jal** and **jalr** instructions is not intuitive, we rarely if ever are required to use these instructions directly in our programs. We instead use one of the pseudoinstructions associated with these instructions, which we conveniently list in Table 10.18. Here is some useful stuff to note about Table 10.18:

- You can find more details associated with each of the listed pseudoinstructions in Table 10.18 in the assembly language manual.
- The assembler is responsible for converting every pseudoinstruction into a single or set of base instructions.
- The programs we write typically use the **j**, **call**, and **ret** pseudoinstructions; we use the **jr** instruction much less often. We use the **j** pseudoinstruction as a jump associated with iterative loops; we use the **call** and **ret** pseudoinstructions when we access subroutines.
- Generally speaking, the assembler translates the **call** instruction to two instructions. Using two instructions allows for a larger jump range. Your assembler may be smart enough to only use one instruction, but probably not.
- Once again, despite what any other documentation says, you must use actual labels with these instructions. The assembler then converts those labels to actual numerical values, which as then subsequently stored as part of the machine code associated with the base instruction.

Instruction Form	Equivalent Base Instruction(s)	Example Usage	Comment
<code>j label</code>	<code>jal x0, label</code>	<code>j label</code>	Jump to instruction associated with label
<code>jr rs1</code>	<code>jalr x0, 0(rs1)</code>	<code>jr x8</code>	Jump to instruction at address in rs1
<code>call rd, label</code>	<code>auipc rd, hi(label)</code> <code>jalr rd, lo(rd)</code>	<code>call x5, subrot</code>	Jump to instruction associated with label; Store current address in rd
<code>call label</code>	<code>auipc x1, hi(label)</code> <code>jalr x1, lo(x1)</code>	<code>call subrot</code>	Jump to instruction associated with label; Store current address in x1
<code>ret</code>	<code>jalr x0, 0(x1)</code>	<code>ret</code>	Jump to instruction at address in x1

Table 10.18: The program flow control pseudoinstructions and their base instruction translations.

One final comment here. The primary difference between these two instructions is how they calculate where they jump to. The `jal` instruction uses the immediate value as a signed offset from the current instruction, while the `jalr` instruction uses a register as to hold the address to branch to. The `jal` instruction can thus jump as `FIXME` far in instruction memory as the `jalr` instruction. We'll cover the underlying details in the hardware section of this text.

Example 10.4: Program with Unconditional Branch

Write a RISC-V assembly language program that continuously reads data from port address 0x11003000, negates that data, and outputs the result to port address 0x11005000.

Solution: Figure 10.10 shows the solution to this example. This problem has similarities to previous problems, so we'll only describe the new stuff:

- This program includes a header that provides a description of program on lines (00-04). Code goes in the text segment so we use a `.text` assembler directive on line (05).
- The initialization code includes an `init` label, which the program uses to place the port addresses into registers. Note that the choice of registers in this program is arbitrary in that we could use any register other than `x0`.
- The main code in the program starts at line (10), as indicated with the `main` label. The program inputs data on line (10) using an `lw` instruction. The problem did not state a data size so we opt to use words. The program negates the input data on line (11) using a `neg` pseudoinstruction, and then output on line (12) using a `sw` instruction. The fact that the problem stated to "negate" the input data implies that the input data was signed data; the `neg` pseudoinstruction treats the data a signed when it performs a two's complement on the data.
- The `j` instruction on line (11) is the program flow instruction, which is a pseudoinstruction for that performs an unconditional branch. . This instruction directs program execution to some other executable instruction in the program, which is in this program, is to the instruction on the line with the `main` label.

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x11003000, negates that data, and outputs the result to port
(03) # address 0x11005000.
(04) #-----
(05) .text                                # code goes in the text segment
(06)
(07) init:  li    x10,0x11003000          # input port address
(08)        li    x11,0x11005000          # output port address
(09)
(10) main:  lw    x20,0(x10)              # input data
(11)        neg   x20,x20                 # take the 2's complement of the data
(12)        sw    x20,0(x11)              # output data
(13)
(14)        j     main                    # repeat I/O sequence

```

Figure 10.10: Solution for this example.

10.4.2.2 Conditional Branch Instructions

While the unconditional branch instructions don't provide any options in terms of program flow control, the conditional branch instructions do. Unconditional branch instructions utilize the two register values in order to determine whether to branch or not. When certain conditions associated with those values test as true, the instruction takes the branch; otherwise, the instruction does nothing and program flow control passes to the instruction following the branch instruction (which is the next instruction in program memory).

There are six base conditional branch instructions in the RISC-V MCU instruction set. There are also ten other conditional branch pseudoinstructions derived from the six base conditional branch instructions. The fact that there are a relatively high number of base and pseudoinstructions underscores their importance in assembly language programming. Table 10.19 lists the six conditional branch base instructions. And of course, here are a few interesting items to note about Table 10.19.

- Most importantly, notice that conditional branches are based upon the conditions in two register values. This is sometimes quite restrictive, but that's what we have to work with in the RISC-V ISA. The existence of "set if less than" instructions provide somewhat of a workaround for the limited flexibility of conditional branch instructions, but we leave those to another section.
- We once again use the "label" notation in our instruction description. Other documentation uses "imm" to reflect an immediate value, but the assembler actually rejects numerical value for the immediate operand.
- We opt to use C programming language operators to show the relationship between registers in the Comment column; we also add an external note as well.
- The instructions consider the values in the source registers to be signed values unless stated otherwise. The **bgeu** and the **bltu** instructions are the only two instructions that treat the source operands as unsigned values.

Instruction Form	Example Usage	Comment
beq <i>rs1,rs2,label</i>	beq <i>x10,x11,label</i>	branch if $x[rs1] == x[rs2]$; (equal)
bne <i>rs1,rs2,label</i>	bne <i>x23,x10,label</i>	branch if $x[rs1] != x[rs2]$; (not equal)
bge <i>rs1,rs2,label</i>	bge <i>x20,x21,label</i>	branch if $x[rs1] >= x[rs2]$; (\geq)
bgeu <i>rs1,rs2,label</i>	bgeu <i>x8,x9,label</i>	branch if $x[rs1] >= x[rs2]$; (unsigned)
blt <i>rs1,rs2,label</i>	blt <i>x28,x29,label</i>	branch if $x[rs1] <= x[rs2]$; (\leq)
bltu <i>rs1,rs2,label</i>	bltu <i>x4,x11,label</i>	branch if $x[rs1] <= x[rs2]$; (unsigned)

Table 10.19: The RISC-V conditional branch base instructions.

Table 10.20 lists the ten pseudoinstructions and their base instruction equivalents. Here are a few things to note about Table 10.20:

- You can use these instructions as listed in your source code; the assembler translates your pseudoinstructions to base instruction.
- We once again use the “label” to mean a label in your source code. The associated documentation often uses an immediate value instead, even though the assembler rejects such numeric values.

Instruction Form	Example Usage	Equivalent Base Instruction	Comment
beqz <i>rs1,label</i>	beqz <i>x7,label</i>	beq <i>rs1,x0,label</i>	branch if $x[rs1] == 0$
bnez <i>rs1,label</i>	bnez <i>x25,label</i>	bne <i>rs1,x0,label</i>	branch if $x[rs1] != 0$
bgez <i>rs1,label</i>	bgez <i>x23,label</i>	bge <i>rs1,x0,label</i>	branch if $x[rs1] >= 0$
bgt <i>rs1,rs2,label</i>	bgt <i>x20,x21,label</i>	blt <i>rs2,rs1,label</i>	branch if $x[rs1] > x[rs2]$
bgtu <i>rs1,rs2,label</i>	bgtu <i>x8,x9,label</i>	bltu <i>rs2,rs1,label</i>	branch if $x[rs1] > x[rs2]$; (us)
bgtz <i>rs1,rs2,label</i>	bgtz <i>x4,x8,label</i>	blt <i>x0,rs2,label</i>	branch if $x[rs1] > 0$
ble <i>rs1,rs2,label</i>	ble <i>x4,x11,label</i>	bge <i>rs2,rs1,label</i>	branch if $x[rs1] <= x[rs2]$
bleu <i>rs1,rs2,label</i>	bleu <i>x14,x12,label</i>	bgeu <i>rs2,rs1,label</i>	branch if $x[rs1] <= x[rs2]$; (us)
blez <i>rs1,rs2,label</i>	blez <i>x14,x8,label</i>	bge <i>x0,rs2,label</i>	branch if $x[rs1] <= 0$
bltz <i>rs1,label</i>	bltz <i>x22,label</i>	blt <i>rs1,x0,label</i>	branch if $x[rs1] < 0$

Table 10.20: The RISC-V conditional branch pseudoinstructions.

10.5 Standard Assembly Language Constructs

Assembly language is truly a type of programming language and thus shares some basic constructs associated with structured programming. These basic constructs include if-else constructs and iterative loops. When compared to higher-level languages, assembly languages differ wildly in the way they implement these basic constructs. The ungood news is that the RISC-V ISA has a distinct approach to encoding these basic constructs. The good news is that it’s not overly complicated once you understand it and use it a few times. Additionally, we can view an assembly language program as a large conglomeration of these constructs fit together to make a working program that solves the problem at hand. This section provides an overview of these constructs.

10.5.1 If-Then-else Construct

Figure 10.11(a) shows an example of a RISC-V assembly language version of an *if/else* construct. As the name implies, the code does one thing (if some condition is met), or *else* it does some other thing (if the condition is not met). In the case of the program in code fragment in Figure 10.11(a), bases the condition that may or may not be met on the state of the two register operands when the MCU executes the branch instruction. The program

takes one code path if the data meets the condition and another path if the data does not meet condition. Here is a full description of the program.

- The program starts by loading two registers with port addresses on lines (01-02). The “init” label implies this code is some type of initialization code, which inherently means that the program only executes the code once. In the context of this program, that means this code is outside the main loop in the program.
- The program inputs data on line (04), then does some arbitrary task on line (05). The if/else construct action happens starting with the condition branch instruction on line (06). If the value input on line (04) is zero, program control continues with the instruction associated with the “set_zero” label; we can consider this the “if” part of the if/else construct. This the “if” fails (the **beq** instruction condition is not true), the branch is not taken and the program control transfers to the next instruction following the **beq** instruction on line (07). The instruction on line (07) would then be the else part of if/else construct.
- Once the instructions associated with the else clause execute, program flow control transfers to the instruction on line (11) by way of an unconditional branch instruction on line (08). In other words, the code jumps over the code associated with the “if” clause. This is typical if/else operation.
- As with all RISC-V assembly language programs, the non-initialization code forms a loop. After the if/else stuff is done happening, program control transfers to the instruction associated with the “main” label by issuing an unconditional branch instruction on line (12).
- We’ve include a flowchart that models our program. Note circular terminal shape for the start of the program. Most importantly, notice that we have a decision box in there also (the diamond shaped box); be sure to notice that there are two arrows leaving the decision box where one uses a “yes” label and the other uses a “no” label. No “maybe” arrows here.

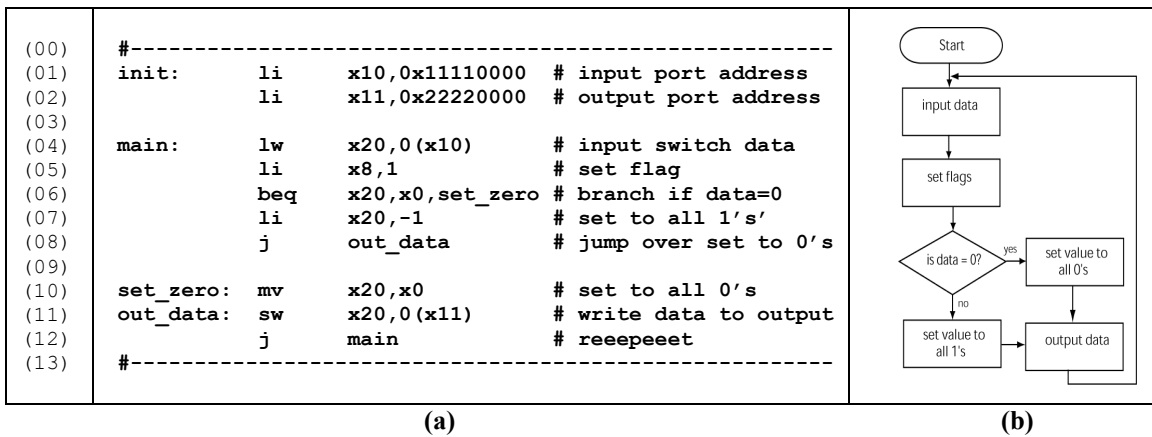


Figure 10.11: An example of if/else construct (a) and a supporting flowchart (b).

10.5.1.1 Special if/else Coding Considerations

You’ve now seen several approaches to coding if/else constructs in assembly language. While all of these ways are functionally equivalent, some approaches are more efficient than others. The issue here is that if/else constructs have at least one unconditional branch instruction (a jump) that directs program flow control to another area in the program. However, there is only one jump. The tendency in assembly language programming, especially with programmers who are new to the assembly language, is to have two unconditional branch statements, one jumps over the if clause, and the other jumps over the else clause. This is functionally correct, but is not efficient. Always strive to ensure you encode your if/else constructs using only one unconditional branch instruction.

Figure 10.12 shows an alternate but equivalent form of the if/else clause of Figure 10.11(a). We show this to remind programmers that there truly are different and truly equivalent ways to code if/else constructs. The code

in Figure 10.12 essentially swaps the order of the if and else clauses compared to the code in Figure 10.11(a). Comparing and contrasting these two examples may be spiritually enlightening.

```

(00) #-----
(01) init:    li    x10,0x11110000 # input port address
(02)        li    x11,0x22220000 # output port address
(03)
(04) main:    lw    x20,0(x10)    # input switch data
(05)        li    x8,1          # set flag
(06)        bnez  x20,set_ones  # branch if data=0
(07) set_zero: mv   x20,x0      # set to all 0's
(08)        j     out_data     # jump over set to 0's
(09)
(10) set_ones: li    x20,-1     # set to all 1's'
(11) out_data: sw   x20,0(x11)  # write data to output
(12)        j     main        # reeepeeet
(13) #-----

```

Figure 10.12: An alternate but equivalent form of the code in Figure 10.11(a).

Example 10.5: Program with Conditional Branch

Write a RISC-V assembly language program that continuously reads data from port address 0x1100A000; if that data is non-negative, the program multiplies that data by two, then outputs that data to port address 0x1100B000; otherwise, the program does nothing with the data.

Solution: Figure 10.13 shows the solution to this example. This problem has similarities to previous problems, so we'll only describe the new items:

- The main code in the program starts at line (07), as indicated with the “main” label. The program inputs data on line (10) using an **lw** instruction. The problem did not state a data size so it is best to simply use word size for the input.
- Once the data is input, the program only acts on it “if” the data is non-zero. So if the data is non-zero, multiply it by two; “else” do nothing. In this case, doing nothing refers to doing nothing to the data; the program conditionally branches to the input instruction on line (10) if the input data is zero. We opted to use the **beq** conditional branch instruction for the if/else construct. If the input data is zero, we branch to “main” to input more data; else, program control drops to the next instruction on line (13).
- If program flow makes it to line (13), the input data in x20 needs to be multiplied by two. There is no multiply instruction in the RISC-V instruction set, but we can use a shift left instruction to accomplish the desired multiplication. We opt to use a **slli** instruction on line(13) with the immediate value of ‘1’ handling the number of times to shift. Once the data is multiplied by two, we output the data on line(14).
- The **j** instruction on line (16) is an unconditional branch instruction, which transfers program control to the instruction associated with the main label on line (10). This causes the program to repeat ad naseum.

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x1100A000; if that data is non-zero, the data is divided by two
(03) # and output to port address 0x1100B000; otherwise the data is discarded.
(04) #-----
(05) .text                                # instruction code goes in text segment
(06)
(07) init:  li    x15,0x1100A000          # input port address
(08)        li    x16,0x1100B000          # output port address
(09)
(10) main:  lw    x20,0(x15)              # input data
(11)        beq   x20,x0,main             # do nothing if data is zero
(12)
(13)        slli  x20,x20,1               # multiply by 2
(14)        sw    x20,0(x16)              # output data
(15)
(16)        j     main                    # repeat I/O sequence

```

Figure 10.13: Solution for this example.

Figure 10.14 shows a flowchart modeling the operation of this program; here are a few interesting items regarding this flowchart.

- This is our first program that contains in conditional branch instruction. This program has an if/else construct, which the flowchart models using a decision box.
- The decision box has one entry point and two exit points. We provide a “yes” and “no” label on the exit points indicating whether the condition in the decision box were met or not. The program takes a different flow path based on this condition.
- Decision boxes in general always have two exit points: a “yes” and a “no”. This never changes. Despite the fact that the decision box in Figure 10.14 contains only one entry point, decision boxes can have multiple entry points.

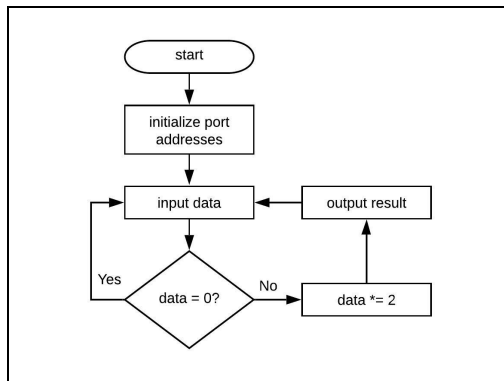


Figure 10.14: A flowchart modeling the operation of this example program.

Example 10.6: Negation and I/O Excitement

Write a RISC-V assembly language program that continuously reads data from port address 0x11003000, negates that data, and outputs the result to port address 0x11005000.

Solution: Figure 10.10 shows the solution to this example. This problem has similarities to previous problems, so we’ll only describe the new stuff:

- This program includes a header that provides a description of program on lines (00-04). Code goes in the text segment so we use a “.text” assembler directive on line (05).
- The `j` instruction on line (11) is the program flow instruction, which is a pseudoinstruction for that performs an unconditional branch. . This instruction directs program execution to some other executable instruction in the program, which is in this program, is to the instruction on the line with the main label.

```

(00) #-----
(01) # Program Description: The program continuously reads data from port
(02) # address 0x11003000, negates that data, and outputs the result to port
(03) # address 0x11005000.
(04) #-----
(05) .text                                # code goes in the text segment
(06)
(07) init:  li    x10,0x11003000           # input port address
(08)        li    x11,0x11005000         # output port address
(09)
(10) main:  lw    x20,0(x10)              # input data
(11)        neg   x20,x20                 # take the 2's complement of the data
(12)        sw    x20,0(x11)             # output data
(13)
(14)        j     main                    # repeat I/O sequence

```

Figure 10.15: Solution for this example.

10.5.2 Iterative Constructs

Probably the most common construct we use in assembly languages is the iterative construct. A significant portion of assembly language programs apply the notion of doing something repeatedly, which we do using some type of iterative construction. There are two types of iterative constructs; we can implement each of those two constructs in two different ways.

In any iterative construct, we employ the conditional branch instruction in such a way as to discern whether we need to continue iterating or not. The conditions we check for fall into two different categories: 1) we know in advance now many times we need to iterate, or 2) we don't know in advance how many times we need to iterate. The conditions for the first type are based on a known count; we thus continue iterating the required amount of times. In other words, the program knows the number of times the construct iterates before it enters the construct. The number of times we iterate for the second type of is determined by the condition of some register that the program is using. In this way, the program does not know how many times the construct will iterate before it enters the construct. Another way to look at this is whether the program knows the iteration count at assemble time or run time. If the iteration count is a constant, then the assembler knows that count (assemble time); otherwise, the iteration count is a variable and the exact count is only known then the construct executes (run time).

We typically refer to these constructs as loops, so we'll do that from now on. We can implement either of these loops in two different ways: 1) while loops, or, 2) do-while loops. The difference between these two loops is simple and distinct: using a do-while loop ensure the loop iterates at least one time. When you use a while loop, the loop may not be iterated even a single time depending upon the condition controlling the loop. A do-while loop always iterates one time even if the iteration count is a variable (not known until run time).

We provide a few examples of iterative constructs in the following section, but we state this disclaimer first. When you're writing code, you need to make whatever comparison you need to do to make the code do what you want it to do. Every loop necessarily contains a conditional branch instruction. Being that there are six base and ten conditional branch pseudoinstructions, we can't provide example of each instruction. Our approach in the next two sections to provide one example of each type of conditional construct. It's the form of the construct that you should strive to understand, which allows you to use whatever conditional branch instruction you need to make your particular construct work in such a way that it your code solves the problem at hand.

10.5.2.1 while Loops

The while loop is one type of iterative loop. The main characteristics of the while loop is that the loop condition is checked before execution enters the body of the loop as well as after each loop iteration. While loops contain at least one conditional and one unconditional branch instruction, where the conditional branch instruction provides a conditional exit from the loop and the unconditional branch provides a continuation of the loop. The unconditional branch in a while loop typically branches to the conditional branch instruction.

Figure 10.16 and Figure 10.17 show an examples of a program fragments that include a while loop that utilizes a known number of iterations and an unknown number of iterations, respectively. For these and the other loop examples that follow, you must pretend the body of the loop is doing something meaningful. Here is stuff to note about Figure 10.16 and Figure 10.17:

- The only thing that makes these loops different is what the code uses for the iteration count. For the example in Figure 10.16, the iteration count is set on line (05), and is thus constant and the iteration loop count is known at assemble time. For the example in Figure 10.17, the code determines the iteration count by inputting a value from the outside world, which means the loop count is variable and is not known until run time. Another way to look at this is that the code in Figure 10.16 knows the iteration count when the code is assembled (it is “known”) while Figure 10.17 does not know the count until the program runs (it is unknown until runtime).
- The example initializes some registers on lines (01-03); the port addresses are arbitrary, but we do differentiate between input and output addresses.
- The iteration count is set on line (05), which do with the **li** instruction in Figure 10.16 and an **lw** instruction in Figure 10.17.
- The first instruction in the while loop is a conditional branch instruction which checks to see if the iteration count is zero. If the count is zero, the program takes the branch and program flow control exits the loop by branching to the instruction on line (15); otherwise, the program simply advances to the next instruction on line (08).
- The body of the while loop is on lines (08-10); the code inputs data, complements that data, and then outputs that data to some external device. Not too exciting, but a placeholder for something else more exciting.
- The administrative part of the loop is on lines (12-13). The instruction on line (12) decrements the iteration count; program flow is then unconditionally directed to line (07), which is effectively the start of the loop.
- The first instruction in the loop is the conditional branch instruction; this is the check condition that effectively allows no iterations of the loop to occur by branching out of the loop (to the “done” label). The unconditional branch instruction unconditionally branches to the start of the loop after the iteration count was decrement on line (12).

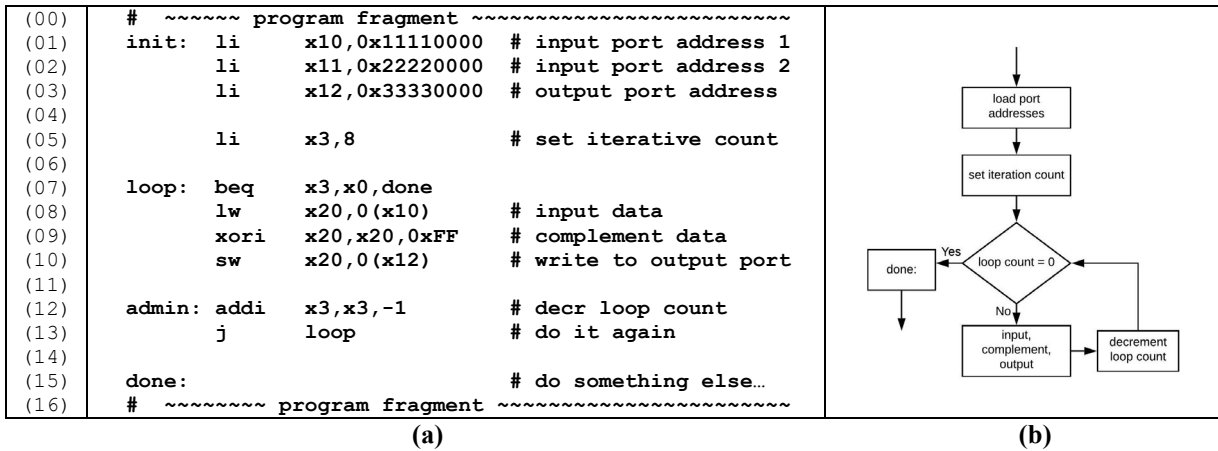


Figure 10.16: An example of a while loop with a known numbers of iterations.

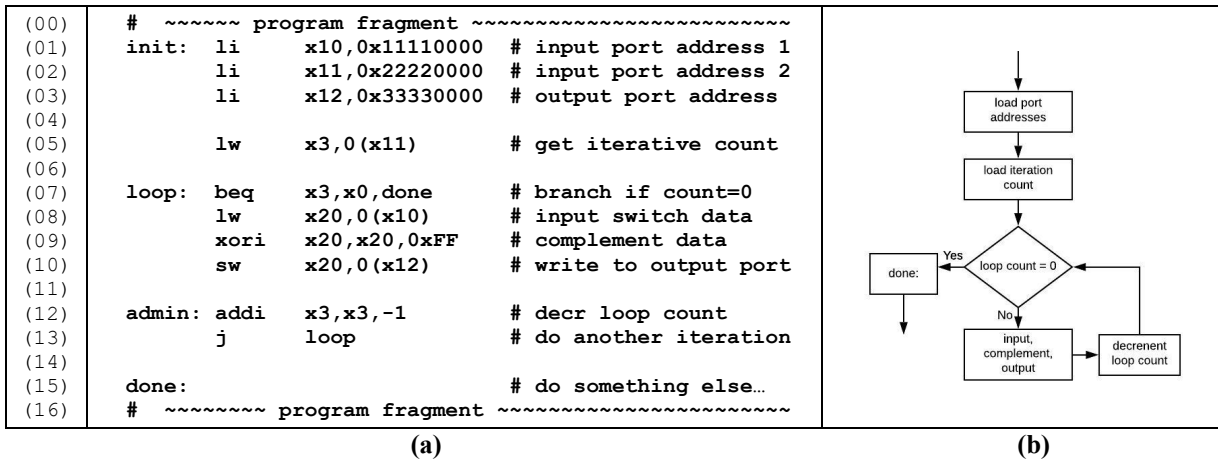


Figure 10.17: An example of a do-while loop with an unknown numbers of iterations.

Figure 10.18 shows an alternative approach to writing the while-loops in these examples, which is to have the conditional branch at the end of the code instead of the unconditional branch. In this case, the program would take the branch when more iterations are necessary. This code would work fine, but the value of the iteration variable would need to be checked before entering the loop to determine if it was zero, which we do before we enter the while-loop on line (06). You never want to allow your code to decrement a loop count of zero before checking to see if that count is zero. The conditional branch on line (06) prevents the code from decrementing a zero on line (12), with the notion is the count is zero, we never want to enter the loop in the first place.

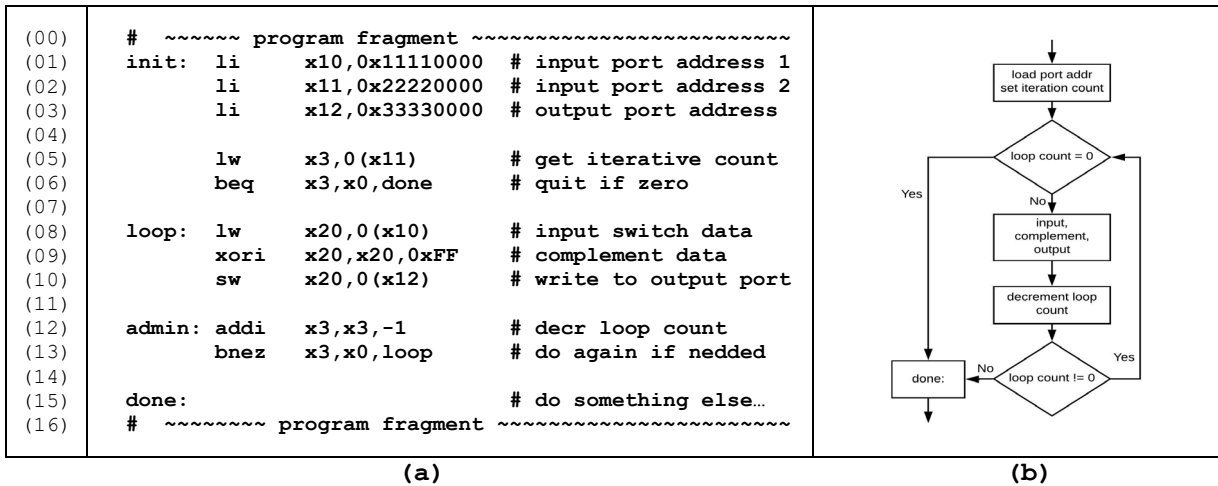


Figure 10.18: An alternative approach to a while-loop with an unknown numbers of iterations.

10.5.2.2 Do-While Loops

The do-while loop is the other type of iterative loop. The main characteristic of the do-while loop is that the at least one iteration of the loop is guaranteed to be executed. After that first execution of the loop body, the do-while loop effectively becomes a while loop after the first iteration complete. Keep in mind that there is a potential problem with do-while loops that have an unknown number of iterations at runtime. If you need to do something “zero times”, and you do it once, your program could die an ugly death. For these cases, you may want to make sure you code does the “safe thing” if this condition could potentially occur.

The fragment of code in Figure 10.20 shows an example of a do-while loop with a known number of iterations. Even though this code does not do anything meaningful, here are a few fun facts to see in this example:

- We included an “init” label as a comment; the instruction on lines (01-04) initializes registers that the code below it uses.
- The body of the loop is on lines (06-08). Note that the program always executes these three lines at least once.
- The loop administration starts on line (10) and includes line (11), which we like to indicate using the “admin” label. We first decrement the loop count and then jump back to do another loop iteration if that count is not zero. If the loop count is zero, then the condition fails and program execution drops through to the instructions starting at the “done” label (so pretend there are some instructions there).
- We’ve included a charming flowchart that models the code for your viewing pleasure.

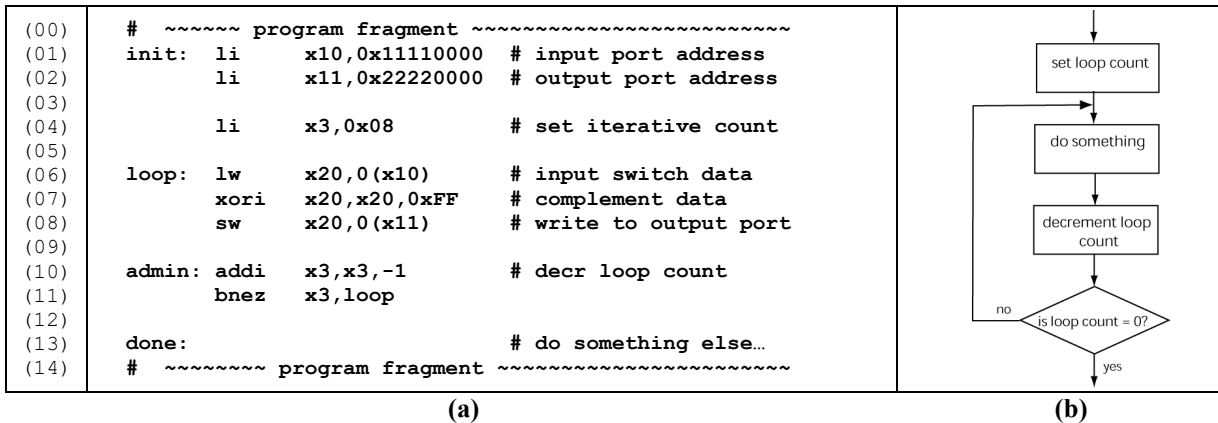


Figure 10.19: An example of a do-while loop with a known numbers of iterations.

Figure 10.20 shows an example of a do-while loop that iterates an unknown number of times. Because the accumulation of the input value determines how many times the loop iterates, we don't know what that value is. Here are a few other things to notice about Figure 10.20.

- The code uses an accumulator, so part of initialization of this fragment is to use a `mv` instruction to clear a register to use as an accumulator, which we do on line (03).
- The body of the loop is to input a value from the outside world and accumulate it, which we do lines (05-06). These lines always execute at least one time based on the notion we've modeled this code as do-while loop.
- The loop administration for this loop is on line (08); it comprises of a check to see if the accumulated value has surpassed the arbitrary threshold we created on line (02). If the threshold has not been exceeded, the loop does at least one more iteration (the branch is taken); otherwise, the branch is not taken and program flow continues on to the code below the "admin" label.

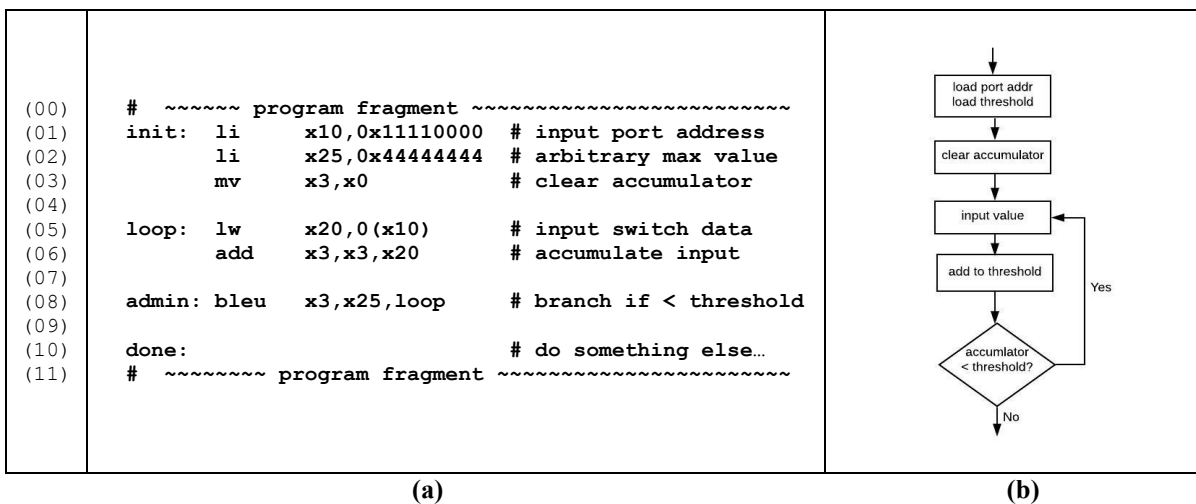


Figure 10.20: An example of a do-while loop with an unknown numbers of iterations.

10.5.3 Iterative Construct Off-By-One Issues

Off-by-one issues essentially means that you mean think your code executes a loop X number of times, but the loop is actually executing either $X+1$ or $X-1$ number of times. Off-by-one errors are particularly common in assembly language programming because coding at such low levels forces you to have a 100% understanding of

how the instruction actually work⁸. Once again, some loops are easy: you've coded a million of them, you never make a mistake. Coding these loops is easy because understanding the loop parameters are not too complicated. However, when you need to code a loop construct that does something somewhat special, you're more apt to make a mistake⁹.

In real life, it makes no sense to “decrement the loop count” before you started executing the loop. Often times in computerland, you must examine something that is “zero-based”, and you're using the iterative count as an index. For example, you want to examine the first ten locations in main memory. The first ten locations in a start at address zero and continue to address nine (0-9). It's easy to exit the loop on the ninth iteration before you do what you need to do in the loop, meaning you only iterated the loop nine times instead of ten. Keep this idea in mind; it may save your arse someday.

Example 10.7: Iterative Construct with a Known Iteration Count

Write a RISC-V assembly language program that continuously does the following: the program inputs a word of data from port address 0x11000004, divides that data by 64, then outputs the data to port addresses 0x11000008.

Solution: This is a simple program that we'll do in two different ways to show some of “the possibilities”, just because we can program necessarily uses a loop construct; Figure 10.21 shows all the gory but interesting details for the first approach.

- The program first inputs data on line (10). We then plan on using a do-while loop that iterates six time and divides the input data by two each time through the loop. To do this, we load a register with the iteration count (six) on line (12). The loop divides the input value by two each iteration using a `slri` instruction that shift right one bit position.
- The administrative part of the loop include decrement the iteration count on line (14) and then checking the condition using a conditional branch on line (15). We output the final divided value on line (17).

```
(00) #-----
(01) # Program Description: This program inputs word from port address 0x11000004
(02) # then divides the value by 64, then outputs the data to port address
(03) # 0x11000008.
(04) #-----
(05) .text                                # instruction code goes in text segment
(06)
(07) init:  li    x15,0x11000004          # input port address
(08)       li    x16,0x11000008          # output port address
(09)
(10) main:  lw    x20,0(x15)              # input count data
(11)
(12)       li    x10,6                    # load iteration count
(13) loop:  slri  x20,x20,1               # divide by two
(14) admin: addi  x10,x10,-1              # decrement iteration count
(15)       bne   x10,x0,loop              # check loop count
(16)
(17)       sw    x20,0(x16)               # output data
(18)       j     main                      # rinse, repeat
```

Figure 10.21: The solution to this example problem.

⁸ Which is not as true for higher-level languages (or at least it requires a different sort of understanding).

⁹ Coding loop constructs become so second nature, that you forget why it is the loop you coded actually works properly. Then when you have a special loop to code, meaning a loop that is not as straight-forward as all the other loops you coded, you have to really be careful because you've forgotten how the loops actually work. Welcome to assembly language programming.

Figure 10.22 shows the better solution. Note that there is no loop; we can use one `slli` instruction on line (11) to complete the division. The cool thing to note here is that the RISC-V shift-type instructions can operate as barrel shifters of any with from 1 to 32 bit locations (inclusive). The code itself contains fewer instructions, including not configuring of the loop count for the do-while loop. Be sure to always take advantage of such handiwork. The moral of this story is that the second solution does not requires a loop construct, which enable the program to be run more efficiently, and thus do more inputting/outputting.

```

(00) #-----
(01) # Program Description: This program inputs word from port address 0x11000004
(02) # then divides the value by 64, then outputs the data to port address
(03) # 0x11000008.
(04) #-----
(05) .text # instruction code goes in text segment
(06)
(07) init:  li    x15,0x11000004 # input port address
(08)       li    x16,0x11000008 # output port address
(09)
(10) main:  lw    x20,0(x15)     # input count data
(11)       slli  x20,x20,6      # divide by two, six time (divide by 64)
(12)       sw    x20,0(x16)     # output data
(13)
(14)       j     main           # rinse, repeat

```

Figure 10.22: The solution to this example problem.

Figure 10.23 shows two flowcharts representing the two solutions to this example problem. The flowchart in Figure 10.23(a) shows a decision box that models the do-while loop in the first solution; the decision box is gone in the second solution because the required shift operation completes in one instruction.

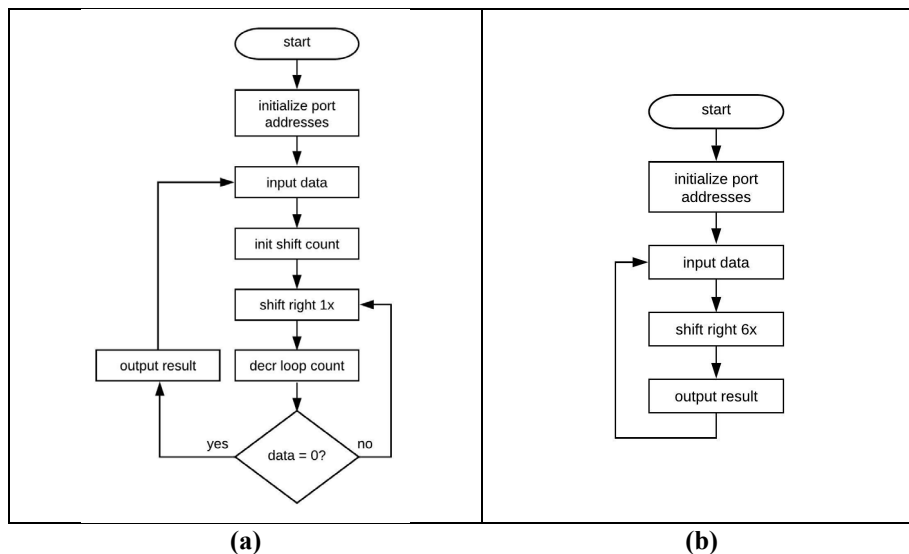


Figure 10.23: Flowcharts to the first (a) and second (b) versions of the solution to this example.

Example 10.8: Iterative Loop with Unknown Count

Write a RISC-V assembly language continuously does the following: the program inputs two pieces of data from port addresses 0x11004000 & 0x11004004, respectively. The first piece of input data is a byte and represents a count value. The second piece of data is a halfword. The program then outputs the input data to port address 0x11008000 for as many times as is represented by count inputs. The output data is also work data.

Solution: This program necessarily uses a loop construct; Figure 10.24 shows all the gory but interesting details:

- The program requires an iterative loop, which we know because we needs to do something a given number of times. We have a choice of what type of loops to use for this problem, so we must make that decision. The guiding factor in this problem is that the iterative count, which we input, could be zero. This means that we need to check the value before we execute the body of the loop (meaning we can't use a do-while loop for this problem). Note since the iteration count is input when the program runs, we don't know in advance (at assembly time) what that value is, thus this construct can iterate a variable number of time.
- The initialization part of the program includes placing port addresses into registers, which we do on lines (07-09).
- We input the count data on line (11) and the actual data on line (12). The problem stated the count data and actual data to be in byte and halfword form, respectively, which is why we use the **lb** and **lh** instructions.
- The loop construct (while-loop) checks the condition on line (20) before continuing with the loop. The body of the loop is the data output instruction on line (16).
- All loop constructs require some type of "loop administration", which we do in this loop starting on line (18) with a decrement of the loop count. We use an **addi** instruction with a source register of "-1" to handle the decrement; the instruction officially adds "-1" to the current value of the loop count in x20. The other part of the loop administration is an unconditional branch back to the conditional branch instruction and determines if the loop should continue or not.

```

(00) #-----
(01) # Program Description: This program inputs a count and data; the data
(02) # is then output for as many times as is in the count. The program
(03) # does this continuously.
(04) #-----
(05) .text                                # instruction code goes in text segment
(06)
(07) init:  li    x15,0x11004000           # input port address
(08)        li    x16,0x11004004           # output port address
(09)        li    x17,0x11008000           # output port address
(10)
(11) main:  lb    x20,0(x15)               # input count data
(12)        lh    x21,0(x16)               # input data to output
(13)
(14) loop:  beq   x20,x0,main              # Check count value; start again if zero
(15)
(16)        sh    x21,0(x17)               # output data
(17)
(18) admin: addi   x20,x20,-1              # decrement iteration count
(19)        j     loop                      # repeat output part, not input part

```

Figure 10.24: The solution to this example problem.

Figure 10.25 shows a flowchart modeling this example. The important item to notice is that the flowchart reflects a while loop in that the program checks the input count condition before it proceeds. This supports the notion that the count condition that the program inputs could be zero.

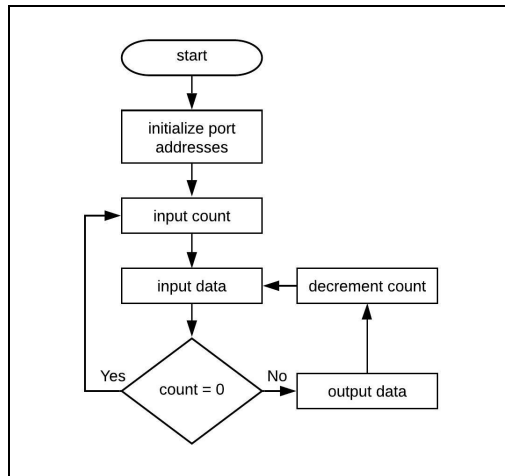


Figure 10.25: A flowchart modeling the operation of this example program.

Example 10.9: Iterative Loop with Known Count

Write a RISC-V assembly language continuously does the following: the program inputs ten pieces of data (unsigned data guaranteed to be non-zero) from port address 0x1100A000 and sums that data. The program then divides the sum by two as many times as it requires to ensure the data is less than 0x47. The program outputs the final value in to port address 0x11009800. Assume the input data never overflows a 32-bit register.

Solution: This program necessarily uses two different loop constructs; Figure 10.26 shows the solution and the following describes its amazing glory:

- The program has the typical initialization stuff, but now there is more. First, on line (11) we put the value 0xr47 into a register because the conditional branch instructions work using register values and not immediate values. Second, on line (12) we initialize the loop count to ten as the program requires. Third, since this program keeps a running total of input values, we use a register as an accumulator and clear that register on line (13). The ordering of these three instructions is important; the value of 0x47 never changes, but we must “re-initialize” the iteration count (x17) and the accumulator (x18) each time we start processing another ten pieces of data. There are a few ways to do this, but to save two instructions somewhere else in the program, we’ll be jumping back to the “start” label when the program completes processing of all ten pieces of data.
- We use a do-while loop to input the data since we know how many times we’ll be iterating that loop (ten), which means there is always a first time. The “do” of the do-while loop starts on line (15) by inputting data, then adds that data to the running total (the accumulator) on line (16). The administrative part of the loop starts on line (21) by decrementing the iteration count and continues with a check of the iteration count on line (19). If the loop needs to continue, program control transfers back to line (15); otherwise it drops through to line (21).
- The code on lines (21-23) is a while-loop with an unknown iteration count. We may need to divide the sum by two many times (line (22)), or we may not need to divide the sum at all. We use a srli (shift right logical immediate) instruction to divide by 2. We unconditionally jump on line (23) to check the condition after each division.
- We output the final value on line (25) and then jump back to the start label, which we previously commented on.

- Note that we've simplified the reading of the program by using whitespace (blank lines) to delineate different "sections" of the program; the instructions in a given section are all supporting the same task and are different from the previous or next section.

```

(00) #-----
(01) # Program Description: This program continuously inputs and sums ten pieces
(02) # of data (unsigned words guaranteed to be non-zero) from port address
(03) # 0x1100A000. The program then divides the sum by two as many times as
(04) # it requires to ensure the data is less than 0x47. The program outputs
(05) # the final value in to port address 0x11009800.
(06) #-----
(07) .text # instruction code goes in text segment
(08)
(09) init:  li   x15,0x1100A000 # input port address
(10)      li   x16,0x11009800 # output port address
(11)      li   x18,0x47      # load threshold value
(12) start: li   x17,10       # iteration count
(13)      mv   x30,x0        # clear counter
(14)
(15) main:  lw   x20,0(x15)    # input data
(16)      add  x30,x30,x20    # accumulate input data
(17)
(18) admin: addi  x17,x17,-1  # decrement iteration count
(19)      bne  x17,x0,main    # branch if iteration count non-zero
(20)
(21) loop:  ble  x30,x18,out_val # jump to output data if less than 0x47
(22)      srli x30,x30,1      # divide by two
(23)      j    loop          # jump to check again
(24)
(25) out_val: sw  x30,0(x16)   # output data
(26)      j    start         # repeat entire process again

```

Figure 10.26: The solution to this example problem.

Figure 10.27 shows a flowchart that models the assembly language solution to this example. The program had both a do-while loop and a while loop, which we see with the two decision boxes in Figure 10.27. Recall there is one decision both for each the do-while and while loop.

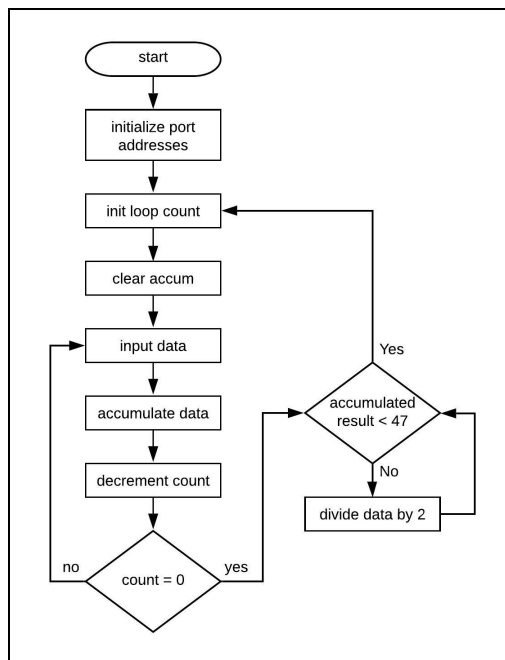


Figure 10.27: A flowchart modeling the operation of this example program.

Example 10.10: Gathering Input Statistics

Write a RISC-V assembly language continuously does the following: inputs 20 unsigned word values from port address 0x1100F000 and counts how many of those data are less than 0x58 and less than 0xA4. After the ten values are input, the program outputs the two counts as bytes to port address 0x1100E000 in two consecutive outputs.

Solution: This is a rather contrived program in that it does not do much and there are many approaches to solving this problem. Our intent is to use the set-if-less-than-type instructions, so Figure 10.28 shows our choice of solutions along with the gut-wrenching details:

- The has initialization code loads the I/O addresses into registers, and also places one of the threshold values to check in a register also (line (10)). We don't put both less than threshold values into registers simply to prove a point that we'll describe in a later comment.
- The code at the "start" label is also initialization code but differs from the previous initialization code in that we need to execute the code on lines (12-15) each time the program does its main task. The body of the loop changes the values in the second set of initialization code, so it needs to be re-initialized; the program never changes the register values loaded by the first three instructions.
- We use a do-while loop to input the data since we know how many times we'll be iterating that loop (20), which we'll always enter the loop for the first time. The "do" of the do-while loop starts on line (15) by inputting data. We then use two slt-type instructions on lines (18-19) to examine the two less-than threshold values. We use a register-type instruction on line (18) and an immediate-type instruction on line (19). This is somewhat arbitrary just to show we can do it. The only useful comment here is that the sltiu instruction uses a constant value that is set at assemble-time while the sltu instruction uses a register that the program can change at any time. This program never changes the value but it's good to know it's possible.
- The algorithm works by using the register values set by the slt-type instructions as incrementing values on lines (20-21). The values in these registers is either '1' (less-than checks true) or '0' (less-than checks false), so incrementing in this case is a viable approach.
- Lines (23-24) contain the loop administration code including a decrement of the loop count and a conditional branch back to the "main" label to continue the loop.
- The program completes by outputting the two counts on lines (26-27), then unconditionally branching to the start label to redo the entire program (except for the first three instructions).

```

(00) #-----
(01) # This program continuously does the following: inputs 20 words values
(02) # (unsigned) from port address 0x1100F000 and counts how many of those
(03) # data are less than 0x58 and less than 0xA4. The program outputs the two
(04) # counts to port address 0x1100E000 as bytes in two consecutive outputs.
(05) #-----
(06) .text                                # instruction code goes in text segment
(07)
(08) init:  li    x15,0x1100F000          # input port address
(09)        li    x16,0x1100E000          # output port address
(10)        li    x17,0x58                # load threshold value
(11)
(12) start: li    x18,20                  # iteration count
(13)        mv    x30,x0                  # clear iteration counter
(14)        mv    x10,x0                  # clear less than 0x58 counter
(15)        mv    x11,x0                  # clear less than 0xA4 counter
(16)
(17) main:  lw    x20,0(x15)              # input data; start of do-while
(18)        sltu  x25,x20,x17            # check < 0x58
(19)        sltiu x26,x20,0xA4           # check < 0xA4
(20)        add  x10,x10,x25             # add to < 0x58 count
(21)        add  x11,x11,x26             # add to < 0xA4 count
(22)
(23) admin: addi  x18,x18,-1              # decrement iteration count
(24)        bne  x18,x0,main              # branch if iteration count non-zero
(25)
(26) store: sb   x10,0(x16)              # output less than 0x58 count
(27)        sb   x11,0(x16)              # output less than 0xA4 count
(28)        j    start                    # jump to do all over again

```

Figure 10.28: The solution to this example problem.

The slt-type instructions in this program allowed us to essentially make a comparison but not branching as a result of that comparison. An action like this is quite handy when you need it, but programs typically use it less often than other branch-type instructions. Try not to forget about slt-type instructions.

Figure 10.29 shows one possible flowchart that models the solution. We're getting to the point with our flowcharts that we're not trying to represent every action in detail; we're now more interested in the overall form of the flowchart. Keep in mind that the problems we've done so far are not overly complicated; the real usefulness of flowcharts comes with problems that require complex algorithms to complete.

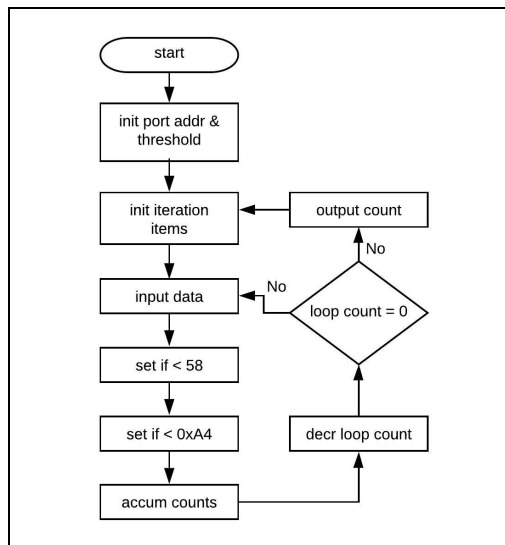


Figure 10.29: A flowchart modeling the operation of this example program.

10.6 Bit Manipulations for MCUs

We design MCUs, or microcontrollers as some people call them, to do exactly what their name implies: they control things. By things, we mean external computer peripherals. In general, MCUs monitor status inputs and send out control outputs. Recall that this functionality is similar to FSMs, but MCUs control things via software (or firmware) while FSM are purely a hardware-oriented device. The way MCUs control things is by interpreting input bits (status) and then send out output bits (control); thus, bit manipulations are a key element in writing meaningful programs for MCU. This section describes some of the details regarding the finer points of bit manipulations.

10.6.1 Tweaking Bits

Bit-tweaking, or *bit banging*, is a well-known assembly language trick. In this context, we use the word “tweaking” to mean modifying individual bit values. There are only four things you can do with a bit: 1) setting, 2) clearing, 3) toggling (complementing), and, 4) holding the bit value (doing nothing). The notion of tweaking bits is slightly misleading because MCUs such as the RISC-V MCU have instructions that typically only operate on complete register values. More specifically, the three logic-type instructions (AND, OR, and EXOR) operate on the entire register, which is why we refer to them as bit-wise operations. Although many modern MCUs have instruction that can perform logic-type operations on individual bits of a register, most do not.

Having instructions that manipulate individual bits are handy but they make the instruction set larger than they need to be and the hardware more complex than they need to be, which is why most instruction sets don’t have such instructions. One of the unstated requirements of assembly language programming is that you need to be clever. In other words, you need to work with the instructions you have to do what you need to do with a reasonable amount of complexity. You won’t always have the exact instruction you need every time but the instruction set should always have the functionality to create the operation you’re looking for (although it may take a few instructions instead of just one instruction).

You can do four things with a bit: set it, clear it, toggle it, or hold it (do nothing with it). Table 10.21 shows the four possible things you can do with a bit as well as the logic operations you use to do those four things. There are many ways to “hold” bits; Table 10.21 shows three of the more common approaches.

Bit Operation	How to do it
setting	Logical OR with ‘1’
toggling	Logical XOR with ‘1’
clearing	Logical AND with ‘0’
holding (do nothing)	Logical OR with ‘0’ Logical XOR with 0 Logical AND with ‘1’

Table 10.21: The four possible things you can do to a bit and how to do it with logic operations.

10.6.2 Bit Masking

Since we generally use MCU to control various computer peripheral devices, it would make sense that we can use single bits to control these devices rather than entire bytes. The issue with most MCUs is that they can only operate on large chunks of data at a time. The result is that your assembly programs typically require the use of *bit-masks* in order to manipulate individual bits in a register. The bit-masks, combined with executing conditional branch instructions, allows the microcontroller to perform different functions based on the status of individual bits of registers rather than the entire register. As you can probably imagine, bit-masking is really useful and common in assembly languages. Table 10.22 shows a few examples regarding bit-mask possibilities, all of which makes more sense when you see it used in a few examples.

There are two main uses for bit masks: 1) checking individual input bits in a register, and, 2) assigning values to individual bits in a register. Note that programs check individual bits and use those bit values in control the program flow in programs. In most cases, your MCU is controlling some peripheral device, which means your MCU is typically reading status inputs from external devices and assigning control outputs to external devices.

For the RISC-V MCU, we input values from the outside world into a register, where we can then “check” them; we then output values to the outside world from a register.

Example	Explanation
<code>ori x10,x10,0x02</code>	Sets second to right-most bit-1 in r1 (no other bits change)
<code>ori x11,x10,0x0F</code>	Sets four LSB in x10; store result in x11 (no other bits change)
<code>andi x12,x12,0x0F</code>	Clears all but the four LSBs in x12 (no other bits change)
<code>andi x13,x13,0x0FF</code>	Clears all but the eight LSBS in x13 (no other bits change)
<code>xori x14,x14,0x03</code>	Toggles the lower two bits in x14 (no other bits change)
<code>xori x15,x15,0x0F</code>	Toggles the lower four bits in x15 (no other bits change)

Table 10.22: Examples of bit-masking operations.

Example 10.11: Bit-Masking and Bit Setting

Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that reads data from the switches. Consider the data on the switches to represent a 16-bit unsigned number. If that number is greater than 255, then the program sets the bottom four bits of the data before it outputs it to the LEDs. Otherwise, the program outputs the input value. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.

Solution: Most of this solution is similar to the previous solution so we’ll only describe the interesting parts of this solution. Figure 10.30 shows the full solution; the following blather could prove interesting:

- The main trick in this problem is how we check to see if the value is greater than 255. We could simply load a register with the value 255 and use that register in a conditional branch statement, but that approach would require that we use an extra register. Our approach is to clear the bottom byte in the input value; if the resulting value is non-zero, then the input value had bits set above the seventh bits from the right, which means the input value is greater than 255. Part of our initialization code set a mask value into a register (line (09)); we use this value to clear the lower byte of the input on line (12). The mask value is greater than 12-bits (meaning we can’t define the mask using 12-bits) so we place the value in a register and use an **andi** instruction on line (12).
- We input the data on line (11), mask it on line (12), and branch to the output if the input value is 255 or less using a **beq** instruction on line (13).
- If the value is greater than 255, program flow drops to the **ori** instruction one line (14); this instruction sets the lower 4-bits of the value to 0xF. Note that some of these values may have already been 1’s, but that does not matter. The instruction does not know what is in the register; it sets the lower four bits no matter what is in the register.
- The program outputs the value to the LEDs on line (16) and branches unconditionally to do it all again on line (17).
- The main structure of this program is an if/else statement; the program repeats itself using the unconditional branch on line (17).

```

(00) #-----
(01) # This program reads data from the switches; if the data is greater than
(02) # 255, then the program sets the lower four bit of the value before outputting
(03) # it. Otherwise, it outputs the value to the LEDs without changing value.
(04) # The port address of the switches is 0x1100C000; the port address of
(05) # the LEDs is 0x11008000. Assume 16 switches and the same number of LEDs.
(06) #-----
(07) init:    li    x10,0x1100C000    # put switch address (input) to register
(08)         li    x11,0x11008000    # put LED address (output) in register#
(09)         li    x15,0x0000FF00    # bit mask for values > 255
(10)
(11) main:   lhu   x30,0(x10)        # input data
(12)         and   x31,x30,x15       # mask the lower byte
(13)         beq   x31,x0,out_val     # jump to output (input greater than 255)
(14)         ori   x30,x30,0xF       # set lower four bits
(15)
(16) out_val: sh   x30,0(x11)         # send value to LEDs
(17)         j     main               # rinse, repeat

```

Figure 10.30: The solution to this example problem.

Example 10.12: Parity of a Switches

Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that continually reads data from the switches and outputs the parity of the input data to the LEDs. Specifically, no LEDs on indicates the parity of the input was even; the right-most LED on indicates odd parity of the input. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.

Solution: Some of this solution is similar to the previous solution so we'll only describe the new parts. This is a classic problem that we'll see again in a later chapter. This is our first program that sort of does something meaningful in that parity is popular topic and we use a clever algorithm to generate our result; Figure 10.31 shows the incredibly interesting details.

- The initialization part of the program includes loading the port addresses to registers, and putting the LEDs into a known state. We do all this on lines (06-09).
- After inputting data on line (11), we check to see if that data is zero; if it is zero, we branch to the code that outputs to the LEDs; otherwise we continue into the code. We wrote the algorithm as while loop because we checked the condition first. The structure of the code is an embedded loop, where one where we essentially have a do-while loop embedded in the while loop. The while-loop is the outer loop and starts on line (11); the do-while loop is the inner loop and is on lines (13-18). This is a classic structure you use quite often in assembly languages.
- The algorithm masks all but the LSB of the input data (line (14)) and then adds that result to x20, which is a register we use as a counter. Note that part of the init code we clear that counter and then clear it again after we complete the inner loop and output the result to the LEDs on line (22). The administrative part of the inner loop includes shifting the original input data to the right. Note that we treat the outer while-loop as if we don't know the iteration count; we could have used an algorithm that used an iteration count of 16, but that would mean the inner loop would always run 16 times. The way we've written the algorithm, the loops ends when that value we're using to calculate parity runs out of 1's.
- The final portion of the algorithm is when we break out of the inner loop. At that point we mask 1's count on line (20) and output that value on line (21). In this way, if the count of 1's is odd, a '1' in the LSB position is output to the LEDs; otherwise the algorithm turns off all LEDs.

- In preparation of starting the outer loops again, we clear x20, which use to keep track of the set bits in the input data.

```

(00) #-----
(01) # This program inputs data from the switches, calculates the parity of
(02) # of the input data, then outputs that parity to the LEDs with no LEDs
(03) # on indicating even parity and the right-most LED on indicating odd
(04) # parity.
(05) #-----
(06) init:    li    x10,0x1100C000    # put switch address (input) to register
(07)         li    x11,0x11008000    # put LED address (output) in register#
(08)         mv    x20,x0           # clear register to use for LED state
(09)         sh    x20,0(x11)        # turn off all LEDs
(10)
(11) main:    lhu   x30,0(x10)        # input data
(12)
(13) loop:    beq   x30,x0,done        # branch if zero
(14)         andi  x21,x30,1          # mask LSB
(15)         add   x20,x20,x21        # accumulate bits
(16)
(17) admin:   srli  x30,x30,1         # shift right one bit
(18)         j     loop              # jump to keep counting
(19)
(20) done:    andi  x20,x20,1          # mask LSB
(21)         sh    x20,0(x11)        # output result
(22)         mv    x20,x0           # clear counter
(23)         j     main              # rinse, repeat
(24) #-----

```

Figure 10.31: A codespace efficient solution to this example.

Figure 10.32 shows a flowchart modeling the operation of this program. There are many ways to draw flowcharts; we typically draw the ones in this text to save vertical space. There are sometimes better ways to draw them, but making the flowcharts as “horizontal” as possible reduces page count issues. One issue to notice in the flowchart of Figure 10.32 is the fact that there are two “clear accum” boxes there. We could have structured our code to use only one such box, which would definitely have a few minor advantages.

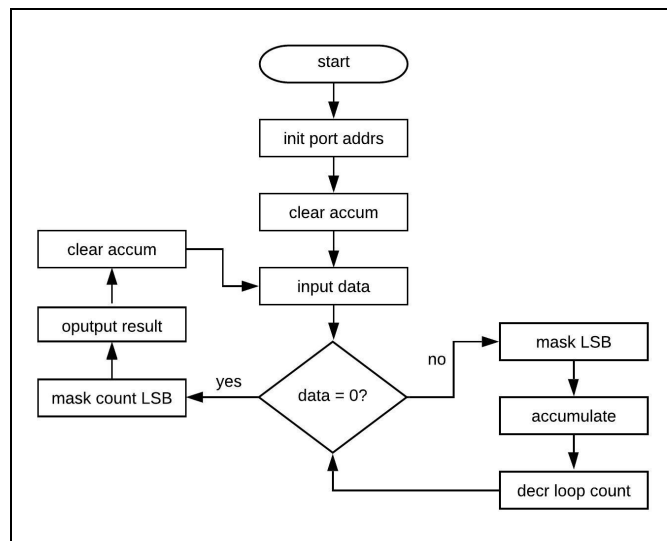


Figure 10.32: A flowchart modeling the operation of this example program.

Example 10.13: Simple Bit Mask with Conditional Output

Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that reads data from the switches. If the second to right-most switch is on (on=1), then the program turns on all LEDs; otherwise the program turns off all LEDs. The port address of the switches and LEDs is 0x1100C000 & x11008000, respectively.

Solution: The most interesting part of this example is the notion we have to act on the status of a single switch. This is somewhat of a problem because the RISC-V MCU only operates on larger chunks of data such as bytes, halfwords, and words. This where the notion of bit masking comes in. Figure 10.33 shows the full solution; here are some fun facts about the solution:

- The solution contains a nice program header on lines (00-06) that describes what the program does with a modest amount of detail. All programs should have such a header (or file banner); we sometimes omit them to save time.
- The program first initializes stuff on lines (7-10); we use the “init” label to indicate this. The initialization code is typically code that we execute only once in a program. We use this code to essentially “create” constants in the code. This notion is that we set the constant values once, and then save codespace and execution time by not setting them again in the remainder of the code. For this initialization code, we put the two I/O addresses in registers; we also put the two chunks of output data in registers (all 1’s and all 0’s).
- The main code consists of inputting data, and if/else construct, and a few strategically placed unconditional branches. We’ve noted the main code using the “main” label on line (12). The unconditional branch is on line (17). The if/else construct spans lines (12-17).
- The problems states that the hardware only has 16 switches (inputs) and 16 LEDs (outputs). Because of this, the problem opts to use **lh** instructions for inputting data and **sh** instructions for outputting data. We could have used **lw** and **sw** instructions and the code would have worked, but using **lh** and **sh** is a better option because it better reflects the actual size of the data involved. Additionally, we could have used **lhu** for inputting data as well.
- We first input the data on line (12). We’re only interested in the second to right-most bit, so we mask all the other bits using the **andi** instruction one line (13). The mask on this instruction (mask = 2) clears all the bits except the bit we’re interested in (recall that the assembler represents 2 as 0x002, or 000000000010₂; the underlying RISC-V MCU hardware extends the instruction length from 12 to 32 bits in order to fill the register with known values). The result from the AND operation leaves either a zero or two in x20. If the result is two, that means the switch was on when the program executed the **lhu** instruction. If the result is 0, that means the switch was off.
- The **beq** instruction on line (14) is part of the if/else construct. This instruction checks to see of the result of the mask operation was zero or not, and branches accordingly. If the value in x20 is zero (the second to right-most switch was off), the code takes the branch and the instruction on line (19) is the next instruction to execute; otherwise, the code does not take the branch and the following instruction (line (16)) is the next instruction to execute.
- We structured the code such that we need two unconditional branch instructions: one for the if clause and another for the else clause. We could have done this another way that we’ll list after this code.

```

(00) #-----
(01) # This program reads data from the switches; if the second to right-most
(02) # switch is on (on=1), then the program turns on all LEDs; otherwise
(03) # the program turns off all LEDs. The port address of the switches is
(04) # 0x1100C000; the port address of the LEDs is 0x11008000. Assume
(05) # there are 16 switches and an equivalent number of LEDs.
(06) #-----
(07) init:    li    x10,0x1100C000    # put switch address (input) to register
(08)         li    x11,0x11008000    # put LED address (output) in register
(09)         li    x8,0xFFFF         # load reg with one output value
(10)         mv    x9,x0             # load reg with other output value
(11)
(12) main:    lhu   x20,0(x10)        # input data
(13)         andi  x20,x20,2         # mask 2nd to right-most bit
(14)         beq   x20,x0,out_off    # if not zero, branch to off
(15)
(16) out_on:  sh    x8,0(x11)         # turn on all LEDs
(17)         j     main              # do it again
(18)
(19) out_off: sh    x9,0(x11)         # turn off all LEDs
(20)         j     main              # do it again

```

Figure 10.33: The solution to this example problem.

Figure 10.34 shows an alternative solution to this example. We include this solution because it handles the output in a different manner. The previous solution had two different output instructions (**sh**); this solution only has one output instructions. This solution assigns the output value to a generic register; both the if and else clause assign a value to that register. The code in this alternative solution is actually longer, but this is a good programming form to know about. This example problem only had two possible outputs; if there were 20 different possible output values, the structure of this alternative solution would be clearly more space efficient.

```

(00) #-----
(01) # This program reads data from the switches; if the second to right-most
(02) # switch is on (on=1), then the program turns on all LEDs; otherwise
(03) # the program turns off all LEDs. The port address of the switches is
(04) # 0x1100C000; the port address of the LEDs is 0x11008000. Assume
(05) # there are 16 switches and an equivalent number of LEDs.
(06) #-----
(07) init:    li    x10,0x1100C000    # put switch address (input) to register
(08)         li    x11,0x11008000    # put LED address (output) in register
(09)         li    x8,0xFFFF         # load reg with one output value
(10)         mv    x9,x0             # load reg with other output value
(11)
(12) main:    lhu   x20,0(x10)        # input data
(13)         andi  x20,x20,2         # mask 2nd to right-most bit
(14)         beq   x20,x0,set_off    # if not zero, branch to off
(15)
(16) set_on:  mv    x30,x8            # load register with 1's
(17)         j     out_val           # do it again
(18) set_off: mv    x30,x9            # load register with 0's
(19)
(20) out_val: sh    x30,0(x11)        # turn off all LEDs
(21)         j     main              # do it again

```

Figure 10.34: An alternative solution to this example problem.

Figure 10.35 shows the two flowcharts for the two solution to this example. The flowcharts are the same up to the point of outputting data. The alternative solution in Figure 10.34 contains only one output instruction, which results in one less process block for the flowchart in Figure 10.35(b) representing the alternative solution. The two programs, however, are indeed functionally equivalent.

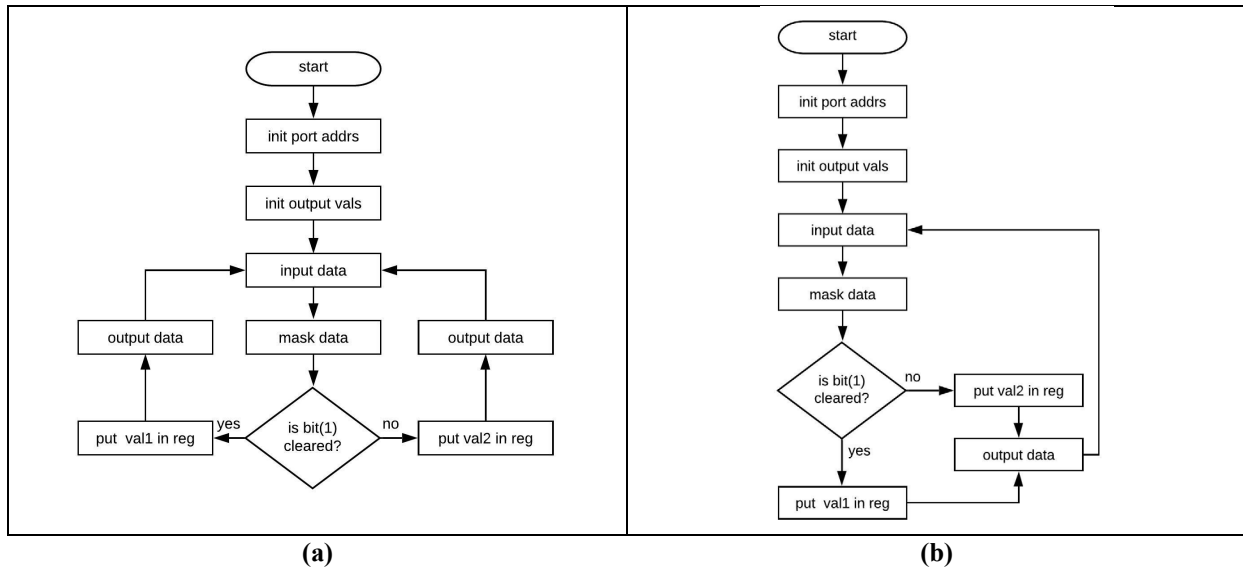


Figure 10.35: Flowcharts modeling the two solutions to this example.

Example 10.14: Bit-Masking with Blinking LED

Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that reads data from the switches. If the two right-most switches are off (on='1'), the program toggles the right-most LED; in all other cases, the program does not change the state of any LED. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.

Solution: Most of this solution is similar to the previous solution so we'll only describe the interesting parts of this solution. Figure 10.36 shows the full solution; here is some pertinent commentary:

- Anytime you're working with output such as LEDs, it's always a good idea to put the LEDs in a known state. We do this despite the problem statement saying nothing about it. Line (09) clears a register; line (10) writes that value to the output port address controlling the LEDs. These two lines effectively turn off all the LEDs as part of the initialization sequence so there no question as to the state of the LEDs when the program exits the initialization code. We're effectively using register x9 to save the current state of the LEDs, which is a standard assembly language programming approach.
- This program is going to do perform a compare to ensure the two LSBs are set. There are many ways to do this, but we'll do it in this problem by comparing the masked switch values to 3. This is why on line (11) we place 3 in a register.
- The program inputs data from the switches on line (13) and masks it on line (14). We use the value of "3" in the mask because the number 3 in binary has the two LSBs set; the result is that all the bits in the register other than the two right-most bits are cleared; the code masks the two right-most bits, which in this case means the `andi` instruction does not change them. We are able to use an immediate type instruction for the bit-mask because the value fits nicely into a 12-bit field, which is the upper limit for the `andi` instruction.

- Line (15) contains a conditional branch instruction (`bne`), which directs program flow back to the instruction associated with the `main` label if either of the two right-most bits are set. In the case where the two right-most bits are cleared, program flow drops to the instruction on line (17).
- If the code makes it to line (17) that means we need to toggle the right-most LED. We opted to save the state of the LEDs in `x9`, so to toggle an LED, we need to exclusive OR that particular bit with a '1', which we do on line (17) using an `xori` instruction. The `xori` instruction toggles the right-most value in the designated LED register but does not change any other bit in `x9`. The data in `x9` is then output to the LEDs on line (18) using a `sh` instruction.
- The program contains an initialization section followed by an if/else construct; the program looks a like a big pile of code, but it only really contains these two sections of code.

```

(00) #-----
(01) # This program reads data from the switches; if the two right-most
(02) # switches are off (off=0), then the program toggles the right-most LEDs;
(03) # otherwise the program does not change the state of the LEDs.
(04) # The port address of the switches is 0x1100C000; the port address of
(05) # the LEDs is 0x11008000. Assume 16 switches and the same number of LEDs.
(06) #-----
(07) init:    li    x10,0x1100C000    # put switch address (input) to register
(08)         li    x11,0x11008000    # put LED address (output) in register#
(09)         mv    x9,x0             # clear register to use for LED state
(10)         sh   x9,0(x11)         # turn off all LEDs
(11)         li    x20,3            # load compare value (two LSBs set)
(12)
(13) main:    lhu   x30,0(x10)       # input data
(14)         andi  x30,x30,3        # mask two right-most bits
(15)         bne  x30,x20,main      # branch if two LSBs are 1
(16)
(17) both_off: xori  x9,x9,1        # toggle right-most bit in LED register
(18)         sh   x9,0(x11)         # send value to LEDs
(19)         j    main             # rinse, repeat

```

Figure 10.36: The solution to this example problem.

Figure 10.37 shows a flowchart modeling the solution to this example. No comments here as the flowchart is strangely similar to previous examples.

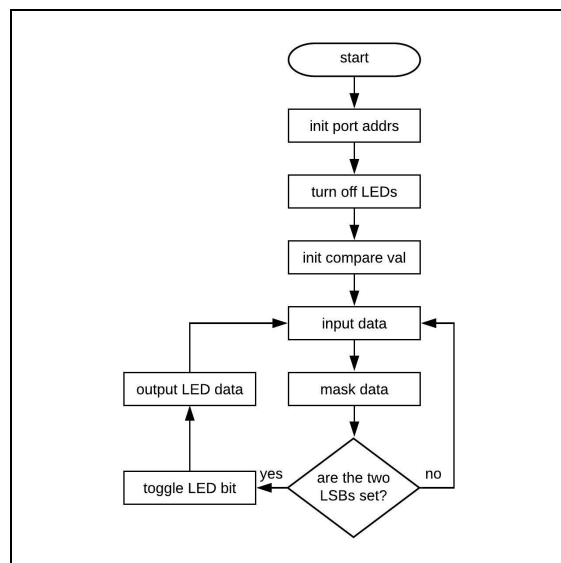


Figure 10.37: A flowchart modeling the operation of this example program.

10.7 Chapter Summary

- There are three types of bit-crunching instructions:
 - 1) logic (AND, OR, XOR)
 - 2) arithmetic (ADD, SUB)
 - 3) shift (right & left barrel shifts)
 - Program flow control instructions allow programs to execute instruction other than the “next” instruction in program memory. The types of program flow control instructions include jumps and branches, which offer unconditional and conditional program flow control, respectively.
 - Labels in program code provides jump and branch destinations used by the assembler, but also provide messages to human programmers without increase code space.
 - There are three primary programming constructs that form the bases of structured programming; all well-written programs can be decomposed into these three constructs.
 - 1) sequences
 - 2) if-then-else constructs
 - 3) iterative constructs (while & do-while loops)
 - Iterative constructs can have constant or variable iteration counts. The values of constant iteration counts are known at assemble time while the value of variable iteration counts are only known at run time.
 - There are four things you can do with a bit, 1) set it, 2) clear it, 3) toggle it, and 4) hold it. Assembly languages set bits by ORing them with ‘1’, clear bits by ANDing them with ‘0’, and toggling bits by EXORing them with ‘1’. Assembly languages hold bits in many different ways.
 - Bit-masking is the act of using logic instruction to operate on only a given set of bits (set, clear, toggle, hold). Most MCUs must use bit-masking because the MCU’s instructions operate only on larger chunks of data.
 - The RISC-V MCU has a set of instructions that don’t fall into other standard categories. These instructions include “set if less than” instructions, load address (**la**), load immediate (**li**), other data loading instructions (**auipc** & **lui**).
-

10.8 Chapter Exercises

- 1) In your own words, describe what the term “program flow control” means.
- 2) Briefly describe why you think it is that every assembly language program has at least one unconditional branch instruction.
- 3) Briefly describe why it is important to have an area in your program for “initialization code”.
- 4) Describe a situation where a “lesser” amount of code requires more execution time than a “greater” amount of code. Assume these sets of code perform identical tasks.
- 5) What does the term “bit-wise” mean in terms such as “bit-wise AND operation”?
- 6) What other type of logic operations are the other than bit-wise operations?
- 7) Briefly describe why you feel it is that most MCUs don’t have bit-level instructions despite the fact that the MCU typically is operating on bits rather than bytes.
- 8) List the two purposes of labels in assembly language programs.
- 9) Briefly describe why the RISC-V instruction set contains an add immediate instruction (**addi**) but not a subtract immediate instruction.
- 10) Briefly describe the two possible classification of conditions associated with do-while and while loops.
- 11) Briefly describe why it is that labels in programs do not increase program size.
- 12) What is another name for an unconditional branch operation?
- 13) Describe the four things you can do with a single bit.
- 14) There are standard approaches to setting, clearing, and toggling bits in assembly languages; provide examples of these approaches.
- 15) Name the three logic-based approaches to “holding” a bit value.
- 16) Briefly describe why the shift instructions in the RISC-V instruction set are actually barrel shifting instructions.
- 17) Briefly describe the only use for a nop pseudoinstruction?
- 18) Briefly describe the main function difference between a **mv** pseudoinstruction and a **li** pseudoinstruction.
- 19) List five ways the RISC-V assembler could implement a nop instruction.
- 20) In your own words, briefly describe what the term *bit masking* refers to.
- 21) Why do MCUs have a need for bit-masking?
- 22) Briefly explain why it is that most MCUs don’t have instructions that operate on the bit-level.
- 23) Briefly describe the primary advantage of “set if less than” instructions.
- 24) Will a do-while loop always iterate once when the iteration count is not known at run-time? Briefly explain.
- 25) Briefly describe the notions of “assemble time” and “run time”.

26) For the following RISC-V assembly language code fragment, if the label **init** has a value of 0x0000F104, provide the following information:

- a) What are the numeric values (in hex) of all the labels in the fragment
- b) What is the address in program memory of the **beq** instruction?
- c) What are the relative address of the following:
 - i. **cf2** relative to **loop**,
 - ii. **junk** relative to **loop**,
 - iii. **cf2** relative to **B_10**
 - iv. **B_10** relative to **init**

```

init:    mv    x15,x10      # save a copy
         li    x21,0x00000F00 # 100's bit mask
cf2:     li    x22,0x000000F0 # 10's bit mask
         li    x23,0x0000000F # 1's bit mask
         mv    x20,x0      # zero accumulator

B_10:    or    x15,x15,x21  # mask 100's nibble
         srli  x15,x15,8    # shift to lowest position
loop:    beq   x15,t_10     # go to tens if zero
         addi  x20,x20,100  # accumulate 100s
         sub   x15,x15,-1   # decrement loop count
         slti  x16,x17,0x34 # do something important

junk:    j     init        # do it again

```

27) For the following RISC-V assembly language code fragment, if the label **init** has a value of 0x00001B08, provide the following information:

- a) What are the numeric values (in hex) with all the labels in the fragment
- b) What is the address in program memory of the **beq** instruction?
- c) What are the relative address of the following:
 - i. **init** relative to **loop**,
 - ii. **pig** relative to **init**,
 - iii. **leave** relative to **done**
 - iv. **loop** relative to **restore**
 - v. **restore** relative to **loop**

```

init:    mv    x20,x0      # clear accumulator
         li    x15,32     # number to sum
         mv    x16,x10     # copy original address

loop:    beq   x15,x0,done  # leave if finished
         lw    x11,0(x10)  # get value from memory
         add   x20,x20,x11  # accumulate
         addi  x15,x15,-1   # decrement loop count
pig:     addi  x10,x10,4    # advance addr to next data
         j     loop        # done with iteration, do again

done:    srli  x20,x20,5    # divide by 32
restore: mv    x10,x16     # restore original x10 address

leave:   ret              # come on up to the house

```

10.9 Chapter Programming Exercises

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code looks exquisite in terms of readability
 - Fully comment your code
 - Don't worry about overflow issues unless the problem specifically state that you need to
- 1) Write a RISC-V assembly language program that continuously inputs a word of data from port address 0x11000C00, adds three to that data, then outputs the result to port address 0x11000E00. Don't worry about overflow issues.
 - 2) Write a RISC-V assembly language program that continuously inputs a word of data from port address 0x11001100, divides that data by two, adds eight to the result, then outputs the final the result to port address 0x11001200.
 - 3) Write a RISC-V assembly language program that continuously does the following: inputs a halfword of unsigned data from port address 0x11002200; if that data is less than 255, the data is doubled and output to port address 0x11003300; otherwise there data is halved and output to port address 0x11004400. Don't worry about overflow for this problem.
 - 4) Write a RISC-V assembly language program that continuously does the following: inputs ten words of unsigned data from port address 0x11001110, sums those inputs, divides the result by 4, then outputs the result to port address 0x11002220. Don't worry about overflow for this problem.
 - 5) Write a RISC-V assembly language program that continuously does the following: inputs a byte of data from port address 0x11003000; this byte represents the number of unsigned data words to input from port address 0x11004000. The input data is sum. The program then divides that input data by two enough times to make the data less than 0xFF. The final value is output to port address 0x1100AAA0 . Don't worry about overflow for this problem.
 - 6) Write a RISC-V assembly language program that continuously does the following: inputs 64 unsigned halfwords from port address 0x11000022 and outputs the average of the value to port address 0x11000066.
 - 7) Write a RISC-V assembly language program that continuously does the following: inputs a word from port address 0x1100FF00, the outputs the eight nibbles in that word (one nibble at a time as a byte value) to port address 0x1100EE00. Output the right-most nibble first and work towards the left.
 - 8) Write a RISC-V assembly language program that continuously does the following: inputs a word from port address 0x11002300; if more than 16 of the bits in that data are set, it outputs the number of set bits to port address 0x11002400; otherwise it outputs zero to the same port address. Both output values are unsigned bytes.
 - 9) Write a RISC-V assembly language program that continuously does the following: inputs 60 unsigned halfwords values from port address 0x11000066. The program counts the number of these values that are greater than 255 and evenly divisible by 16. The program then outputs the count to port address 0x11000077.
 - 10) Write a RISC-V assembly language program that continuously does the following: inputs a word from port address 0x11000F00, sums the eight nibbles in the word, and outputs the sum to port address 0x11000E00 as an unsigned halfword.
 - 11) Write a RISC-V assembly language program that continuously does the following: inputs a word from port address 0x11000F00. This word should be eight valid BCD values. The program sums those values and outputs the result as an unsigned word if every nibble is a valid BCD value. If even one nibble is not a valid BCD value, the program outputs 0xFFFFFFFF. The output port address is s 0x11000E00.

- 12) Write a RISC-V assembly language program that continuously does the following: inputs 32 halfwords from port address 0x11002300 and outputs 16 words. Each word output is comprised of the two consecutive halfwords from the input, where the first value input is the lower 16-bits in the output word, and the next value input becomes the upper 16-bits in the input word. Output a word after inputting two halfwords. The port address for the output is 0x11002400.
 - 13) Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that continuously does the following: inputs a signed halfword from the switches and adds that value to a running total. If the running total is less than or equal to zero, the program turns all the LEDs off. Otherwise, the program outputs the lower 16-bits of the running total to eh LEDs. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.
 - 14) Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that inputs the switch value. If the number of switches that are on is one, the program outputs 0xFFFF to the LEDs; otherwise the program outputs the number of bits set as a binary number to the LEDs. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.
 - 15) Consider the notion that the RISC-V MCU is controlling a board with 16 switches and 16 LEDs. Write a RISC-V assembly language program that reads data from the switches and considers those switches to be an unsigned binary number. The program then converts that binary number to stone age unary and outputs that value to the LEDs. The value input from the switches is never greater than 16. The port address of the switches is port address 0x1100C000; the port address of the LEDs is 0x11008000.
-

11 Working with Memory

11.1 Introduction

Though you have not seen it yet, most everything you with assembly language involves interfacing with memory in some way. We've introduced the load and store instructions in a previous chapter, but we did not show the full power of memory-type instructions with meaningful examples. This topic is so important that we've opted to dedicate an entire chapter to it. This chapter shows the full functionality and potential of interfacing with memory on the RISC-V MCU.

Main Chapter Topics

- **ASSEMBLY ADDRESSING MODES:** This chapter describes the various ways that assembly language instructions locate the data they require from various memory elements in the computer hardware.
- **MAIN MEMORY AND REGISTER FILE DATA ACCESS AND USAGE:** This chapter provides an in-depth and intuitive description of how instructions access and utilize main and register file memory.

Why This Chapter is Important

This chapter is important because it shows the full flexibility and functionality of RISC-V memory-type instructions.

11.2 Overview

Our rough model of a computer was a device that uses the instructions in the underlying program to tell the computer hardware what to do with data. Additionally, the instructions also tell the hardware where to obtain that data they require. In solving a problem using a computer, the computer waits for the outside world to make a request and then acts on that request when it arrives; eventually the computer outputs a result.

We write assembly language programs using a text editor to create a file that is a collection of the instructions in our program. We then use a special program (an assembler) which translates the instructions (text) in our program into machine code; we then use another mechanism to place that machine code into the program memory of the computer hardware.

One of the stated benefits of using an MCU to solve our problems (as opposed to FSM controlled hardware) is that fact that programs are much more flexible. This form of flexibility means that I can use the MCU to solve many different problems by simply changing the program. Using digital circuits to solve problems that do not use a MCU are not flexible in that you must make major hardware redesigns for each problem you solve. The point here is that MCUs are flexible; the hardware in a computer is effectively non-changeable, which underscores the major point of this diatribe: the flexibility in using MCUs to solve problems lies in the flexibility associated with changing programs. Thus, this flexibility lies in the instructions themselves.

11.3 Flexibility in Instructions

Us human programmers essential use computer instructions to direct the computer on how we want data to move through the computer hardware. When we speak of data, we inherently speak of two issues: where to get the data from and what to do what that data. Some instructions are responsible for “crunching” data and other instructions

are responsible for moving data around. Both of these instruction types have the issue of where to obtain the data from and where to place it; not all instructions actually operate on that data (such as an arithmetic operations). The flexibility we're interested in here is where the instructions obtain the data from and where to put it once the instruction completes.

11.3.1 Register Addressing vs. Memory Addressing

We have two different places to obtain data from in the RISC-V MCU: the register file and the main memory. Each of these locations are "storage" locations in that both modules are types of memory. As with all memory, we can read from and write to these modules. When we read from them, we provide them with the address of the data we want to read; when we write to them, we provide the address of where we want to place the data and the actual data we want to put in the memory. The approach instructions use to address register file memory vs. main memory is distinctively different.

11.3.1.1 Register Addressing

The RISC-V MCU contains 32 general-purpose registers, which is a relatively small number in the context of computer memory. Because this number is relatively small, the instructions in the RISC-V ISA require that we state directly which register we want to use. For example, the instruction "`add x10, x11, x12`" calls out that we use three registers, two are source registers (x11 & x12) and one is the destination register (x10). This type of instruction grabs from data from somewhere, crunches it, and then places the data somewhere else. This instruction goes to the register file (memory) to obtain the two operands, crunches the data, and stores the result back into the register file. This instruction is reading data from memory and writing some data back to memory, which means there is some underlying addressing going on, which is not overly apparent from the instruction text itself.

The reality is that when we specify "x10" in an instruction, the assembler forms the actual address that the computer hardware uses to access the memory we specify. In other words, the assembler translates to numeric value following the "x" in the register specification to an address that the hardware can use. Specifically, because there are 32 register, the assembler translates the number following the "x" specification to a five-bit field and encodes it as part of the instruction. The ramifications here is that the assembler assigns the specific memory address at "assemble time" and makes it part of the machine code associated with that instruction. This means that the assembler fixes the register address (memory address) at assemble time and the program can never change that address without re-assembling the program. The fact that the assembler fixes the address at assemble time effectively makes the instruction "lacking of flexibility" because the particular instruction is always using data at the same address. Note here that we can change the data at that address; we simply can't change the address.

11.3.1.2 Main Memory Addressing

The RISC-V accesses memory using load-type and store-type instructions. Recall that load-type instructions read data from memory and writes that data to a register address while store-type instructions read data from a register and writes that data to main memory. As with all memory, we need to provide a memory location to write to or read from. The RISC-V MCU load-type and store-type instructions have a special way to "form" the memory addresses, which we show again in Table 11.1. Note that there is nothing special about this approach; it's simply the approach RISC-V chooses to use.

Table 11.1 shows that the memory address calculation has two parts: the offset value and the base address value; the underlying RISC-V hardware adds the offset value to the base address value in the specified register to form the absolute memory address. The hardware does the final address calculation at "runtime"; at assemble time, the assembler does not know what the absolute address will be. The only thing known at assemble time is which register the hardware uses to form the absolute address. Recall that when dealing with register values directly such as in number crunching instructions, the assembler know the absolute address of the register (memory) assemble time. With load and store-type instructions, the underlying hardware calculates the absolute address at runtime. While it is true that we're always using the same register value for a particular load or store-type instruction, the flexibility in this approach is that we use the program to change the value in that register.

The fact that the load and store-type instructions calculate the absolute memory address at runtime provides flexibility to the programs we write. Because the hardware calculates the absolute memory addresses at runtime

effectively means that we can access different parts of memory according to the needs of our program. As we show in the remainder of this chapter, this approach allows us to write incredibly flexible programs, which in turn allows us to solve more problems in an efficient manner.

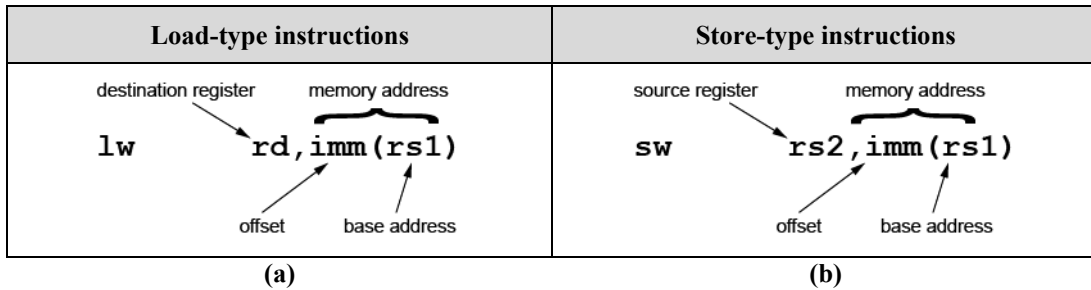


Table 11.1: Overview of the load and store-type instructions.

11.3.2 Assembly Language and Addressing Mode

The term addressing modes refers to the ways we can use the instruction set to access data. The notion of addressing modes are special items in the context of assembly languages because the more different number of addressing modes you have, the more flexibility you have in the programs you write. While the notion of having many addressing modes in the ISA quite appealing, the notion of having many addressing modes creates several issues.

- 1) **Having many addressing modes complicates the underlying hardware.** As with the issues of the RISC-V load and store-type instructions, the hardware does the absolute address calculation at runtime. This means there needs to be hardware in the computer to do the actual calculation. The effect is that more addressing modes you have, the more likely it is that your hardware is larger and/or more complex. There are cases where you can reuse existing computer hardware to perform the required calculation, but this is not always the case, or you'll need to extend the execution time of the instruction to use that hardware. Actually, the previous statement is somewhat misleading.
- 2) **Having many addressing modes is not always helpful for humans.** The addressing modes in some computer architectures are surprisingly complex. While you can figure out how to use them if you stare at them long enough, humans tend to stick with simple addressing modes. So why have so many addressing modes? Having many addressing modes makes compiler writers really happy. While us humans find it hard to wrap our brains around some of these modes, the compiler is a program that can utilize the various modes quite readily¹.

11.4 Memory Access: Solved Problems

The best way to see the flexibility of memory access instructions is to see them in actual assembly language code. The section shows a few of these problems with extra justification of why they are so flexible.

Example 11.1: Register Data Swap

Write a fragment of code that swaps the values in registers x7 & x9.

Solution: Figure 11.1 shows the solution to this example. Here are the pertinent points to note about the solution:

- There are several ways to swap data in two registers; this is one approach. This particular approach uses an extra register to do the register swap; the code opts to temporarily store one of the values in another register, which we commonly refer to as a “working register”. For this code

¹ Assuming of course that the people writing the compiler know what they are doing.

fragment, x20 is the working register and is going to be overwritten by the program, which may be an issue.

- The approach the code takes is to 1) save data in one register, 2) copy data from one register to the other, and 3) copy the originally saved register data to the other register.

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)         mv    x20,x9          # copy data in x9 to working register (x20)
(03)         mv    x9,x7          # copy data from x7 to x9
(04)         mv    x7,x20        # copy working register data to x7
(05)
(06) #~~~~~ program fragment ~~~~~

```

Figure 11.1: The solution to this example problem.

You're probably thinking that this is not too exciting; you're absolutely correct. But here's the point. This code fragment always uses data from the same place to do the swap: it's always swapping data in the x7 & x9 registers; it never does anything different. We can change the data in those registers before we swap them, but we always have to use the same registers. This works, but it lacks flexibility in the way it address the data it needs to swap.

Example 11.2: Memory Data Swap

Write a code fragment that swaps the words in two different memory locations. Registers x6 & x7 provide the locations of the data to swap; specifically, these two registers hold the addresses of the data to swap.

Solution: This is yet another version of swapping something, but this time the problem is switches between two memory locations as opposed to two registers as we did in the previous example. Figure 12.11 shows the solution to this example; here are some other fun facts to fill your mind:

- The code effectively embodies the previous code in that the code on lines (05-07) are structurally identical to the previous example. But that's not the point...
- The approach this code takes is to 1) load the data from memory into general-purpose registers, 2) swap the data, and 3) put the data back into the original storage locations, but swapped.
- The problem stated that we were working with words, so we use **lw** instructions (load word) to read the data into registers and **sw** instructions (store word) instructions to return the data from registers to memory.

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)     lw    x10,0(x6)      # load the data to swap
(03)     lw    x11,0(x7)
(04)
(05)     mv    x12,x10      # copy data in x10 to working register x12
(06)     mv    x10,x11      # copy data from x11 to x10
(07)     mv    x11,x12      # copy working data to x11
(08)
(09)     sw    x10,0(x6)      # store the swapped data
(10)     sw    x11,0(x7)
(11)
(12) #~~~~~ program fragment ~~~~~

```

Figure 11.2: The solution to this example problem.

Once again, you're probably thinking this is another boring problem. The point of this problem is that we are not limited to swapping data from the same two register; now we can swap data from any address in data memory simply by changing the data in the base addresses (x6 & x7). In other words, the addresses of the data we need to swap are no longer fixed as they were when we used registers to swap the data. It is true that we need to use the same two registers as base addresses, but we can change the values in those register under program control.

Example 11.3: Memory Data Swap Yet Again

Write a code fragment that swaps the halfwords in two different memory locations. Register x15 provides the memory address of the first data to swap; the other piece of data to swap directly follows the first piece of data in memory.

Solution: This is yet another version of swapping something, but the solution is much more efficient this time. Here are the details:

- The problem statement only provided one address, stating the data to swap was effectively continuous in memory. The data is halfwords, so we use lh and sh instructions to account for the halfword data size.
- The approach this code takes is to load the data from memory into general-purpose registers then write the data back out to memory by effectively swapping the address. Note that the code on line (03) and line (05) use the value of two as the offset value to accomplish the swap.

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)     lh    x10,0(x15)     # load the first halfword
(03)     lh    x11,2(x15)    # load the second halfword
(04)
(05)     sh    x10,2(x15)    # store the swapped data
(06)     sh    x11,0(x15)    #
(07)
(08) #~~~~~ program fragment ~~~~~

```

Figure 11.3: The solution to this example problem.

Not a boring problem once again. We accomplished the swap without changing any register values as we did in the previous solution, which is generally a good thing. We can't swap data in any two addresses in memory, but we can swap two contiguous pieces of data at any memory location. The point here is that the code is still quite flexible, but not as flexible as the previous problem.

Example 11.4: Modifying a Section of Memory

Write a code fragment that adds three to each contiguous byte in a section of memory starting at the address in x25. The length of that section of memory is 32. Don't worry about overflow for this problem.

Solution: This is a problem where we need to go to a section of memory and modify each byte in that section of memory. Note that the problem calls out that the section of data is bytes. Here are the details regarding the solution in Figure 11.4:

- The problem statement only provided one address, which is the location of the first byte of data in the section of memory. The problem also states that the data at that address and the data at the next 31 addresses needs to have three added to it.
- The first thing we do is set the iteration count to 32, which we do on line (02). The problem stated 32, so we put that number into a register to use as our loop counter.
- The code in the body of the loop (starting with the “loop” label) includes loaded the data (line (04)), adding three to the data (line (05)), and storing the data at the same memory address location we loaded the data from (line (06)). Note that the problem gave the address of the first byte of data, which is in x25; we use that register as an address throughout this piece of code.
- The loop administration starts at the instruction associated with the “admin” label on line (08). We first increment the register holding the memory address on line (08); we add one because the problem states that we are working with byte data. We then decrement the loop count on line (09). We finally check the loop count with a conditional branch on line (10). If the loop count is zero (the value in x0), then we fall through to the next instruction (not listed); otherwise we branch to the instruction associated with the loop label.
- We modeled this fragment as a do-while loop because we knew we had to do at least one iteration. If we did not know how many iterations we needed to do, we would have modeled the loop as a while loop (because the loop count may be zero).

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)     li    x10,32           # load iteration count
(03)
(04) loop:  lb    x11,0(x25)    # load a byte from memory
(05)     addi x11,x11,3        # add 3 to data
(06)     sb    x11,0(x25)    # store the swapped data
(07)
(08) admin: addi x25,x25,1      # advance memory address
(09)     addi x10,x10,-1       # decrement loop count
(10)     bne  x10,x0,loop      # branch for next iteration
(11)
(12) #~~~~~ program fragment ~~~~~

```

Figure 11.4: The solution to this example.

Figure 11.5 shows a flowchart that models the solution to this example. In an effort to save vertical space in this text, we used two columns in the flowchart. Using two columns required us to cross flow lines on the lower right portion of the flowchart, which is common in more complex flowcharts. The flow lines do not intersect and effectively remain independent of each other. We also added a note to indicate the loop administration part of the loop.

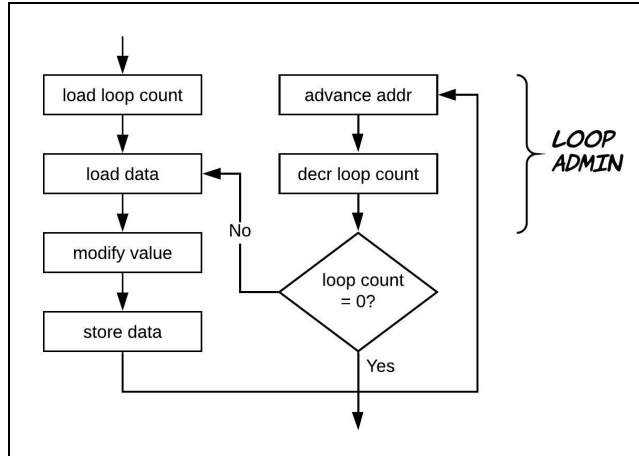


Figure 11.5: A flowchart modeling the operation of this example program.

The point of the problem is that we went to the data in a section of memory and modified each piece of data in that section of memory. Because we needed to modify 32 chunks of data, we could not have possibly stored that data in registers. The approach in this example leveraged the generic nature of memory access using the built-in flexibility of the RISC-V memory access instructions. The solution in Figure 11.4 is very space efficient being that we used an iterative construct; the code is arguably relatively runtime efficient.

Just for the heck of it, Figure 11.6 shows the same solution using a while loop. This solution does not have a natural feel to it as did the solution with the do-while loop; this solution feels a bit klunky.

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)         li    x10,32          # load iteration count
(03)
(04) loop:    addi  x10,x10,-1     # decrement loop count
(05)         beq   x10,x0,done     # branch for next iteration
(06)
(07)         lb    x11,0(x25)     # load a byte from memory
(08)         addi  x11,x11,3      # add 3 to data
(09)         sb    x11,0(x25)     # store the swapped data
(10)
(11) admin:   addi  x25,x25,1     # advance memory address
(12)         j     loop
(13)
(14) done:    # some other part of the program
(15) #~~~~~ program fragment ~~~~~
  
```

Figure 11.6: An alternative solution to this example.

Example 11.5: Gathering Statistics About a Section of Memory

Write a code fragment that counts the number zero values in a contiguous section of memory starting at address x10. The number of values to scan is given in register x30 and is guaranteed to be non-zero. Store the count in register x15. Consider the memory values to be halfwords.

Solution: This is yet another version of going to memory and doing something. In this problem, we are going to memory and gather some information about the data in that particular section of memory. Figure 11.7 shows the solution to this example: here are the gory details:

- The iteration count for this problem is in a register, which means that this count may be zero (the problem did not state that it would be non-zero). This means two things for us: first, we must initialize the final count before we start the iteration, which we do on line (02). Second, we must use a while loop to implement the iteration construct, which we do to account for the fact that the count may be zero. If you performed an iteration first (as in a do-while loop), the program would fail miserably when the iteration count is zero. We check the loop counter on line (04).
- The body of the loop loads data on line (06) and then checks to see if it zero on line (07); we toss in the “check” label for added commentation. If the value is non-zero, we branch to the code that implements the loop administration on line (10); otherwise, we drop to line (11) and increment the counter keeping track of the number of zero’s in the section of memory.
- The loop admin consist of advancing the address by two (line (10)) and decrementing the iteration count (line (11)). An unconditional branch follows the loop administration on line (12), which is part of the while-loop iterative structure.

```

(00) #~~~~~ program fragment ~~~~~
(01)
(02)      mv    x15,x0      # clear zero counter
(03)
(04) loop:   beq   x30,x0,done # branch for next iteration
(05)
(06)      lw    x11,0(x10)  # load a halfword from memory
(07) check: bnez  x11,admin # jump if value not equal to zero
(08)      addi  x15,x15,1   # increment the count
(09)
(10) admin:  addi  x10,x10,2 # advance memory address (halfword length)
(11)      addi  x30,x30,-1  # decrement count
(12)      j     loop
(13)
(14) #~~~~~ program fragment ~~~~~

```

Figure 11.7: The solution to this example.

Yet once again, we could have done this problem using data in registers. First, the fact that register usage is not generic would have stopped us. Second, we don’t know how much data we need to inspect. Once again, we can access main memory in a generic manner, something we can’t do with register memory. Figure 11.8 shows a flowchart modeling the solution for this example.

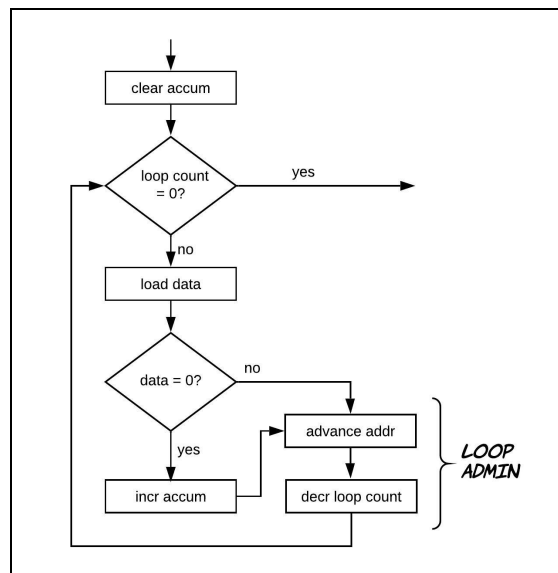


Figure 11.8: A flowchart modeling the operation of this example program.

Example 11.6: Copying a Section of Memory.

Write a code fragment that copies one section of memory (words) to another. The section to copy starts at the address in x10; the section of memory to copy to starts at the memory address in x20. The number of values in x5 is always non-zero.

Solution: This is yet another version of swapping something, but the solution is much more efficient this time. Here are the details:

- The problem does not state how many times we need to iterate, but it is at least once (as the problem states). Because of this, we implement our solution using a do-while loop.
- The approach this code takes is to load the data from one memory location then write it to the other location (lines (02-03)). The loop administration includes advance each of the memory addresses (copy from and copy to) by four since the problem is dealing with words (lines (05-06)). Lastly, we decrement the loop count on line (08) and to it all again if the loop count is non-zero (line 09).

```
(00) #~~~~~ program fragment ~~~~~
(01)
(02) loop:    lw    x11,0(x10)    # load a word from memory
(03)        sw    x11,0(x20)    # copy value to new location
(04)
(05) admin:  addi  x10,x10,4     # advance "copy from" memory address (word)
(06)        addi  x20,x20,4     # advance "copy to" memory address (word)
(07)
(08)        addi  x5,x5,-1      # decrement iteration count
(09)        bne  x5,x0,loop     # branch if iteration count not zero
(10)
(11) #~~~~~ program fragment ~~~~~
```

Figure 11.9: The solution to this example.

Yet another problem that we solved generically, thus advertising the flexibility of RISC-V memory access instructions. We could not have done this problem using registers to hold the data we needed to mess with. It's true we did use registers in this problem, but we used the registers to primarily hold addresses, which gave us the ability to access different data in memory by simply changing the address values in the registers. Figure 11.10 shows a flowchart that models this solution. Have lots of fun.

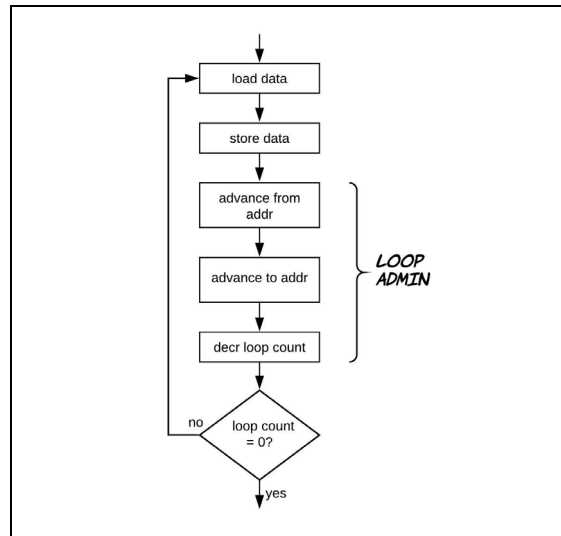


Figure 11.10: A flowchart modeling the operation of this example program.

11.5 Chapter Summary

- Addressing modes in assembly languages is an important topic with classical trade-offs. Simple and few addressing modes result in smaller hardware; many and complex addressing modes make the hardware larger.
 - The two primary memory modules in the RISC-V are the register file and the main memory. Both of these modules are memory, but the assembler effectively treats them differently. Both of these memory modules require an address to access data in the memory, but the instructions handle the addressing of the data in significantly different ways.
 - Instructions that use data held in register include the 5-bit address of the registers as part of the underlying instruction word, which means the address of the data being accessed by such an instruction is known at assemble time. Instructions that access main memory use registers in the absolute address calculation, but the absolute address is calculated at run time. The fact that the absolute address in main memory accesses are calculated at run time make the instructions very flexible, and thus useful.
-

11.6 Chapter Exercises

- 1) List the main differences between register file memory and main memory in the RISC-V.
 - 2) List the ways main memory and the register file memory are strangely similar and quite different from a simple memory module.
 - 3) Briefly describe what is meant by the terms “absolute address” and “relative address”.
 - 4) Briefly describe the main reason why the RISC-V uses relative addressing as part of the instructions but later converts the relative addresses to absolute addresses.
 - 5) Briefly describe whether relative addresses are signed or unsigned values, and why they are that way.
 - 6) Being that instructions only encode relative addresses, briefly describe whether that places any limits on the values that can be created and used as absolute addresses.
 - 7) If a relative address encoded with a given number of bits only needed to go in one direction, could it go farther in that direction than if the same number of bits was used to go in both directions? Briefly explain (sorry for the cr*ppy wording of problem).
 - 8) Briefly describe how the absolute addresses are generated for accessing memory in the register file.
 - 9) Briefly describe how the absolute addresses are generated for instructions that access main memory.
 - 10) Briefly describe the differences between generating absolute addresses at run time and and at assemble time.
 - 11) Briefly describe why absolute addressing for register memory access is considered not flexible.
 - 12) Briefly describe why absolute addressing for main memory access is considered flexible.
 - 13) Does the assembler know the absolute address used by load and store-type instructions? Briefly explain.
 - 14) Explain the notion of “addressing modes” in the context of assembly language programming.
 - 15) Briefly explain why it is advantageous to have many addressing modes for a given computer architecture.
 - 16) Briefly explain the main drawback of having many addressing modes for a given computer architecture.
 - 17) Briefly describe why main memory data access is considered more flexible and accessing the data in the register file.
-

11.7 Chapter Programming Problems

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code
- 1) Write a code fragment that divides the word data by 32 in a section of memory starting at the address in x15. The length of that section of memory is in 50 words.
 - 2) Repeat the previous problem but round up the result after the division.
 - 3) Write a code fragment that examines 50 contiguous words in memory. If the data at a location is odd, then write zero to that particular memory location.
 - 4) Repeat the previous problem but store the number of times the code was zero a value in register x25.
 - 5) Write a code fragment that divides word data in a section of memory by two until the data is less than 256. The each word is stored back at the same memory address. The starting address of the memory is in x22. The length of that section of memory to operate on is given in x10.
 - 6) Write a code fragment that copies swaps halfword data from one section in memory (signed halfwords) to another section in memory. The section of memory to copy from starts at the address in x15; the section of memory to copy to starts at the memory address in x25. The number of values is in x10 and may be zero.
 - 7) Write a code fragment that examines two sections of unsigned halfwords in memory with the starting addresses stored in x10 and x20. Store the larger of two values at the address in x20 for the number of halfwords stored in register x30. Note that the data at some memory locations need to be swapped, but not at all the locations.
 - 8) Repeat the previous problem, but count the number of swaps and stores that value in x31.
 - 9) Write a code fragment that examines three sections of contiguous signed bytes in memory with the starting addresses stored in x10, x11, and x12. If all three of the values at the data in each location are equal, write 0xFF to each location; otherwise don't change the data. The number of sets of three signed bytes to check is the number in x30.
 - 10) Write a code fragment that examines three sections of contiguous words in memory with the starting addresses stored in x20, x21, and x21. This fragment of code counts the number of time zero appears in each of the three sections of memory and stores the result in x15. The number of values to check in each segment is in x30.
-

12 Subroutines and Supporting Structures

12.1 Introduction

Subroutines and their usage in assembly language programming represent a major issue both programming and writing well-structured and efficiently operating programs. The proper use of subroutines allows for the creation of understandable, maintainable, reusable, and relatively runtime efficient assembly language programs. The only catch is that writing great programs requires programmers to understand the relatively few, but significantly important aspects regarding subroutine. This chapter describes subroutines from primarily a programming aspect, but it does reach into some significant hardware aspects as well.

The notion of subroutines falls into the category of program flow control because calling and returning from subroutines necessarily changes the normal sequential execution of instructions in an assembly language programming. You'll soon see that there are no new instructions involved with calling subroutines and returning from them, the `jal` and `jalr` instructions have all the required functionality.

Main Chapter Topics

- **THE STACK:** This chapter describes the abstract data type known as a stack, how the assembly language program uses the stack, and some important functional issues regarding proper stack usage.
- **SUBROUTINES:** This chapter describes the many issues involved in implementing and calling subroutines in the context of the RISC-V MCU.
- **PASSING VALUES:** This chapter describes the concepts of passing values to and from subroutines.
- **SAVING CONTEXT:** This chapter discusses the procedures for saving and restoring operating context in subroutines.
- **NESTED SUBROUTINES:** This chapter describes the special issues the RISC-V MCU has with nested subroutines.
- **SUBROUTINE OVERHEAD:** This chapter describes the various overhead issues regarding subroutine implementations.

Why This Chapter is Important

This chapter is important because it describes the details involved in the design and implementation of subroutines in assembly languages.

12.2 Subroutine Supporting Structures: The Stack

The word stack has many different meanings for the people who use the word. Some of the definitions include haystack, smoke stack, pancake stack etc., but we won't use these definitions here in technical-land. Here in technical-land, there are two main definitions of a stack. In software-land, the stack is one of the classic *abstract data types*, or, *ADTs*. In this context, the definition of an abstract data type is a data type that we describe in terms of the operations the data type supports rather than how we actually implement the data type. In other words, we define an ADT by its interface while placing no constraints on the implementation details.

In the context of computer architecture, the stack has less of a computer science-type definition because we can describe the implementation of the stack in terms of simple hardware. Furthermore, the stack in standard computer architecture is an important part of any architecture because the hardware necessarily uses the stack for important program flow control mechanisms such as implementing subroutines and interrupts¹.

The basic concept behind a stack is simple: it is nothing more than an object that stores data. The most basic definition of the stack lies in the description of the accessibility of the things that have been stored on the stack. The short definition of a stack is that the most recent thing that you place on the stack is the first thing that you can remove from the stack. We refer to this functionality as *Last In, First Out*, or *LIFO*.

Before going further, let's define a couple of terms for so we'll be speaking the same language regarding stacks. These terms are standard for any stack implementation; anyone dealing with stacks roughly knows what these terms mean in the context of the particular stack implementation they are working with. You need to know all of these terms for a hardware context, but you only need to know the first two terms if you're strictly a programmer. Note that none of these terms provides any actual implementation details.

- PUSH – This is the accepted term to mean that you are placing something onto the stack.
- POP – This is the accepted term meaning that you are removing something from the stack.
- Top of the Stack – We define the “top of the stack” to be the most recent object that we place, or *push* onto the stack. If the stack is empty (nothing has been pushed onto the stack), the top of the stack then has somewhat ambiguous meanings.
- Stack Pointer – We use the “stack pointer” as the “thing” we use to point to the top of the stack, where the top of stack is the most recent thing placed on the stack.

There are two ways to demonstrate the operation of the stack. The first way is more of the computer science approach² while the second way is a hardware approach. The hardware approach is more of what we're interested in as it is how the RISC-V MCU implements the stack in hardware, but the first approach makes for a nice introduction to the second approach. Figure 12.1 shows the first approach. Here is a description of the changes that take place in Figure 12.1; note that in Figure 12.1 we use the word “top” to indicate the top of the stack.

- Image 1: the stack in its empty state. For the empty stack, the top label is not well defined.
- Image 2: the stack after one item has been pushed onto the stack. The top label is associated with the most recent item placed on the stack.
- Image 3: the stack after four items (three since image 2) have been pushed onto the stack. The number 34 was the first number pushed onto the stack, followed in order by 29, 19, and then 17.
- Image 4: the stack after one item has been popped from the stack (the number 17 was removed).
- Image 5: the stack after three items (two since image 4) have been removed from the stack.

¹ As you will see later, the RISC-V OTTER interrupt architecture does not directly use the stack.

² Probably a better “computer science” approach would be to show a “linked list-type implementation.”

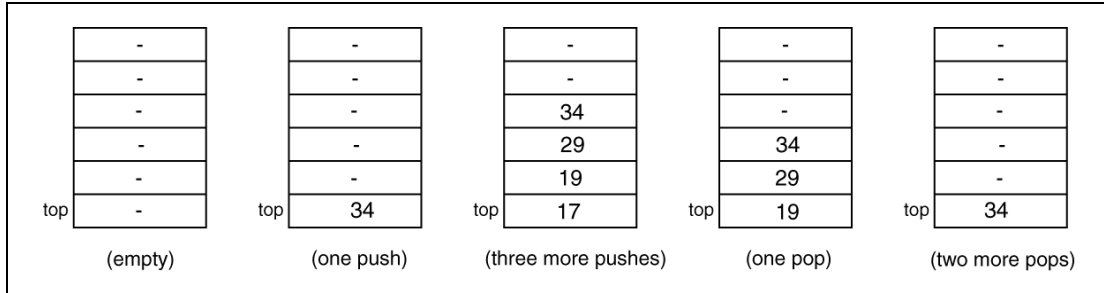


Figure 12.1: Example of a software-based stack implementation.

The key feature of the stack implementation shown in Figure 12.1 is that all the stack elements move each time a push or a pop operation executes. Note that this implementation would be inefficient for a hardware version of the stack because each stored stack item would need to be re-written for every push and pop operation, because each push and pop operation changes the location of all remaining data in the stack. This being the case, we can better define the concept of a stack pointer by examining a hardware-type stack implementation.

The stack on the RISC-V MCU is nothing more than a designated area in main memory. The size of the stack, as well as starting and ending locations are arbitrary. We’ve included the current RISC-V memory map once again in Figure 12.2 to provide a visual representation of the “preferred” stack location. As you will see later, you can “place” the stack anywhere in main memory except for the space dedicated to the program memory (represented by the section marked “Code Segment” in Figure 12.2).

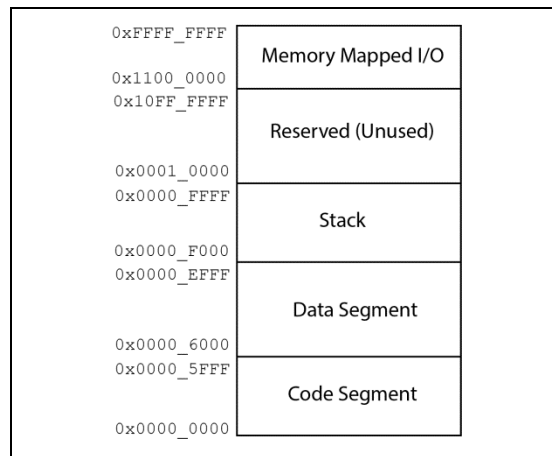


Figure 12.2: The RISC-V MCU memory map.

12.2.1 Pushing and Popping on the RISC-V MCU

You could say that a set of instructions common to most MCUs is some form of a push and pop instruction. But somewhat surprisingly, the RISC-V MCU has no such instructions. While push and pop instructions are handy to have in the instruction set, we don’t need them because push and pop operations are simply memory writes and reads from a “special place” in memory. We refer to that special place in memory as the stack.

The first order of business is the stack pointer. The stack pointer needs to be a register that holds an address; the MCU uses that address when doing operations on the stack. While many MCUs have a dedicated register, typically referred to as the stack pointer, the RISC-V MCU does not have such a register. The RISC-V MCU architecture is generic enough to use any register in the register file as a stack pointer, but it standard practice to use register x2 as the stack pointer. It’s handy that the alternative name for register x2 is “**sp**”, which stands for stack pointer. Despite the significant amount of flexibility in the RISC-V MCU’s approach to stack operations, we’ll only deal with a basic approach in this text.

Table 12.1 shows a summary of push and pop operations on the RISC-V MCU. Here is the important information to note about Table 12.1:

- Pushing and popping of one register requires issuing two instructions. We have to both adjust the stack pointer, and then execute a memory access operation.
- Push operations require a memory write operation (store) while pop operations require a memory read operation (load).
- The “Usage” columns in Table 12.1 lists the two forms of pushes and pops. Either form is acceptable, but the left-most usage column shows the most widely used form.
- There is often a need to push or pop more than one register in a section of source code. In these cases, it’s tempting to issue two instructions per push or pop operation, similar to the first two rows in Table 12.1. The better approach is to adjust the stack pointer only once per set of push or pop operation, then use the offset field of the memory access instructions (**lw** & **sw**) to form the correct memory access address. This approach is both more time efficient and space efficient.
- In this example, are pushing and popping entire registers, which is why we use **lw** & **sw** instructions. If you only need to save parts of the registers, you could use memory access instruction dealing with bytes and halfwords.

Operation	Usage Example	Alternate Usage	Comment
push (1 reg)	addi <i>sp, sp, -4</i> sw <i>x8, 0(sp)</i>	sw <i>x8, -4(sp)</i> addi <i>sp, sp, -4</i>	Push x8 onto stack (store/write value in x8 into memory)
pop (1 reg)	lw <i>x8, 0(sp)</i> addi <i>sp, sp, 4</i>	addi <i>sp, sp, 4</i> lw <i>x8, -4(sp)</i>	Pop x8 off stack (load/read value into x8)
push (2 regs)	addi <i>sp, sp, -8</i> sw <i>x8, 0(sp)</i> sw <i>x9, 4(sp)</i>	sw <i>x8, -4(sp)</i> sw <i>x9, -8(sp)</i> addi <i>sp, sp, -8</i>	Push x8 then x9 onto stack (store/write x8, x9 into memory)
pop (2 regs)	lw <i>x8, 0(sp)</i> lw <i>x9, 4(sp)</i> addi <i>sp, sp, 8</i>	addi <i>sp, sp, 8</i> lw <i>x8, -4(sp)</i> lw <i>x9, -8(sp)</i>	Pop x8 then x9 off stack (load/read into x8, x9)

Table 12.1: Summary of RISC-V push & pop operations for one & two registers.

One of the big mistakes that programmers can sometimes make when pushing and popping is to pop registers off the stack in a different order than was put on the stack. This is because the stack is a LIFO abstract data type, so ordering does matter. Table 12.2 shows both a correct and incorrect popping sequence for a given example of two pushes. The incorrect sequence does not cause the assembler to give you an error, which is why you have to be extremely careful when popping values off the stack. The incorrect pop sequence column does actually serve a purpose: it swaps the values in registers x8 and x9, which can sometimes be useful.

Push Operation	Correct Pop Sequence	Incorrect Pop Sequence
addi <i>sp, sp, -8</i> sw <i>x8, 0(sp)</i> sw <i>x9, 4(sp)</i>	lw <i>x8, 0(sp)</i> lw <i>x9, 4(sp)</i> addi <i>sp, sp, 8</i>	lw <i>x9, 0(sp)</i> lw <i>x8, 4(sp)</i> addi <i>sp, sp, 8</i>

Table 12.2: Correct and incorrect pops for two a two register push.

Example 12.1: Push and Pop Code Fragments

Write two fragments of RISC-V assembly language code; one pushes register x20, x21, & x30 on the stack, and the corresponding code that pops those values off the stack.

Solution: Figure 12.3 shows the solution to this example. There are several particularly important things to surprise your brain with in this solution:

- We are pushing three registers, so we need to reserve room on the stack for three registers worth of data, or, 12 bytes. We do this by adjusting the stack pointer backwards (lower address) by 12 on line (02). We include a “save” label for human clarity.
- We next use the **sw** instruction to store the three registers designated by the problem on lines (3-5). The ordering of these instructions does not matter, but the offset values for each **sw** instruction do matter. We push x20 on line (03) with a zero offset because the stack pointer is currently pointing 12 byte locations back from its original value. We need to push three registers, the order in which we push them is arbitrary.
- The solution includes two options for restoring the registers with pops. The pop operations utilize the **lw** instruction to restore the pushed 32-bit register values back into their original registers. The two versions, labeled “restor1” and “restor2” both pop the registers before adjusting the stack pointer last. This is not the only approach but is the best approach. We include two versions to show that we can, and that the ordering on instructions does not matter. What does matter is that we pop the correct data back into the correct register, which we do in both solutions.

```

(00) #~~~~~ program fragment ~~~~~
(01) #
(02) save:  addi  sp,sp,-12    # adjust stack pointer (sp) to hold 3 registers
(03)      sw   x20,0(sp)     # push x20
(04)      sw   x21,4(sp)     # push x21
(05)      sw   x30,8(sp)     # push x30
(06)
(07) #~~~~~ program fragment ~~~~~
(08) #
(09) restor1: lw   x20,0(sp)   # pop x20
(10)        lw   x21,4(sp)   # pop x21
(11)        lw   x30,8(sp)   # pop x30
(12)        addi sp,sp,12    # re-adjust sp
(13) #
(14) #~~~~~ alternate program fragment ~~~~~
(15) #
(16) Restor2: lw   x21,4(sp)   # pop x21
(17)        lw   x30,8(sp)   # pop x30
(18)        lw   x20,0(sp)   # pop x20
(19)        addi sp,sp,12    # re-adjust sp
(20) #
(21) #~~~~~ program fragment ~~~~~

```

Figure 12.3: Solution for this example.

12.3 Subroutines Overview

It is frequently necessary for a program to execute the same set of instructions at several different points in a program. If the set of instructions is relatively short, you could simply place the code in the program wherever your program requires it. However, if this section of code is relatively long, a more effective use of codespace to have only one piece of the code that needs repeating. When that section of code needs to execute, the program control transfers to the section of code that requires execution, the program executes that code, and then the program control transfers back to the code that it originally transferred from. Thus, when the special section of code completes execution, program control returns to where it was before the program executed that special piece of code. The thing we are describing here is what we know in assembly language terms as a *subroutine*.

We refer to this same mechanism in a higher-level language a *function* or a *method*, depending on the higher-level language you're working with.

Subroutines have another major purpose besides saving associated with them that is extremely important to assembly language programs: they give the programmer the ability to modularize and thus organize their programs. In essence, there are situations where you *must* use subroutines as they save you program code space, but there are also situations where you *should* use subroutines to keep your programs organized. The use of subroutines is important for many reasons; the main ones we list below.

Codespace Efficiency: The section of code that needs executing multiple times appears only one time in the source listing. This represents a significant savings in program codespace because despite the code appearing only once, the programmer can easily execute it multiple times. The efficiency we refer to here is program memory space efficiency, which is different from runtime efficiency.

Program Readability and Understandability: Placing bunches of code in subroutines and giving it an appropriate name makes the program more readable. This in turn allows the programmer to abstract the code to higher levels, which means you don't necessarily need to understand the workings of code at low level (the code in the subroutine) in order to use the code. Additionally, providing the code with a self-commenting label (or subroutine name) allows another human to quickly understand the purpose of the code on a high level.

Maintainability: Compartmentalizing the code allows you to quickly locate and easily change only the code you need, particularly if it later turns out that there is a problem with that code.

Reusability: Placing sections of code in meaningful blocks, namely subroutines, increases the chances that you or some other programmer can reuse the code later, which saves time by preventing multiple programmers from "re-inventing the wheel". This is also why you should always provide adequate commenting on your subroutines.

In the end, the notion of using subroutines in your programs can't be overemphasized. Good programmers use appropriately named and structured subroutines in order to control the complexity of their code. Conversely, you can always detect code from beginning assembly language programmers because they tend to avoid using subroutines and/or try to use jumps and branches instead. The concept of subroutines from a programmer's level is straightforward; please don't fear the subroutine.

Figure 12.4 shows an example of software flow diagrams that justifies the use of subroutines. In Figure 12.4 (a), program flow continues in a linear manner and executes the sections of code represented by A, B, C and X. The code represented by X appears two times in the section of code. In Figure 12.4(b), program flow jumps from A to X and then from X to B. Likewise, it jumps from B to X and then from X to C. In this way, the code that represents X needs to appear in the code only one time. This represents a direct saving in code space in program memory, though there are other negative ramifications in terms of program run-time execution efficiencies (this is the overhead issue; we'll talk about this later). The notion that Figure 12.4 is attempting to convey is that using subroutines saves program code space.

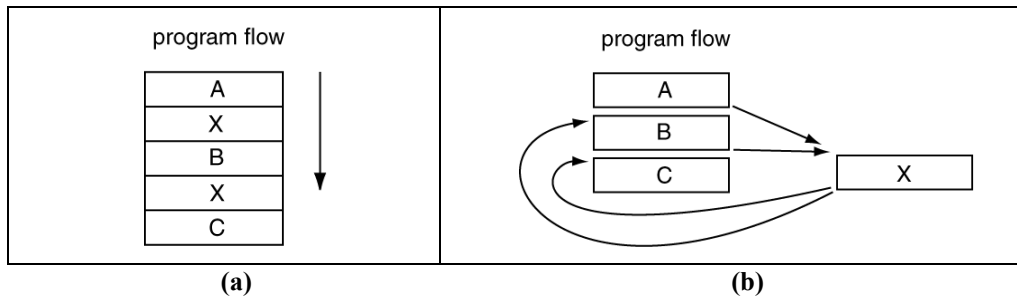


Figure 12.4: A diagram showing program flow both with and without subroutines.

Subroutines are a form of program control not unlike the unconditional branch instruction. When the program executes subroutines, the program temporarily transfers program control to some other place in the program (or

more precisely, some other address in program memory). When the subroutine completes its execution, the program transfers program control back to the instruction that follows the instruction that invoked the subroutine (the calling instruction).

At the end of the subroutine code, there is usually some type of *return* instruction; we'll deal with the official syntax and underlying details later. This instruction indicates that the program completed execution of the code associated with the subroutine and program control should now return to the instruction immediately following the one after the instruction that initiated the subroutine (a call or unconditional branch instruction).

Note that there is an important relationship between the instructions that initiate a subroutine and the instructions that initiate a return from subroutines. Roughly speaking, you must return from every subroutine you initiate in the order you initiate them in. If you violate this universal constant, your programs won't execute properly.

12.4 Subroutines on the RISC-V MCU

Let's start this discussion by showing some examples of subroutines and providing detailed descriptions of those examples. You'll surely see that subroutines are generally not a big deal, but there are some special issues that programmers need to know so they can write working programs.

Example 12.2

Write a subroutine that swaps the values in register x8 & x9.

Solution: Figure 12.5 shows an example of our first subroutine. There are many good subroutine formatting examples in this solution; you should strive to adopt them in your code. Here are the pertinent points to note about the solution:

- We nicely delineate the subroutine from other parts of the code with comments. The delineating comments include an informative file banner and a line of dashes to indicate the end of the subroutine.
- The subroutine has a banner that includes important information about the use of the subroutine. This includes the subroutine name on line (01) and a description of what the subroutine does on line (03). The banner also includes a list of registers that the program changes. You would expect a subroutine to change the values in the registers being swapped, but the subroutine also changes another register.
- The name of the subroutine is a label, no different from other labels we've been using up to this point. Line (07) has the name of the label. Line (08) also has another label name, which we use for clarity. Most subroutines have some type of initialization code; even the subroutine in this example does not have such code, we use "init" label for consistency. The "init" label is optional, but it's always good to use.
- The main function of the code is to swap data in the two registers. There are many ways to do this; the code opts to temporarily store one of the values in another register, which we commonly refer to as a "working register". This means that the program overwrites the value in x10, which may be an issue. We'll deal with this issue in a later section.
- The code ends with a **ret** instruction on line (12). This is actually a pseudoinstruction; we'll deal with the details in another section.

```

(00) #-----
(01) # Subroutine name: Swap_reg
(02) #
(03) # This subroutine swaps the values in x8 & x9.
(04) #
(05) # Tweaked Registers: x8,x9,x10
(06) #-----
(07) Swap_reg:
(08) init:   mv    x10,x9      # copy data in x9 to working register
(09)        mv    x9,x8       # copy data from x8 to x9
(10)        mv    x8,x10     # copy working data to x8
(11)
(12)        ret   # transfer program control back
(13) #-----

```

Figure 12.5: The solution to this example problem.

Figure 12.6 shows a flowchart modeling the solution to this example. This flowchart is the first flowchart where we use two terminal symbols. Subroutines, unlike programs, have the notion of “ending”, which is why we include an ending type terminal symbol (the one with the “return” text) in the diagram. Keep in mind that subroutines have one entry point (so one start-type terminal symbol), but can have multiple exit points, so there is no limit to the number of ending-type terminal symbols we can use to represent a subroutine. You’ll see that in later examples.

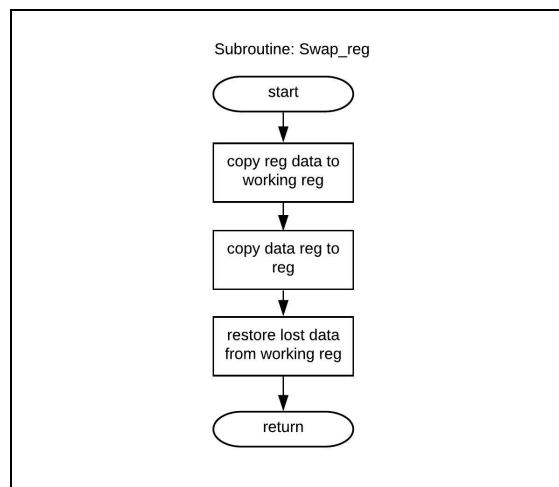


Figure 12.6: A flowchart modeling the operation of this example program.

Example 12.3

Write a subroutine that swaps the values in register x8 & x9. Don’t change any register values other than x8, x9, and sp.

Solution: This is the same example, but now with a constraint that we can’t change any register values other than two registers to swap and sp. The mentioning of sp is a hint that we’ll do this swap using the stack. Figure 12.7 shows the solution to this example. Here a few other good things to note about the solution:

- The subroutine uses comments to delineate and provide the human reader with information about the subroutine.

- We first made room for the two register values on line (08); we followed that with what are effectively two pushes onto the stack (x8 & x9). We then purposely pop the two registers off the stack in a different order to implement the required swap. The out-of-order popping of data is generally a mistake, but we use it to accomplish the goals of the problem.
- Even though the code in this example is functionally equivalent to the code in the previous example in that the subroutines do the exact same things, there are important differences. There is always a tradeoff when programming, the tradeoff in this example is that the previous example requires less time to run (because it has less instructions), but it uses one more register (x10) than the current example. Then again, the code in this example changes memory (in the stack), which is generally much less of a deal than changing the code in a register as the first example did. The stack exists for such operations and there is a lot of memory there as opposed to registers, where there are only 31 of them (at most) that we can use for general-purpose storage.

```

(00) #-----
(01) # Subroutine name: Swap_reg_stk
(02) #
(03) # This subroutine swaps the values in x8 & x9.
(04) #
(05) # Tweaked Registers: x8,x9
(06) #-----
(07) Swap_reg_stk:
(08) init:      addi  sp,sp,-8      # create room on stack
(09)           sw    x8,0(sp)      # save value on stack in order
(10)           sw    x9,4(sp)
(11)
(12)           lw    x8,4(sp)      # remove values off stack out of order
(13)           lw    x9,0(sp)
(14)
(15)           addi  sp,sp,8       # return sp to original value
(16)
(17)           ret                    # transfer program control back
(18) #-----

```

Figure 12.7: The solution to this example problem.

Figure 12.8 show a flowchart modeling the solution. The interesting thing to note about this flowchart is that we go out of our way to keep the flowchart independent of the computer the associated code will be implemented on. Thus, the process boxes in Figure 12.8 have relatively high-level descriptions of the code.

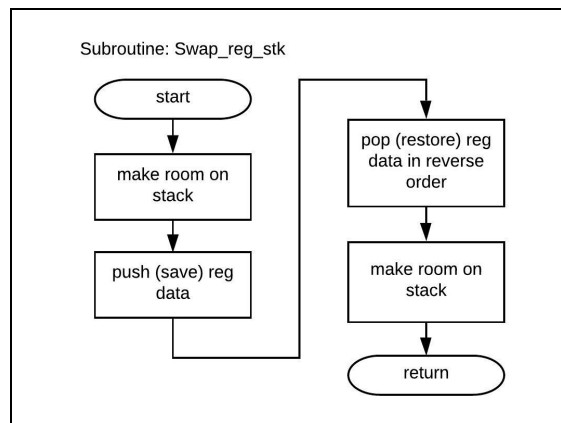


Figure 12.8: A flowchart modeling the operation of this example program.

12.4.1 Calling Subroutines and Returning from Subroutines

Calling subroutines and returning from subroutines is a topic we first mentioned in section 10.4.2.1; we'll fill in more information in this section. Similar to pushing and popping operations, there are no base instructions dedicated to calling and returning from subroutines. However, lucky for us, we don't need to deal with the low-level details because there are pseudoinstructions dedicated to the use of subroutines. Because of the general usefulness of these pseudoinstructions, we'll direct most of this discussion to the pseudoinstructions.

We prefer to use the **call** and **ret** pseudoinstructions to call and return from subroutines, respectively. The assembler translates these pseudoinstructions to the **jal** and **jalr** base instructions, which is a detail that pure programmers do not need to know. We'll discuss the underlying hardware implementation details in the RISC-V hardware portion of this text.

Table 12.3 lists the **call** and **ret** pseudoinstructions along with some other information. The information in Table 12.3 that is most useful to us the "Example Usage" column. We as programmers don't need to know that underlying base instructions that the assembler uses to implement the pseudoinstructions.

Instruction Form	Equivalent Base Instruction(s)	Example Usage	Comment
call rd,lab	auipc rd,hi{lab} jalr rd,lo{lab}(rd)	call x5,subrut	Jump to instruction associated with label; Store current address in rd
call lab	auipc x1,hi(lab) jalr x1,lo(x1)	call subrut	Jump to instruction associated with label; Store current address in x1
ret	jalr x0,0(x1)	ret	Jump to instruction at address in x1

Table 12.3: The program flow control pseudoinstructions and their base instruction translations.

Figure 12.9 shows an example of the both the use of the **call** and **ret** pseudoinstructions. Here are some extra important details:

- There is a fragment from the calling code on lines (00-04); this fragment does not show the code that places meaningful values in the x8 & x9. Line (02) shows the actual **call** instruction.
- When the program executes the call instruction on line (02), program control transfers to the instruction associated with the "Swap_reg" label; that instruction is on line (14). Because there is no instruction on the same line as the "Swap_reg" label, the label takes the value of the address of the next instruction, which is on line (14), which has the "init" label. In this way, the numeric values associated with the "Swap_reg" and "init" labels are equivalent (a detail the pure programmers don't need to know).
- When the program executes the **ret** pseudoinstruction on line (18), program control transfers back to the instruction following the **call** instruction. No, there is no such instruction in the fragment in the calling code of this example. The important thing to note here is that the **ret** instructions transfers program control to the instruction following the **call** instruction.

```

(00) #----- a fragment from the calling code
(01)
(02)         call    Swap_reg      # perform register swap
(03)
(04) #----- a fragment from the calling code
(05)
(06) #-----
(07) # Subroutine name: Swap_reg
(08) #
(09) # This subroutine swaps the values in x8 & x9.
(10) #
(11) # Tweaked Registers: x8,x9,x10
(12) #-----
(13) Swap_reg:
(14) init:      mv     x10,x9      # copy data in x9 to working register
(15)           mv     x9,x8       # copy data from x8 to x9
(16)           mv     x8,x10      # copy working data to x8
(17)
(18)           ret     # transfer program control back
(19) #-----

```

Figure 12.9: Example usage of the call & ret pseudoinstructions.

12.4.2 Passing Values to Subroutines

The notion of “passing values” comes up quite often when programming computers. This is simply a matter of “what you send and how you send something” and “what and how that thing sends something back”. In higher-level languages, this primarily means the stuff you send to functions (formal parameters) and the stuff the function sends back (return values). For the RISC-V MCU, it simply means what data the subroutine requires and how the calling code sends data to the subroutines that the subroutine expects, and what data and how the subroutines returns data from the subroutine. As it turns out, this problem is not complex with the RISC-V MCU; it’s primarily a notion of learning the standard terminology when working the subroutines.

There are generally only two ways to pass data to subroutines in the RISC-V MCU: via registers or via main memory. Note that both of these items are types of storage on the RISC-V MCU. The truth is that there is only one way, which is registers, but we refer to this as two approaches depending on the meaning of the data in a register. Sometimes the data is pure data, which means we are passing data to the subroutine in registers. Other times the data in the registers is a memory address; in this case, we refer to this as passing addresses to subroutines using main memory.

Figure 12.10 shows the solution to a previous example. This subroutine swaps data between two register: x8 & x9. The calling code (the code that calls this subroutine) effectively passes data to the subroutine in register, namely x8 & x9. The subroutine thus expects to find data in those registers, so it’s up the programmer to put the correct data in those registers before code calls the subroutine.

```

(00) #-----
(01) # Subroutine name: Swap_reg
(02) #
(03) # This subroutine swaps the values in x8 & x9.
(04) #
(05) # Tweaked Registers: x8,x9,x10
(06) #-----
(07) Swap_reg:
(08) init:      mv     x10,x9      # copy data in x9 to working register
(09)           mv     x9,x8       # copy data from x8 to x9
(10)           mv     x8,x10      # copy working data to x8
(11)
(12)           ret     # transfer program control back
(13) #-----

```

Figure 12.10: A subroutine that uses data passed by register.

Example 12.4

Write a subroutine that swaps the words in two different memory locations. The locations of the data to swap are provided in registers x6 & x7.

Solution: This is yet another version of swapping something, which seems to work rather well for these introductory examples. Figure 12.11 shows the solution to this example; here are some other fun facts to fill your head:

- First, this is a bad solution. We present it because it does a great job of illustrating a point in the following section.
- The subroutine must be passed to values in x6 & x7. Because these two registers hold address information, we are passing data to the subroutine via memory. Sort of a fine line with that definition, but it works.
- The problem with the solution is the code on lines (12-14). This code is not actually necessary because all we need to do is use the sw instructions to save the registers at different address locations, meaning we would swap the base register values for the instructions on lines (16-17).

```
(00) #-----
(01) # Subroutine name: Swap_mem_w
(02) #
(03) # This subroutine swaps two word values in memory. The address of the
(04) # values to swap is found in register x6 & x7.
(05) #
(06) # Tweaked Registers: x10,x11,x12
(07) #-----
(08) Swap_mem_w:
(09) init:      lw    x10,0(x6)      # load the data to swap
(10)          lw    x11,0(x7)
(11)
(12)          mv    x12,x10        # copy data in x10 to working register
(13)          mv    x10,x11        # copy data from x11 to x10
(14)          mv    x11,x12        # copy working data to x11
(15)
(16)          sw    x10,0(x6)      # store the swapped data
(17)          sw    x11,0(x7)
(18)
(19)          ret                    # transfer program control back
(20) #-----
```

Figure 12.11: A subroutine that uses data passed by address.

Example 12.5:

Write a RISC-V assembly language subroutine that swaps the data in two registers. Do not change any memory other than the values in those two registers. Consider the registers with the data to be X10 and X11.

Solution: The solution to this problem is a well-known digital “trick”. The solution is hard to understand (but easy to apply), so plan on putting this in your bag of digital tricks because swapping data in two registers is a common occurrence in assembly language programming. Here is some fun stuff embedded in the solution:

- The `Swap_in_place` label on line (12) is the name of the subroutine, which of course provides an idea to human readers what the subroutine is doing.

- The `xor` instruction has three operands: it performs a bitwise exclusive OR on the data in the registers specified by the two right-most (x10 & x11), and stores that data in the register associated with the left-most operand. There is nothing magic about this; this is the way the register operands are accessed by the instructions. Note that someone needs to tell you this (or you need to read the spec); you would not know otherwise.
- Yes, great interview question. The XOR function is somewhat magical; the magic displayed in this problem is how RAID arrays work (look it up).
- The selection of x10 and x11 registers is arbitrary; recall these are all “general purpose” registers.
- The code nicely aligns all the instructions and comments and in the subroutine banner.

```

(00)  #-----
(01)  # Subroutine name: Swap_in_place
(02)  #
(03)  # This subroutine swaps the values in x10 & x11 but does not change
(04)  # any other register or memory values.
(05)  #
(06)  # Tweaked Registers: x10,x11
(07)  #-----
(08)  #-----
(09)  #- Code Fragment: does "in-place" swap of data in two registers
(10)  #-----
(11)
(12)  Swap_in_place:  xor    x10,x10,x11    # three xors; get used to it
(13)                   xor    x11,x11,x10    # well-known trick
(14)                   xor    x10,x10,x11
(15)
(16)  done:          ret                    # take it on home

```

Figure 12.12: Solution to this example problem.

12.4.3 Saving Context in Subroutines

The subroutine represents a different piece of code from the calling code. This being the case, the subroutine may inadvertently modify a register that the calling code is currently using, which would mean a slow death for your program³. To make subroutines more useful to programmers, we typically write subroutines to be independent of the calling code, which we do by saving the operating context of the MCU before we start executing the subroutine. What we mean by this is that we want to write subroutines that we can call and not worry about the subroutine altering a register currently being used by the calling code.

The notion of “saving context” is quite popular in low-level programming. You probably don’t realize it, but your higher-level language compiler is responsible for saving “various contexts” when the call functions⁴. However, because we’re dealing with programming at a low level (the assembly language level), the programmer must be aware of and handle such details. This is actually not a large undertaking, as we don’t have to save the entire operating context of the MCU; we only need to save and restore the registers changed (not used) by the subroutine as the calling code may be using these registers.

We’ve sort of provided warnings regarding this subject in our past subroutines. Note that the subroutines plainly state which registers the subroutine changes in the subroutine header. This is good practice, for sure. Better practice is to simply save the registers the subroutines uses at the beginning of the subroutine (referred to as *saving context*) and then restoring those registers values before the subroutines returns control to the calling code (referred to as *restoring context*). By far the most straightforward way for subroutines to save the context is to push the registers that the subroutines modifies in the body of the subroutine onto the stack at the beginning of the subroutine, then popping them off the stack back into the original registers before the subroutine returns control to the calling code.

³ Dead programs, or any programs that do not work, are bad things.

⁴ This is a deep but important subject; this text does not delve into the details.

Example 12.6

Write a subroutine that swaps the words in two different memory locations. The locations of the data to swap are provided by the addresses registers x6 & x7. Make sure the subroutine does not permanently change any register value.

Solution: This solution once again uses the bad code from the previous solution, but with a good reason. The code in the previous version of this subroutine changed three register values, which means the programmer must make sure that those three registers are not currently being used by the calling program. This is a lot to ask, particularly when programs become complicated. The better solution is to know that you can call the subroutine without affecting the calling program in a detrimental way. Good programmers generally write subroutines in this way; we refer to this approach as the subroutine *saves context* before it does what it needs to do, then *restores context* afterwards. I personally like referring to this as making the subroutines “bulletproof”⁵. Here is some other stuff to note about the solution in Figure 12.13:

- We changed the subroutine name so that it is different from the previous similar solution. We also changed the “Tweaked Registers” comment on line (06) to indicate that the subroutine does not permanently change any register.
- We save context by pushing each register the subroutine uses at the start of the subroutine. We need to push three registers, so we make space on the stack by reducing the stack pointer by 12 on line (09). We follow that operation with three **sw** instructions, which serve to store the three registers on the stack. The overall approach is to first subtract 12 from the **sp**; the **sp** then points at an unused memory address. We push the three pieces of data starting at that address as indicated by the “0” in the offset box in the **sw** instruction on line (10). We do the same for the following two pushes, but we advance the pointer by four using the offset value in the following two **sw** instructions.
- The body of the code is similar to previous examples so we won’t describe it again here.
- Once the subroutine completes the main part of the work, we must restore the context, which we do by popping data off the stack and back into the registers from which that data originated (where we pushed it at the beginning of the program); We do this on lines (24-26). Note that we use a “restore” label on line (24), which alerts the astute human reader as to what the code is doing in an abbreviated format.
- After restoring the data by popping it off the stack, we then adjust the stack pointer back to where it was before we stored context. Thus, line (27) undoes the instruction on line (09). For that matter, lines (24-26) undo the instructions on lines (10-12).

⁵ There are other things programmers do to make their subroutines bulletproof; we’ll discuss those things in a later section.

```

(00) #-----
(01) # Subroutine name: Swap_mem_ws
(02) #
(03) # This subroutine swaps two word values in memory. The address of the
(04) # values to swap is found in register x6 & x7.
(05) #
(06) # Tweaked Registers: none
(07) #-----
(08) Swap_mem_ws:
(09) init:      addi  sp,sp,-12      # make room on stack for storage
(10)           sw    x10,0(sp)      # push 3 items on stack
(11)           sw    x11,4(sp)
(12)           sw    x12,8(sp)
(13)
(14)           lw    x10,0(x6)      # get data to swap
(15)           lw    x11,0(x7)
(16)
(17)           mv    x12,x10        # copy data in x10 to working register
(18)           mv    x10,x11        # copy data from x11 to x10
(19)           mv    x11,x12        # copy working data to x11
(20)
(21)           sw    x10,0(x6)      # store swapped values
(22)           sw    x11,0(x7)
(23)
(24) restore:  lw    x10,0(sp)      # pop data into register
(25)           lw    x11,4(sp)
(26)           lw    x12,8(sp)
(27)           addi  sp,sp,12       # unadjust the stack pointer
(28)
(29)           ret   # transfer program control back
(30) #-----

```

Figure 12.13: A subroutine that uses data passed by address.

12.4.4 RISC-V and Nested Subroutines

The underlying mechanism in the approach the RISC-V MCU uses to call subroutines has some special issues that pure programmer needs to be aware of. Many of these details are associated with the underlying hardware, but we present a working overview here so that programmers can write viable code.

The issue is simple: when you call a subroutine that in turn calls another subroutine, you have to do some special things to make your program work properly. We refer to a subroutine that calls another subroutine as a “nested” subroutine, a topic we’ll discuss further in a later section. This section presents the relatively simple mechanism of making nested subroutines work on the RISC-V MCU.

When the program calls a subroutine, the underlying hardware needs to store the address of the instruction following the subroutine call “somewhere”, because this is the instruction that executes after the return from subroutine instruction (**ret**). The mechanism employed by the RISC-V MCU is to store that address in a specific register, which is somewhat arbitrarily, **x1**. This being the case, the **x1** register has an alternate name of “**ra**”, which conveniently stands for “return address”. When a subroutine is called, the underlying hardware places the address of the instruction after the call instruction into **ra**. When the subroutines completes, it transfers program control back to the calling program by making the instruction stored in **ra** to be the next instruction executed⁶.

The problem with nested subroutines exists because it is most convenient to use the same register (**ra**) for all return addressed. The true issue is that when a subroutine calls another subroutine, the hardware automatically overwrites the **ra** with the return address of the newly called subroutine. This means that if you structure your code to nest subroutines, you first must save the return address of the calling subroutine before that subroutine calls another subroutine. The way to save the return address is to push that address on the stack before the nested subroutine call and then pop it off the stack back into **ra** after the nested subroutine returns. This is not a big deal to implement in your code, which is good because you must do this to make your code work properly.

⁶ Don’t worry: the specific actions of the underlying hardware is much more interesting than this written description.

Example 12.7

Modify the previous subroutine that swaps data in two memory locations such that the actually register swapping portion of the code is done with a nested subroutine. Make sure the subroutines don't permanently change any register values.

Solution: This is the bad solution that won't go away. What makes this solution bad now is that we call a three-instruction subroutine; the overhead associated with such a short subroutine indicates that we should probably put the code inline rather than call a subroutine. However, for this problem, efficiency does not matter, as our intent is to show the special issues involved with using nested subroutines. Figure 12.14 shows the solution to this example; most of the stuff is similar to where we earlier described this solution, so we won't describe that stuff again here. However, there is still some other fun stuff to take note of:

- The problem stated to not permanently change any register, so we push two registers on the stack as part of the initialization code on lines (09-11). This code looks a bit strange because we reserve space for three registers when we adjust the stack pointer on line (09), but we only save two registers. This will make more sense later in the code.
- The “Swap_mem_wsx” subroutine calls another subroutine so we list the nested subroutine at the end of the listing. We named the new subroutine “R_swap” because it swaps the data in two registers. The subroutine implements the swap using the infamous XOR in-place register swap; you can find a complete description of this algorithm and subroutine in the chapter with solved problems. This subroutine has a nice banner describing listing the subroutine name, describing what the subroutine does, and lists which registers the subroutine changes.
- The call to the nested subroutine appears on line (17). Before we make that call, we need to save **ra**. The current value in **ra** is the return address associated with the call to the “Swap_mem_wsx” subroutine. When we call the “R_swap” subroutine using the call pseudoinstruction, **ra** is written with a new return address, which is the address of the instruction on line (18). We save **ra** on line (16) by pushing it on the stack; recall that we already saved space on the stack with the instruction on line (12). When we return from the nested subroutine, we then pop that original value off the stack back into **ra** on line (18). We can actually do the pop that restores **ra** any time before the ret instruction, but we choose to do it after the call so we don't forget and then create an ugly bug in our program.
- Placement of the **ra** saving and restoring mechanism is always an issue. If your subroutine only contained one nested subroutine, it would make sense to place storing **ra** in the initialization section of the code, and restoring the original **ra** somewhere near the end of the code (both of these items would fit nicely into the context saving and storing code). The issue is what happens if a subroutine call contains two different nested calls? In this case, it would make sense to place the **ra** saving/restoring mechanism near the actual nested calls.
- Though it may seem a bit strange, this solution works. There are other approaches to protecting the return address when nesting subroutines, but this is the most straightforward approach, particularly for people new to assembly language programming.

```

(00) #-----
(01) # Subroutine name: Swap_mem_wsx
(02) #
(03) # This subroutine swaps two word values in memory. The address of the
(04) # values to swap is found in register x6 & x7.
(05) #
(06) # Tweaked Registers: none
(07) #-----
(08) Swap_mem_wsx:
(09) init:      addi  sp,sp,-12      # make room on stack for storage
(10)           sw    x10,0(sp)      # push 3 items on stack
(11)           sw    x11,4(sp)
(12)
(13)           lw    x10,0(x6)      # get data to swap
(14)           lw    x11,0(x7)
(15)
(16)           sw    ra,8(sp)       # push current ra on stack
(17)           call  R_swap        # do the register swap
(18)           lw    ra,8(sp)       # pop old ra back into ra
(19)
(20)           sw    x10,0(x6)      # store swapped values
(21)           sw    x11,0(x7)
(22)
(23) restore:  lw    x10,0(sp)      # pop data into register
(24)           lw    x11,4(sp)
(25)           addi  sp,sp,12      # unadjust the stack pointer
(26)
(27)           ret                    # transfer program control back
(28) #-----
(29)
(30) #-----
(31) # Subroutine: R_swap:
(32) #
(33) # This subroutines swaps the values in x10 & x11 (in-place reg swap)
(34) #
(35) # Tweaked Registers: x10, x11
(36) #-----
(37) R_swap:   xor    x10,x10,x11    # three xors; get used to it
(38)           xor    x11,x11,x10
(39)           xor    x10,x10,x11
(40)           ret                    # pass flow control back
(41) #-----

```

Figure 12.14: A subroutine that calls a subroutine (nested subroutine call).

Figure 12.15 shows the flowchart that models this solution. This solution is new in that this subroutine contains a nested subroutine call. We indicate the subroutine call in the flowchart with the “predefined process symbol”, which is essentially a process box with extra vertical lines on the sides. The vertical lines make it painfully obvious that this is not the normal process box. We also opted to include a flowchart for the nested subroutine as part of this flowchart.

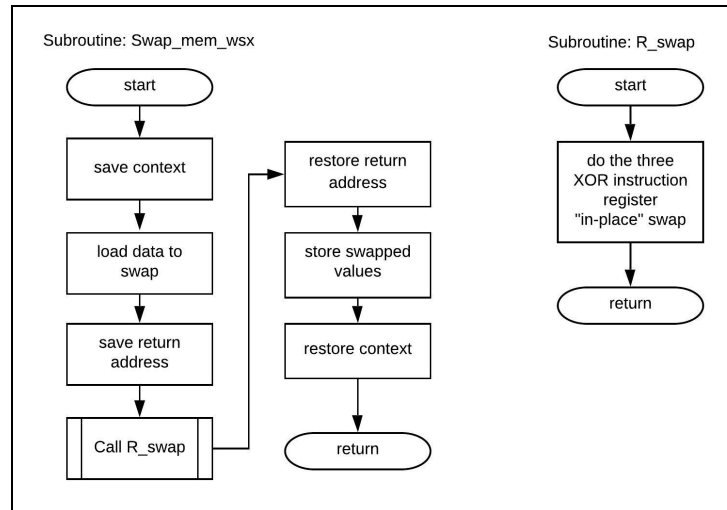


Figure 12.15: A flowchart modeling the operation of this example program.

Example 12.8

Write a RISC-V MCU assembly language subroutine that divides a 2-digit BCD value by 2. The value is sent to the subroutine in `x20`, where the lower two nibbles form the 2-digit BCD value. The halved value is returned to the subroutine in `x20`. Be sure to round up the value in `x20` it needs it (round up when the pre-shifted value in `x20` is odd).

Hint: you need to deal with the LSB of each BCD value.

Solution: This solution does not require nested subroutines, but it does give an example of using the `slt`-type instructions as well as implementing an algorithm. When we divide a binary number by two, we simply shift it right one bit position. The right shift effectively truncates the result, so we sometime want to round the result upwards. The general rounding operation is done when the least significant digit of a number is five or greater. For this problem, when the pre-shifted LSB of the 1's digit is '1', then we need to add a 1 back to the final result for the rounding operation.

It turns out that a similar thing happens for the 10's digit: when the LSB of the 10's digit is '1', that means we lose that value in truncation. To account for this, we add back 5 to the final result, which is of course half of ten. In other words, if we shift off the LSB of the 10's digit, it changes the value of the final result.

Yes, sort of an ugly algorithm. This works great for two digits, but you may consider doing another algorithm if you had to divide a 16 digit number by 2. In that case, you may first want to convert the number from BCD to binary, shift it right one position, add back the LSB, and convert the number back to BCD.

Figure 12.16 shows a solution this problem; below is some verbose description.

- The subroutine header has all the pertinent information for the subroutine including a rough description of the algorithm the subroutine uses. This is good programming practice.
- The subroutine first clears off any data that maybe in the top six nibbles of `x20` with the mask operation on line (14). The subroutine then makes two copies of the value for use later in the subroutine. Note that I did not know I needed these registers until I started coding the algorithm; I put the code on lines (15-16) later in the solution process.

- I want to perform the divide by a shift right, but I need to modify the data first by clearing the LSB of each digit with the mask operation on line (18). The value is ready to divide, which I do on line (22). I then mask and save the LSBs of each digit on line (19-20) for use later in the subroutine.
- Line (24) checks to see if the LSB of the 10's digit was zero; if it was, then it adds 5 (which is half of ten) to the result on line (26). The program then adds the 1's LSB, which is effectively the round up value (which could be zero, no big deal).
- We potentially added two values to the shift result, which means the lower nibble could be greater than nine. If it is, we need to increment the 10's digit (line (43)) and subtract ten from the 1's digit (line (42)). Note that on line (30), we use a `slti` instruction to determine if the value is less than ten or not. This instruction is handy because it allows us to compare a register with an immediate value, which saves up initially placing the value to compare into a register.

```

(00) #-----
(01) # Subroutine name: Half_BCD2
(02) #
(03) # This subroutine divides a two digit BCD number passed to the subroutine
(04) # in x20 by 2. This subroutine rounds the rounds up when fractional part
(05) # of result is 0.5 or greater. The algorithm uses a shift right instruction
(06) # to do the division, which means a 1 shifted out of the LSB location of
(07) # the 10's digit causes a 5 to be added to the lower digit.
(08) #
(09) # Passed values: x20 (data to half)
(10) # Returned values: x20 (result)
(11) # Tweaked Registers: x10,x11,x30
(12) #-----
(13) Half_BCD2:
(14) init:      andi    x20,x20,0xFF    # ensure it's only two nibbles
(15)           mv     x10,x20        # make local copy
(16)           mv     x11,x20        # make local copy
(17)
(18) prepare:  andi    x20,x20,0xEE    # mask two low nibbles & LSBs
(19)           andi    x10,x10,0x1    # mask LSB of low nibble
(20)           andi    x11,x11,0x10   # mask LSB of hi nibble
(21)
(22)           srlr   x20,x20,1      # divide by 2
(23)
(24)           beq    x11,x0,round    # jump over 1's adjust (+5)
(25)
(26) round:    addi    x20,x20,0x5    # add five: half of 10's LSB
(27)           add     x20,x20,x10    # add roundup bit
(28)
(29) chk_1:    andi    x30,x20,0xF    # mask 1's digit
(30)           slti   x30,x30,0xA    # set if no adjust needed
(31)
(32) adjust:   bne     x30,x0,done    # branch if no LSB
(33)           addi   x20,x20,-10     # adjust lower LSB by 5
(34)           addi   x20,x20,0x10    # increment 10's digit
(35)
(36) done:     ret     # take it home leroy

```

Figure 12.16: A subroutine that calls a subroutine (nested subroutine call).

Figure 12.17 shows a flowchart modeling this solution. This could possibly be too much fun stuff for one problem where the code and the comments included in the code still don't describe the algorithm in enough detail to understand completely. In this case, the flowchart is particularly helpful.

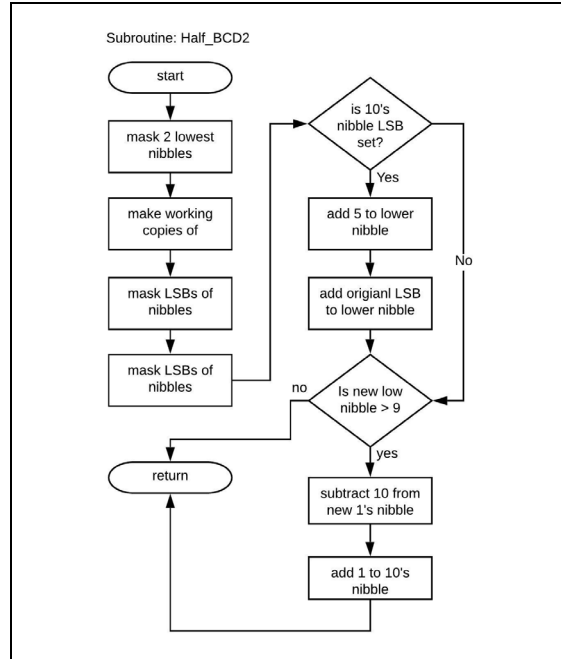


Figure 12.17: A flowchart modeling the operation of this example program.

12.5 Special Subroutine Issues

There are a few other issues regarding subroutine vernacular and usage that programmers need to be aware of. Subroutines are great when used properly, but can be a source of intermittent errors if programmers use them improperly. We all know that intermittent errors are typically the hardest errors to find.

12.5.1 Recursive Subroutines

When we think of subroutines, we generally think of the calling code and the subroutine itself as separate entities, but this is not always the case. There is no reason a subroutine cannot call itself. When a subroutine calls itself, we refer to it as a *recursive subroutine call*, or simply *recursion*. We consider recursion a special type of nested subroutine. Recursion is a special animal in that it's somewhat hard to comprehend and is even trickier to actually use properly in a program. Although we don't generally use recursion, there are times when it's the most straightforward approach to implementing algorithms⁷.

You can better understand recursion if you understand the underlying hardware implements subroutines, namely how the RISC-V hardware handles `call` and `ret` instructions. We don't go there in this chapter because we are purposely excluding hardware details in order to deal with details pure programmers need to know. My personal thoughts about recursion are that you should avoid it at all costs. If you can't avoid it, make sure you understand it well enough to ensure it works properly in your code. There are two issues to be aware of when using recursive subroutines.

- 1) There are generally limits to the levels of recursion you can have based on the notion of subroutines saving the operating context. In addition, because recursive subroutines are inherently nested, programmers need to save the return address (`ra`) on each level recursive level. In other words, use of recursion can bring up stack integrity issues, which if not properly handled, can doom your program.
- 2) Recursion must have stop conditions. The issue here is that stop conditions in recursive subroutines are more "tricky" than stop conditions in other items such as iterative loop. This means you really have to know what you're doing to properly use recursion.

⁷ Once such situation is implementing factorial algorithms.

12.5.2 Stack Overflow

The description of stacks implemented with a structured memory device such as a RAM leaves open the option for improper stack operations. The general rule when working with stacks is to keep the pushes “synchronized” with the pops. This means that if your subroutine saves operating context by pushing five items on the stack, it needs to pop five items off the stack before it returns. This also means when you nest subroutines, you must properly un-nest them as well.

When you follow the proper stack protocol, you should attempt to write anything outside of the bounds of the stack segment. Another way to state this is that the stack pointer should always point to a location in the memory area designated for the stack⁸. If you don’t follow this protocol, you then have a *stack overflow* problem, which typically means a slow if not immediate death for your program. Programmers can avoid stack overflow problems by ensuring their code follows these two rules: 1) never push so many items onto the stack such that the stack pointer exceeds the designated stack area in memory, and 2) ensure that they write their code such that the push and pops are equally paired. Another way of saying the previous item is that for every push in your program, there needs to be a corresponding pop. Keep in mind the issues of pushing and popping primarily have to do with nested subroutine calls, and saving/restoring context associated with making subroutines safe. Recall, this stack is located in memory and that we can model the stack pointer as a counter that increments and decrements.

The stack implementation in the RISC-V MCU has issues programmers need to be properly handl. Because the RISC-V does not have actual push and pop instructions, we rely on adjusting the stack pointer under program control, which means by issuing an add-type instruction to manually adjust the stack pointer. We subtract from the stack pointer with pushes (to support the notion that the stack grows in the negative direction) and add to the stack pointer with pops. This being the case, we have to ensure we eventually match the two directional adjustments to the stack pointer.

Messing up the stack in one of these ways is typically an error that is hard to find, particularly since the problems it causes can be intermittent. As with any error in your code, the most likely time it fails during a customer demo. It is possible to set up checks in software to ensure the integrity of the stack, but these approaches take up codespace and reduce overall runtime efficiency. The better solution is to write good code that naturally supports stack integrity.

The stack can overflow in one of two directions, which brings up the notion of stack overflow and stack underflow. The truth is that whether a stack is overflowing or underflowing is a semantic issue, which we bypass by including both overflow and underflow in the definition of stack overflow.

12.5.2.1 Subroutines and Stack Overflow

Subroutines are a major part of writing modular and reusable assembly language code. When programmers use subroutines, they are responsible for retaining integrity of the stack. When programmers nest subroutines, the must save the return address before calling the nested subroutine; the customary approach to saving the return address is by pushing it on the stack before the subroutine call and popping it off the stack back into the return address register after returning from the nested subroutine. A stack overflow problem exists when the stack operations associated with calling and returning from nested subroutines are not “paired”, which is another way of saying there is not a pop for every push associated with the nested subroutines. This same condition is associated with recursive subroutines.

The stack overflows in the direction of larger magnitude memory addresses when the number of returns from subroutines are greater than the number of subroutine calls. The stack overflows in the other direction when the number of subroutine call is greater than the number of subroutine returns. You may be thinking how such a situation may arise; the answer is that is happens in three situations, all of which are common with nubile assembly language programmers.

- 1) Stack overflow happens is when programmers branch or jump to a subroutine rather than calling that subroutine. This results in the MCU executing a return instruction without a corresponding call instruction. In this case, there is not a valid number in the return address.

⁸ Actually, when the stack is empty, it’s customary to point at an address outside of the stack because the the stack pointer needs to be adjusted (made to be a smaller value) before the program pushes a value onto the stack.

- 2) Stack overflow happens is when a programmer calls a subroutine only to later exit that subroutine with a branch or jump instruction rather than a corresponding return instruction.
- 3) Stacks can overflow when the subroutines nest too deeply, which includes recursive subroutine calls. Another way to say this is that the program issues too many subroutine calls without issuing and return from subroutines. Note that it is possible to overflow the stack with too many subroutine calls and still have your program work. This is possible if the stack pointer wanders into memory space that other parts of the program are not currently using. This issue of course depends upon how the programmer structures their code and initially configures the stack pointer.

12.5.2.2 Context Saving and Stack Overflow

The other common use of the stack is to save operating context upon entering a subroutine or interrupt service routine⁹ and restoring context upon learning those sections of code. Programmers typically save context as part of the initialization code of a subroutine, which obviously is at the start of the subroutine. This context saving code typically saves every register that the subroutine changes in order to make the call to that subroutine “safe” for the code that calls the subroutine. The subroutine then restores the saved registers before the subroutine returns, an operation that is typically the final task before the subroutine executes the return instruction. The key to ensuring the context saving mechanism never causes overflow (or any other problems) is to ensure the same registers that are pushed as part of context saving are later popped as part of context restoration. Note that not ensuring each push has a corresponding pop and each pop has a corresponding push will eventually cause stack overflow if your program does not die immediately.

One nice thing about the RISC-V ISA not having dedicated push and pop instructions is that the pushing and popping operations in context saving/restoring done have to be “in order”. Because the RISC-V implements pushes and pops with store-type and load-type instructions (with corresponding stack pointer adjustment under direct program control) respectively, pushes and pops in the RISC-V can be out of order and use the offset portion of the memory access instruction to target a specific address on the stack. Very handy.

12.5.3 Subroutine Overhead

The underlying problem with subroutine calls is that they also have “overhead” associated with them. The notion of “overhead” in this context is having the MCU execute an instruction that does not actually do anything useful for the given task. While we all know that we can use subroutines to keep our code well organized and efficient in terms of program memory, we can also abuse them. There are potentially two other forms of overheads associated with subroutines.

- 1) Subroutines typically save and later restore the operating context
- 2) Nested subroutines needing to protect return addresses

The issue of subroutine overhead is always something programmers need to consider. We cover this topic in greater detail in section 14.4.2.

12.5.4 Stack Initialization

All programs, particular programs associated with embedded systems, typically have some type of initialization code at the start of the program. We consider this code to be initialization code partially because we only need to run it once. This code is typically associated with placing external peripheral devices into a known operating state.

Another part of the initialization code is to put the MCU into a known state and to get the MCU ready to execute your program. You’ve seen this in many of the examples we’ve done up to this point. However, another thing you really must do: write a value to the stack pointer. Keep in mind that the RISC-V MCU is versatile enough to use most any register as the **sp**, but the best approach is to use x2 as the **sp**, and write a value to as part of your initialization code. When you write a value to **sp** (or x2), you’re officially declaring the top of the stack. The key here is to understand the memory map associated with your system, because knowing where the different parts of your program are located (such as the code and the stack) helps you optimize your system and avoid problems.

⁹ Interrupt service routines and the RISC-V OTTER’s interrupt architecture is the topic of Chapter 13.

12.6 Intelligent Subroutine Usage

Because there are not official constraints or rules regarding the use of subroutines, you should strive to follow a few basic guidelines when you use them.

- Subroutines should contain a piece of code that has some specific purpose. If each subroutine has a specific purpose, there is a greater chance you can reuse that code in another program. In addition, subroutines with specific purposes are easier to document and understand.
- All subroutines should be clearly delineated from other parts of the code by using an appropriate amount of comments. This promotes neatness and readability of your source code, which subsequently support humans striving to understand your code.
- Your subroutines should save the operating context at the start of the subroutine and of course restore it at the end of the subroutine.
- It's generally a good idea to put all your subroutines at the end of your source code as opposed to the beginning of our source code. Code with subroutines intermixed throughout the code makes the code hard for humans to read.
- All subroutines should contain a banner that provides the name of the subroutine, a description of what the subroutine does, a list of register arguments sent to the subroutine, and a list of what registers the subroutine modifies permanently modifies
- All subroutine banners should clearly list how the calling program sends data to the subroutine and how the subroutine returns data back to the calling code. This text doesn't always use this rule in an effort to save space and reading time. .
- Your subroutines should not be too short or too long. If your subroutines are too long, consider breaking them up into smaller subroutines that use nested subroutine calls. If your subroutines are too short, you stand the chance of having the overhead issues with your subroutine that make your code runtime inefficient (which is partially dependent on how often you call the subroutine). A general rule is that short subroutines are OK if they are called many different times from many different parts of the program.
- If you nest subroutines, you must protect the integrity of the **ra** register.

Example 12.9

Write a subroutine that multiplies the unsigned halfword in x8 with the unsigned half-word in x9 and stores the result in x10. Don't permanently change any register other than x10. Make sure your subroutine works in all cases. Write the subroutine with the thought the at least one of the operands will often be zero.

Solution: You'll quickly note that the RISC-V MCU instruction set does not include a multiply instruction. The solution therefore entails implementing multiplication by repeated addition algorithm. We've put every effort into making this solution as efficient and bulletproof as possible. Figure 12.18 show the solutions to this example; here are some other cool things to note about the description.

- We included more information in the subroutine banner that we did not include in our other solutions. For this solution, we've included what values the subroutines expects to be sent by the calling code, which we do on line (06).
- Our first task is to save context, which we do with stack operations on lines (10-12). We need to save these registers first because the next thing we do is mask them.
- We've opted to mask the two operands in order to clear the top two bytes of the registers. The problem stated that they were halfwords, but we want to make sure by clearing whatever value may reside in the top two bytes. We do this on lines (14-16).

- We clear the accumulator on line (17); the subroutine is now ready for an early exit. Because the program description stated that one of the operands will often be zero, we start the subroutine by checking those values for zero. If either value is a zero, we can then quickly exit the subroutine. We do this with the two conditional branch instructions on lines (19-20).
- The algorithm works by continually adding of the operands the number of times of the number in the other operand. At this point, we know that both operands are non-zero, so it does not matter how we use the operands in the solution. We encode this algorithm as a while loop, which is safe because before entering the while loop, we know both operands are non-zero. The entire body of the algorithm is on line (22). Loop administration include decrementing the operand we're using as an iterative count line (25), followed by a conditional branch to possibly exit the loop on line (26).
- When the code exits the loop, the answer is in x10. Our last task is then to restore context, which we do on lines (27-29).

```

(00) #-----
(01) # Subroutine: Mult_regs
(02) #
(03) # This subroutine multiplies the unsigned halfword values x8 & x9 with
(04) # each other and stores the results in x10.
(05) #
(06) # Passed values: x8 & x9
(07) # Tweaked Registers: x10
(08) #-----
(09) Mult_regs:
(10) init:      addi  sp,sp,-8      # make room on stack for storage
(11)           sw    x8,0(sp)     # push 2 items on stack
(12)           sw    x9,4(sp)     # push 2 items on stack
(13)
(14) mask:     li    x10,0x000FFFF # mask value
(15)           and  x8,x8,x10     # ensure values are halfwords
(16)           and  x9,x9,x10
(17) clr_acc:  mv    x10,x0       # clear accumulator
(18)
(19) chk_0:    beq   x8,x0,restore # return if either operand is 0
(20)           beq   x9,x0,restore
(21)
(22) loop:    add   x10,x10,x8    # add value (accumulate)
(23)
(24) admin:   addi  x9,x9,-1      # decrement other value
(25)           bnez  x9,loop      # branch if count non-zero
(26)
(27) restore: lw    x8,0(sp)      # pop data into register
(28)           lw    x9,4(sp)
(29)           addi  sp,sp,8      # unadjust the stack pointer
(30)
(31) done:    ret                    # transfer program control back
(32) #-----

```

Figure 12.18: A solution for this example.

Figure 12.19 shows a flowchart modeling this solution. This is a classic problem where the code and the comments included in the code still don't describe the algorithm in enough detail to understand completely. In this case, the flowchart is particularly helpful.

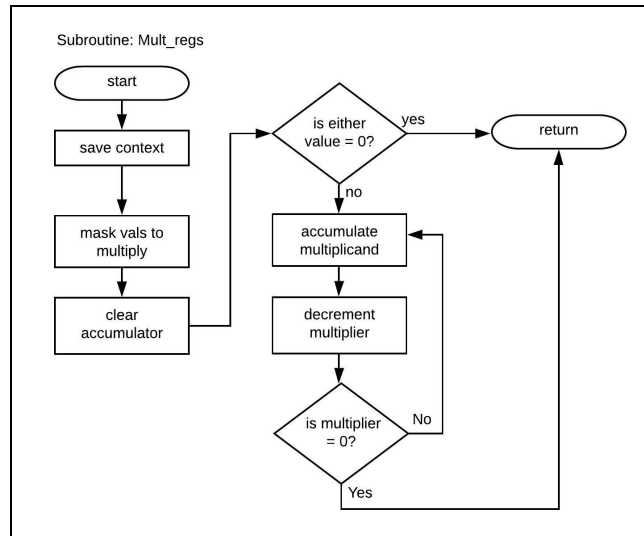


Figure 12.19: A flowchart modeling the operation of this example program.

Example 12.10: Gathering Statistics from Memory

Write a RISC-V assembly language subroutine that counts the number of non-zero values in a span of memory. The memory span is a contiguous set of unsigned bytes starting at the address x10, and checking the number of locations given by value in x15. The subroutine returns the final count in register x15. Don't allow the subroutine to permanently change any registers other than x15.

Solution: A classic subroutine that requires you go to memory and do something; in this case, we go to memory and collect statistics. Note that nothing in the problem states that we should change any value in memory, so we won't be doing that. here are some other cools things to note about the description.

- We included more information in the subroutine banner that we did not include in our other solutions. For this solution, we've included what values the subroutines expects to be sent by the calling code, what the subroutine returns, and the registers that the subroutine alters.
- Our first task is to save context, which we do with stack operations on lines (12-15). We don't know in advance that we need to save these registers; we actually write the context saving and restoring code when we complete writing all the other parts of the subroutine.
- Part of the initialization code is clearing a register for use to use as a counter, which we do on line (17). Because the count could be zero, we check the count first on line (19), which officially makes the loop in this subroutine a while-loop. Checking a count that could be zero in a subroutine is necessary because the value could be zero; subtracting one from zero would create tragic results in the subroutine.
- The body of the whole loop loads a byte of data from memory (20), then checks that value to see if it is zero on line (21) by using a classic if/else construct. If the value is zero, it branches over the counter increment instruction on line (22). Either way, the code makes it to the loop administration code on lines (24-25). The loop admin code include incrementing the address value by one on line (24); we use one because the problem is dealing with bytes. We then decrement the loop count on line (25).

- When the loop count runs to zero, the code exits the loop and drops to the instruction on line (28). This instruction transfers the count value to a register that the program uses to send the subroutine loop count value. We essentially reuse this register so that we don't need to include saving this register in the context storage/restoration parts of the subroutine.
- We restore context on lines (30-33). Note that we needed to save three registers, so we need to restore three registers as well (the same registers). Note that the ordering of register saving/restoring does not matter; the only two things that matter are that 1) the subroutine saves the proper registers, and 2) the offset part of the load-type and store-type instruction are the same per register. When you're writing code, the best approach is to cut-and-paste the context savings code to use as context restoring code, but be sure to change store-type instructions to load-type instruction and to change the sign on the instruction that adjusts the stack pointer.

```

(00) #-----
(01) # Subroutine: Count_zeros
(02) #
(03) # This subroutine counts the number of non-zero values appearing in a
(04) # contiguous chunk of memory (interpreted as bytes) starting at the
(05) # address stored in x10 and checking the number of bytes stores in x15.
(06) #
(07) # Passed values: x10 & x15
(08) # Returned values: x15
(09) # Tweaked Registers: x15
(10) #-----
(11) Count_zeros:
(12) init:      addi  sp,sp,-12      # make room on stack for storage
(13)           sw   x16,0(sp)      # push altered registers on stack
(14)           sw   x20,4(sp)     # push 2 items on stack
(15)           sw   x10,8(sp)
(16)
(17)           mv   x16,x0        # clear register for non-zero counter
(18)
(19) loop:     beq   x15,x0,done    # branch if count is zero
(20)           lbu  x20,0(x10)     # get unsigned byte from memory
(21)           beq  x20,x0,admin   # skip count if zero
(22)           addi x16,x16,1      # increment non-zero count
(23)
(24) admin:    addi  x10,x10,1     # increment address
(25)           addi x15,x15,-1    # decrement loop count
(26)           j    loop          # branch if count non-zero
(27)
(28) done:     mv   x15,x16       # copy count to returned reg
(29)
(30) restore:  lw   x16,0(sp)      # pop data into registers
(31)           lw   x20,4(sp)
(32)           lw   x10,8(sp)
(33)           addi sp,sp,12      # unadjust the stack pointer
(34)
(35) end:      ret                # transfer program control back
(36) #-----

```

Figure 12.20: A solution for this example.

Figure 12.21 shows a flowchart modeling this solution. This flowchart is fairly low level, but the text in the boxes could have been expanded to be more descriptive. Then again, there is nothing too exciting about this algorithm.

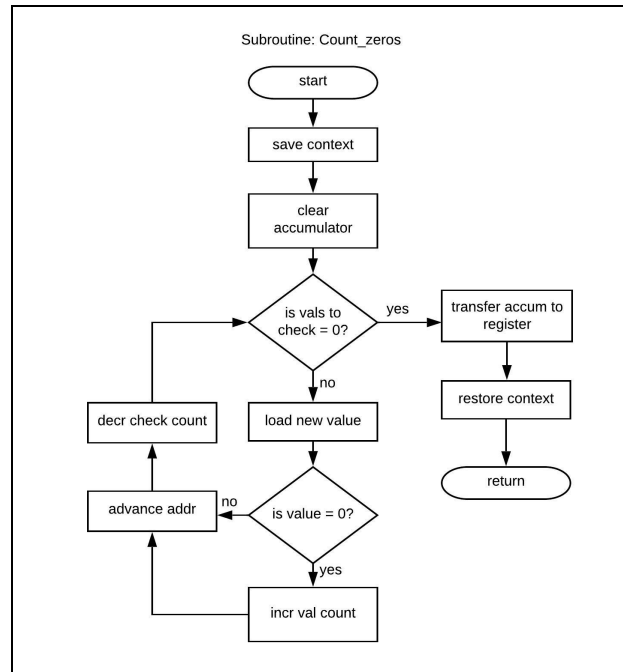


Figure 12.21: A flowchart modeling the operation of this example program.

Example 12.11: Memory Span Characteristic Checker

Write a RISC-V assembly language subroutine that does the following: verifies that a contiguous section of memory has data (halfwords) that comes in pairs. This is if the first piece of data equals the second piece of data for all the pairs in the span, then the subroutine returns a non-zero value in `x10`; otherwise, the subroutine returns a zero in `x10`. The subroutine checks 32 pieces of data to verify if they are in contiguous pairs or not. The starting address of the first piece of data in memory is passed to the subroutine in `x20`. Don't allow the subroutine to permanently change any registers other than `x10`.

Solution: Another subroutine that goes to main memory and tries to do something meaningful. The challenge in some of these problems is to understand the program statement; this is one of these problems. Here are the fun issues worth mentioning regarding the solution in Figure 12.22:

- Part of the initialization sequence is to store context by pushing three registers onto the stack. We do this last when we're writing the subroutine because we don't know in advance which registers we'll use in our solution.
- The other part of the initialization code is to set the loop counter to 16, which is half the stated count of 32; recall that we're checking pairs which is why we divide the original count by 32.
- We know the count is not zero, so we can use a do-while loop, which we start on line (18-19) by loading two halfwords of data. We then check to see if the two halfwords are equal or not. Life is good if they are equal, and we branch to checking the loop condition and other administrative tasks if they are equal. If the values are not equal, we don't take the branch and drop down to line (22), which loads `x10` with a zero indicating a mismatched pair. From there we jump to restoring context and exiting the subroutine. If all is good, the loop eventually fails, which

means all the halfwords are in equivalent pairs. In this case, we load x10 with a non-zero value before restoring context.

- The subroutine restores context on line (31-34). We ended up using three registers.
- We did somewhat of a trick here. We used x10 as the loop counter even though we knew it was acting as a flag variable. We did the so that we did not need to use another register, which would had required us to push it and later pop it as part of context saving/restoring. In the end, taking this approach saves us two instructions and allowed the subroutine to always run using two less instructions. Pretty dang clever. This is a typical trick we always seek out and use in assembly language programming.

```

(00) #-----
(01) # Subroutine: Chk_mem_pairs
(02) #
(03) # This subroutine verifies the 32 halfwords of data in contiguous memory
(04) # starting at the address in x20 arrives in pairs with equivalent values.
(05) # If they do, x10 is assigned a non-zero value; otherwise x10=0.
(06) #
(07) # Passed values: x20
(08) # Tweaked Registers: x10
(09) #-----
(10) Chk_mem_pairs:
(11) init:      addi  sp,sp,-12      # make room on stack for storage
(12)           sw   x20,0(sp)      # push altered registers on stack
(13)           sw   x21,4(sp)     # push 2 items on stack
(14)           sw   x22,8(sp)
(15)
(16)           li   x10,16        # load half total count into register
(17)
(18) loop:     lhu   x21,0(x20)    # get first halfword
(19)           lhu   x22,2(x20)  # get second halfword
(20)
(21)           beq  x21,x22,admin  # jump if OK
(22)           mv   x10,x0        # bad... clear flag, exit subroutine
(23)           j    restore      # jump to restore context
(24)
(25) admin:    addi  x20,x20,2     # advance address by halfword
(26)           addi  x10,x10,-1   # decrement loop count
(27)           bne  x16,x0,loop   # loop
(28)
(29) all_good: li   x10,1        # all good: put non-zero value in x10
(30)
(31) restore:  lw    x20,0(sp)    # pop data into registers
(32)           lw    x21,4(sp)
(33)           lw    x22,8(sp)
(34)           addi  sp,sp,12     # unadjust the stack pointer
(35)
(36) end:      ret                # transfer program control back
(37) #-----

```

Figure 12.22: A solution for this example.

Figure 12.23 shows yet another flowchart supporting yet another example problem. The details may amaze you if you blink your eyes at just the right speed.

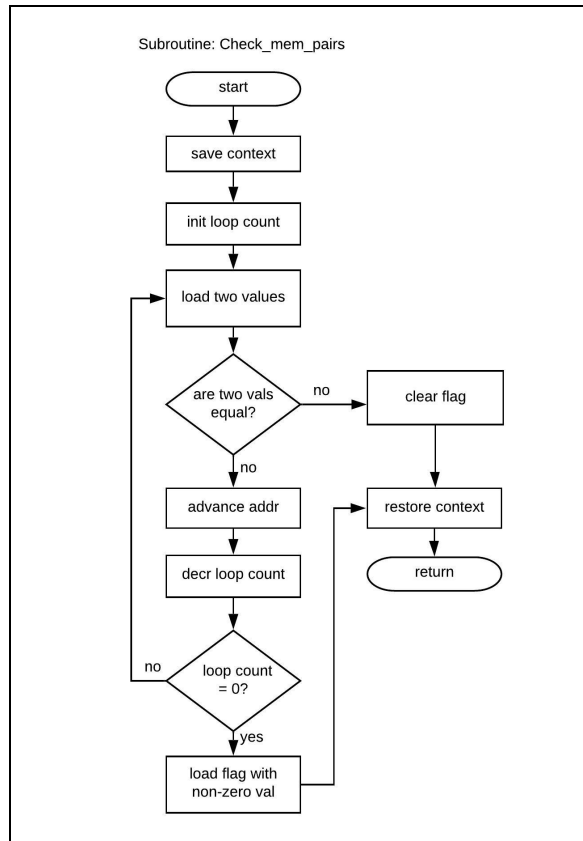


Figure 12.23: A flowchart modeling the operation of this example program.

12.7 Chapter Summary

- Stacks are abstract data types typically used by computer architectures for special types of storage and special program flow control mechanisms such as subroutine implementation.
 - The stack abstract data type is known as a LIFO, which in terms of data management, stands for “last in, first out”. The basic operations of a stack include *pushing* and *poping*. Basic stack definitions include *top of stack* and *stack pointer*.
 - The RISC-V MCU instruction set does not include dedicated push and pop instructions; it instead implements push operations with store-type instructions and pop operations load-type instructions. Both push and pop implementations required two instructions: one for the actual memory access, and a second instruction to adjust the stack pointer according to the push or pop operation.
 - Using subroutines to structure your assembly language programs has many advantages such as increased readability, understandability, maintainability, code reuse, program space efficiency. The use of subroutines typically decrease runtime efficiency.
 - Subroutines appearing in programs should be well-delineated from other source code, include banners describing the subroutine name, description, and registers the subroutine permanently changes.
 - There are three main methods to “pass” value to and from subroutines: 1) registers, and, 2) memory. No matter how you do it, you should document it in the subroutine header.
 - Well-written subroutines typically save context upon entry to the subroutine, and restore context upon exiting the subroutine. In other words, the subroutine stores the values of the registers that the subroutines changes and restores those values once before exiting the subroutine. Saving context means to push the registers the subroutine tweaks onto the stack; restoring context means that to pop the values off the stack back into the original registers before exiting the subroutine.
 - Subroutine calls in the RISC-V architecture automatically use one of the registers in the register file (x1) to store the subroutine “return address”. In this context, the return address is the address of the instruction following the call instruction. For nested subroutines, the value in **ra** needs to be saved before the nested subroutine is called; the best way to save **ra** is to push it on the stack before the nested subroutine call and pop it off the stack back into **ra** after the nested subroutine returns. The same mechanism holds true for recursive subroutine calls, which are subroutines that call themselves.
 - In addition to protecting the **ra** with nested subroutine calls, subroutines calls and returns must be done in the proper order to ensure the integrity of the stack. If a subroutine call does not have an associated subroutine return, or if a subroutine return is not paired with a subroutine call, the stack can either overflow or underflow. In either case, the program dies a slow death if not immediate death because the stack either overwrites important data, or it may provide random data as a return address from the subroutine.
 - All subroutines have overhead associated with them. At the very minimum, this includes the calling of the subroutine and returning from the subroutine, both of which are program flow control actions that effectively don’t do anything useful. Additionally, nested subroutines much expend instruction saving and restoring **ra**. Finally, subroutines generally save and restore context. Saving and restoring context and the **ra** don’t do anything in the big scheme of things. In general, subroutines make programs more space efficient but less runtime efficient.
 - Any assembly language program that uses a stack should place the stack pointer at a known at the beginning of the program. Initializing the stack pointer should thus be part of the programs initialization code.
 - There are specific rules you should follow to make subroutines as useful and meaningful as possible. These are simple rules that every good programmer inherently knows.
-

12.8 Chapter Exercises

- 1) Briefly describe what is meant by the term *abstract data type*?
- 2) Briefly describe why abstract datatypes definitions don't include low-level implementation details.
- 3) Briefly describe the term *last in first out* in terms of data storage in the RISC-V MCU.
- 4) Briefly describe the meaning of the terms *push* and *pop*.
- 5) Briefly describe the relation between the *top of the stack* and the *stack pointer*.
- 6) Briefly describe why the RISC-V MCU requires at least two instructions to perform push and pop operations.
- 7) Briefly describe if you could use **lh** and **sh** instructions as part of pushing and popping. If this is possible, describe the constraints involved.
- 8) What is the minimum number of instruction required to push ten word values onto the stack. Fully describe and explain your answer.
- 9) What is the minimum number of instruction required to pop ten values onto the stack. Fully describe and explain your answer.
- 10) Briefly describe why subroutine banners are a great idea.
- 11) Briefly describe the three main items that should appear in subroutine banners.
- 12) Describe the two ways you can pass data to and from subroutines in the RISC-V MCU.
- 13) Briefly describe the main difference between nested and non-nested subroutine calls in terms of the underlying RISC-V hardware.
- 14) Briefly describe why **call** pseudoinstructions are converted into two rather than one base instruction?
- 15) Describe a situation where a "lesser" amount of code in a subroutine requires more execution time than a "greater" amount of code in a similar subroutine. Assume these subroutines perform identical tasks.
- 16) Briefly describe the maximum depth you can nest subroutines with the RISC-V MCU.
- 17) Briefly describe what recursion means in the context of computer programming.
- 18) Briefly describe what is meant by the "depth of recursion".
- 19) Briefly describe the maximum depth of recursion possible on the RISC-V.
- 20) Briefly describe why subroutines are considered to be more code space efficient but less run-time efficient than not using subroutines.
- 21) Briefly describe the three types of overhead typically associated with subroutine calls.
- 22) Briefly describe why initializing the stack is not required, but is considered a really good idea in any assembly language program.
- 23) Briefly discuss if "the stack" is initialized at the beginning of programs written in higher-level languages.
- 24) Briefly describe the advantages of not have a dedicated "stack pointer" for any given computer architecture.
- 25) What is the maximum number of stacks a RISC-V assembly language program could easily control.
- 26) List a few drawbacks associated with writing long subroutines.
- 27) List a few reasons why for a given subroutine that you may not want/need to save and restore context.
- 28) What would be the major drawback of writing an assembly language program that uses many short subroutines?

- 29) Briefly describe if saving the context directly before and restoring the context directly after a nested subroutine call is a good idea or not.
 - 30) Briefly describe why it is a good idea to provide a subroutine banner for all the subroutines in your program.
 - 31) Briefly describe why it is a good idea to include a list registers modified by a subroutine in a subroutine banner.
 - 32) Briefly explain why the names of subroutines are nothing more than simple labels.
 - 33) List the basic cause of stack overflow.
 - 34) Briefly explain the fact that stack overflow can happen in two different directions.
 - 35) Briefly describe the three common ways misusing subroutines can cause stack overflow.
 - 36) Briefly describe whether it is possible to overflow the stack with subroutine calls and not have your program fail miserably.
 - 37) Briefly describe why it is bad to branch or jump to a subroutine rather than calling the subroutine.
 - 38) Briefly describe why it is bad to branch or jump out of a subroutine rather than returning from a subroutine using a **jalr** instruction or **ret** pseudoinstruction.
 - 39) I decided I did not need to use subroutine calls and returns; I decided to branch or jump to the subroutine and branch or return back from it. Briefly describe what's wrong with this notion.
-

12.9 Chapter Programming Problems

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code
 - Make sure each subroutine has a proper banner
- 1) Write a RISC-V assembly language subroutine that rewrites the data in x10 to be a horizontal nibble-level mirror image of itself. Don't permanently change any registers other than x10.
 - 2) Repeat the previous problem such that it reflects on a bit level rather than a nibble level.
 - 3) Write a RISC-V assembly language subroutine that determines if the data in x20 is a valid stoneage unary value. If it is, the subroutine returns a non-zero value in x30; otherwise it clears x30. Don't permanently change any registers other than x30.
 - 4) Write a RISC-V assembly language subroutine that determines if the data in x20 represents eight valid BCD values. If it does, the subroutine returns a non-zero value in x10; otherwise it clears x10. Don't permanently change any registers other than x10.
 - 5) Repeat the previous problem but also return the number of valid BCD values in the word in register x31.
 - 6) Write a RISC-V assembly language that converts a halfword representing a 4-digit decimal number (meaning the half word represents four BCD values) into a binary number. The value to convert is passed to the subroutine in x10 and also returned to the calling routine in the same register. Don't permanently change any registers other than x10. Assume all BCD values are valid.
 - 7) Repeat the previous problem, but return 0 in x10 if any BCD value is not valid.
 - 8) Write a RISC-V assembly language subroutine that packs all the set bits in register x20 to the right-most positions in the register. Don't permanently change any registers other than x20.
 - 9) Write a RISC-V assembly language subroutine that writes a monotonically increasing value to a span of unsigned halfwords in memory. The addresses of the first piece of data is passed to the subroutine in register x20; the number of pieces of data to inspect is a byte in register x21; the starting value of the count is passed to the subroutine in x22. Don't worry that the counter may overflow. Don't allow the subroutine to permanently change any register values.
 - 10) Write a RISC-V assembly language subroutine that looks at a span of contiguous signed words in memory, multiplies every negative value it finds by -1. The addresses of the first piece of data is passed to the subroutine in register x20; the number of pieces of data to inspect is a byte in register x21. Don't allow the subroutine to permanently change any register values.
 - 11) Repeat the previous problem, but return the number of values the subroutine changes in x30.
 - 12) Write a RISC-V assembly language subroutine that looks at a span of contiguous unsigned words in memory, and determines if the values in that span are always increasing in value. If the values are always increasing, load a non-zero value to register x31; otherwise load zero to x31. The addresses of the first piece of data is passed to the subroutine in register x15; the number of pieces of data to inspect is a byte in register x16. Return zero in x31 if the number of values to check is less than two. Don't allow the subroutine to permanently change any register values.
 - 13) Write a RISC-V assembly language subroutine that looks at a span of contiguous unsigned words in memory, finds the largest word in that span, and clears that word. The addresses of the first piece of data is passed to the subroutine in register x10; the number of pieces of data to inspect is a byte in register x15. If

the largest is repeated, only clear the first large value encountered. Don't allow the subroutine to permanently change any register values.

- 14) Write two RISC-V assembly language subroutines: Push_31 & Pop_31. These two subroutines use x31 as a stack pointer and allow for the pushing and popping of data to those two subroutines. These two subroutines pass data to and from the subroutines using x30. Do not allow the subroutine to permanently change any other registers other than x31.
 - 15) Write a RISC-V assembly language subroutine that copies the unsigned halfword data from one span of memory into another span. The addresses of the memory spans are passed to the subroutine in x10 & x11, where the data at the x10 address is the data to be copied. The number of data to copy is passed to the subroutine in x15. Don't allow the subroutine to permanently change any registers.
 - 16) Write a RISC-V assembly language subroutine that looks at two spans of contiguous unsigned words in memory. If a value at one location is zero, the subroutine makes the values at both memory locations zero. The addresses of the first piece of data in the memory spans is passed to the subroutine in register x20 and x21; the subroutine compares 32 pieces of data. Assume the spans in memory do not overlap. Don't allow the subroutine to permanently change any registers.
 - 17) Write a RISC-V assembly language subroutine that determines if a given span of signed word values in memory are always increasing. If it is, the subroutine returns a non-zero value in x31; otherwise it returns a zero in x31. The addresses of the first piece of data is passed to the subroutine in register x20; the number of pieces of data to inspect is a byte in register x26. Don't allow the subroutine to permanently change any register values.
 - 18) Repeat the previous problem, but ensure the values read from memory are monotonically increasing.
 - 19) Write two RISC-V assembly language subroutines that look at a span of contiguous signed words in memory, changes each word into a reverse image (LSB becomes MSB, etc.). The addresses of the first piece of data is passed to the subroutine in register x30; the number of pieces of data to reverse is in x31. You must use a nested subroutine for this problem. Don't allow the subroutine to permanently change any registers.
-

13 RISC-V MCU Interrupt Architecture (Firmware)

13.1 Introduction

We often base the viability of any MCU-based system on the ability of the system to respond to stimulus from the external world. In order to support these “response-time” issues, MCUs typically have the ability to utilize “interrupts”. In this context, interrupts are essentially a method to allow external hardware (external peripherals) to alter instruction-based program flow control. We consider the notion of interrupts as a type of I/O, but it’s slightly different from I/O as we know it from memory-mapped I/O.

The use of interrupts forms the basis real-time programming as it provides a mechanism for programmers to reduce response time and write programs that operate more efficiently. The approach we take in this text is to introduce the basic concepts of interrupts in the context of the RISC-V OTTER. The issue we need to work around is the fact that the current implementation of the RISC-V OTTER only contains one interrupt input. While this one interrupt is sufficient to introduce the basic concepts involved, one interrupt is not enough to introduce concepts with MCU-based systems that have many interrupts. We leave such concepts to more advanced digital design/embedded systems textbooks.

Main Chapter Topics

- **THE RISC-V INTERRUPT ARCHITECTURE:** This chapter describes interrupt architecture on the RISC-V, which it the hardware and software characteristics to implement real-time programming on the RISC-V.
- **THE SO-CALLED INTERRUPTS:** This chapter describes the basic the basic theory on interrupts on the RISC-V MCU from a programmer’s standpoint. Interrupts are a mechanism that allows hardware to effectively “call” subroutines.
- **REAL-TIME PROGRAMMING:** This chapter describes some of the theory behind real-time programming in the context of MCUs, using several programming examples.

Why This Chapter is Important

This chapter is important because it describes the RISC-V interrupt architecture from the standpoint of an assembly language programmer.

13.2 Interrupt Overview

The concept of interrupts is relatively simple. Essentially, an interrupt is a subroutine call that some device external to the MCU initiates. Recall that a normal subroutine call happens as a result of issuing a program flow control instruction such as `jal`, `jalr`, or `call`; these instructions are necessarily under program control. The execution of the “subroutine” associated with interrupts is not under program control, meaning we can’t issue an instruction that directly causes an interrupt.

The notion of generating an interrupt causes specific actions to happen in the underlying hardware. Because we’re discussing the programming side of the MCU, we save the details of interrupt processing on the hardware level to Chapter 18 in this text. This current chapter primarily describes interrupts and general and the programmer’s responsibilities to using interrupts on the MCU.

There are generally three types of interrupts, which we briefly describe below. For better or worse, the RISC-V MCU currently only has the capability of handling one external interrupt. Although this could be somewhat limiting, the operational characteristics of the RISC-V's interrupt reflects how other MCUs deal with interrupts. We label the notion of how a particular MCU handles an interrupt as that MCU's "interrupt architecture"¹.

- 1) External Interrupts: Some device external to the MCU generates this type of interrupt. We generally refer to these devices as peripherals and include such things as analog-to-digital converters, digital-to-analog converter, real-time clock (RTC) modules, and many other communication-type devices. The thing that makes these devices external is that they physically connect to an interrupt pin on the MCU (as opposed to connecting internally), which is a special pin in that it has the ability to generate interrupts in the MCU itself.
- 2) Internal Interrupts: Some device internal to the MCU generates this type of interrupts. We also refer to these devices peripherals and include the same devices as listed above. In other words, some MCUs contain these peripherals as part of the MCU itself in that these devices live on the interior of the IC. The RISC-V OTTER MCU does not currently have internal peripherals² but most MCUs do. Once you start adding internal peripheral devices, you're necessarily dealing with a microcontroller as opposed to a microprocessor, as microprocessors are primarily CPUs with extremely limited memory and/or I/O capabilities.
- 3) Software-based Interrupts: We typically use these types of interrupts for debug functions and/or to handle "special" conditions that may appear on the MCU and require special handling³. We don't generally see software-based interrupts often as the other two types of interrupts. This text does not discuss software-based interrupts.

13.3 The Theory of Interrupts

If you're like most humans, you societal norms occasionally cause you to think that you need a haircut. It would be a strange world if the person who cuts your hair called you every five minutes and asked you if your hair needs cutting. Naturally, a better approach (more efficient? Less annoying?) would be that when you needed a haircut, you simply call the person who cuts your hair and schedule an appointment. Requesting some type of service is the general approach humans take in most facets of their lives (unless you work in the sales where you're required to continually ask others if they want service). Phone solicitors therefore are not human.

Not surprisingly, an analogous situation exists in computerland. Programs you write generally do something, *i.e.*, they execute some finite number of relatively useful tasks to solve some problem. Additionally, programs are waiting for some indication that they need to do something; this indication is often times input from the outside world. The clearest example of this is your phone. When you don't interact with the display for a given amount of time, the device is smart enough to turn off the display as a power-saving measure. Yet, when you touch the display, it turns back on. More likely than not, the touching of the display told the device to wake up; the device was probably not actively checking to see if you touched the display. The thing to note here is that actively checking to see if someone touched the display is a waste of clock cycles if the MCU could be performing more important tasks or saving power by doing nothing if there was truly nothing to do.

The two approaches to knowing when a task should "take action" in embedded systems (such as a system controlled by a RISC-V MCU) are analogous to the example above: you either constantly check to see if a particular task needs attention from the MCU and act if it does, or you can give those tasks attention only when the tasks tell you they require attention. The notion here is that the task only seeks attention (meaning the execution of instructions) when they actually need attention. Microcontroller lingo refers to the act of constantly asking if a task needs attention as *polling*. MCUs implement polling by placing the program into a "polling loop", which is also appropriately referred to as a "dumb loop"⁴.

Polling is a relatively simple concept but it has one large drawback: it's inefficient to continually ask a device if it needs something when the device has nothing that needs doing. In terms of MCU processing, if the MCU is

¹ If you haven't figured out by now, the word "architecture" gets a lot of use in computerland.

² But there is nothing stopping you from adding them if you're working with an FPGA...

³ We often label these types of situations as "exceptions" and/or "traps".

⁴ No offense meant here to academic administrators.

polling, it is not doing something else that could be potentially more important (as in something time critical such as restarting some dude's heart). The result is that you lower the overall throughput of your system if you're wasting clock cycles in a polling loop. Once again, a more efficient approach in terms of MCU processing is to allow individual circuit elements that occasionally need attention from the MCU to have those circuit elements directly request processing, or "service" from the MCU. The notion of a hardware interrupt on MCUs provides a mechanism for such a request; the "interrupt architecture" on an MCU is simply a description of that mechanism.

We must be fair here and note that it's comfortable to say polling is bad, but in reality, it's only bad if the processor has something more important to do. In real life, your MCU may be idle sometimes when nothing needs doing; during those times, you can consider polling acceptable. The only possible problem here is that your program can be "stuck" in a polling loop and never be aware that peripherals in the circuit need the MCU's attention. Thus, there are gray areas in this discussion. But if you're processor is idle most of the time, you may want to choose a "less powerful" MCU or certainly an MCU with a low-power mode.

The term *interrupt* comes from the fact the normal operation of the microcontroller is temporarily *interrupted* to handle some other task. Once microcontroller handles the other task, the microcontroller returns to the task it was executing when it received the interrupt. Though microcontrollers in general use three types of interrupts (internal interrupts, external interrupts, and software interrupts), the RISC-V OTTER MCU currently only handles a single external interrupt. Keep in mind that there is no single method used by all microcontrollers to handle interrupts, so examining the interrupt architecture is one of the first things you typically do when working with a new microcontroller. We refer to "handling" these tasks only when the task requests service as *interrupt driven*, or "real-time", and thus require the use of the MCU's *interrupts*.

13.3.1 Using Polling for Inputting Data

Most of the RISC-V MCU programs we've written thus far used some form of polling. In this context, the "something useful" statement refers to the notion that most programs interface with the outside world in one way or another, which requires them to input data from that outside world. The MCU typically reacts to that input and then outputs something to the outside world. When the MCU requires something (such as a specific condition) from an external device, one approach to obtain that information is to constantly ask the device if it's ready to provide that information, which is the classic definition of polling. An example of such a system would be an MCU that receives input from an external sensor at a set frequency.

Figure 13.1 shows an example of this basic program procedure that uses polling. The program needs to do something when a certain switch is turned on; because the program does not know when the switch will turn on, it must constantly monitor the switch, which it does in Figure 13.1 using a polling loop. The polling loop is on lines (12-14); the program inputs data, masks that data to isolate the switch in question, and then reacts to the state of the switch. If the switch is not on, the program turns off all LEDs and then continues in the polling loop by branching back to the input instruction on line (12). If the switch is on, then the condition associated with the branch instruction evaluates as false and the program does not branch, and instead exits the loop and drops down in the code to do other things, which in this example is turn all the LEDs on.

Although this program works great and everything seems fine, there is a problem. The polling loop in Figure 13.1 is not problematic according to our definition because the program has no other tasks it needs to attend to. Another way of saying this (the official embedded systems way of saying it) is that there are no other pending tasks that require the attention of the MCU. The problem arises when there are other tasks. In this case, the other tasks may need attention also, but they won't receive it as long as the MCU is stuck in a polling loop such as the one on lines (12-14). More than likely, the switch does not require as much attention as this program is giving it, which can be an issue in an actual problem.

Keep in mind we designed these examples and definitions to be simple. In real life embedded systems applications, these situations can become exponentially complicated as the number of tasks that the program needs to monitor increases. In this context, the number of tasks refers to the number of items (such as inputs and outputs) that require the MCUs attention. The notion of interrupts is important because any meaningful embedded system (an embedded system with many tasks) probably would not work properly, if at all, if it relied solely in polling. The solution is to utilize real-time programming, particularly by taking advantage of the MCU's interrupt architecture.


```

(00) #-----
(01) # This program reads data from the switches; if the second to right-most
(02) # switch is on (on=1), then the program turns on all LEDs; otherwise
(03) # the program turns off all LEDs. The port address of the switches is
(04) # 0x1100C000; the port address of the LEDs is 0x11008000. Assume
(05) # there are 16 switches and an equivalent number of LEDs.
(06) #-----
(07) init:    li    x10,0x1100C000    # put switch address (input) to register
(08)         li    x11,0x11008000    # put LED address (output) in register
(09)         li    x8,0xFFFF        # load reg with one output value
(10)         mv    x9,x0            # load reg with other output value
(11)
(12) main:    lhu   x20,0(x10)        # input data
(13)         andi  x20,x20,2        # mask 2nd to right-most bit
(14)         beq  x20,x0,out_off    # if not zero, branch to off
(15)
(16) out_on:  sh   x8,0(x11)        # turn on all LEDs
(17)         j    main            # do it again
(18)
(19) out_off: sh   x9,0(x11)        # turn off all LEDs
(20)         j    main            # do it again

```

Figure 13.1: The solution to this example problem.

13.3.2 Moving Towards Real-Time Programming

Interrupts are an extremely important part of any computer system. Thus, understanding the interrupt architecture is vital to writing good programs that drive efficient systems. The notion of interrupts becomes more important with working with embedded systems and particularly at the assembly language level. In order to successfully work with interrupts, you must understand the low-level details of the interrupt architecture associated with the computer you're working with. Being that there is no one method used by all MCUs to handle interrupts, you'll soon discover that one of the first things you must do when working with a new MCU is to examine the interrupt architecture. First, you look at the architecture, then you look at the instruction set, then the I/O architecture, and finally, you look at the *interrupt architecture*. You'll need to establish the flavor and number of interrupts the microcontroller handles and how exactly the MCU handles the interrupts, since the use of polling rather than interrupts is an indication of a nooby programmer.

The term *interrupt* comes from the fact the normal operation of the microcontroller is briefly *interrupted* to take care of some other special task (by special, we inherently mean more important). Once the MCU *handles* the task, the MCU returns to the processing it was doing when the interrupt arrived. The basic model is that some peripheral device can *request service* from the MCU. We do this by allowing the external device to directly connect to MCU by way of a dedicated signal. We refer to this input on the RISC-V MCU as the interrupt input.

Figure 13.2 shows the top-level diagram for the RISC-V MCU; the input with the “INTR” label is the dedicated interrupt input. Because the design of the overall system including how the hardware is set up is not a programming concept, we'll leave those details for the hardware section of this text. What we'll say now is that when MCU hardware detects the signal connected to the INTR input at a '1' state, the RISC-V MCU executes a special subroutine. We usually refer to this special subroutine as the “interrupt service routine”, or “ISR”, but other people refer to it as the “interrupt handler”. We'll deal with some of these specifics in this chapter when we discuss the required real-time programming details.

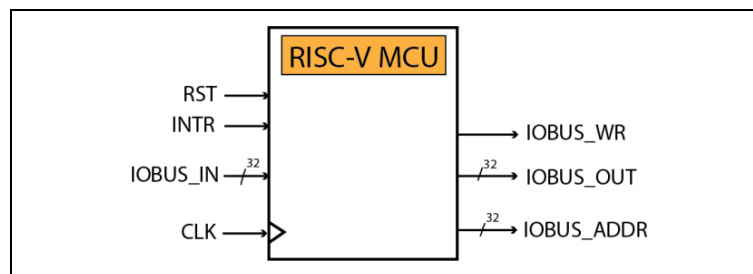


Figure 13.2: The RISC-V MCU schematic symbol.

13.3.2.1 The Advantage of Real-Time Programming

There are essentially an infinite number of approaches you can take using an MCU to solve a problem. In this context, solving the problem using a MCU requires two separate approaches. First, someone needs to design the hardware for the system; we consider the MCU to be an important part of that hardware. Second, we need someone to write the firmware for the system. There are many options and trade-offs in designing both hardware and firmware. Additionally, there is no “set of rules” that exist such if you follow the rules, you magically have a well-designed and well-functioning system.

The first step in any MCU-based design is to know the system requirements before you start. We tend to try to design stuff to run fast so we can impress out friends, but that’s not always the most important design issue. Recall that possibly the biggest design issue faced by modern embedded system designers and programmers is not only making your system work, but make it work efficiently. In this context, the notion of efficiently allows the program to be run at low power, thus making battery powered applications happy. The point of this paragraph is that you’ll find in all meaningful applications that, designers and programmers use real-time programming to meet their goals⁵.

In the general case, writing real-time programs has two basic advantages over systems that rely exclusively on polling. What this means is that most of the time, using interrupts in your design is going to make your design a better. Here are the two major advantages, or maybe “potential” advantages of implementing a real-time design.

- 1) **Increases System Throughput:** In this context, we define the term *throughput* as the amount of meaningful things an MCU does over a given amount of time that it’s active⁶. The problem with a polling loop is that although the processor is executing instructions at high rate, we can view the instructions as not really doing anything until the condition the loop is polling for materializes. In other words, polling represents a relatively high percentage of useless instructions. The throughput is low in this situation because the MCU is executing instructions, but it is doing no meaningful work. Once again, if there are no other tasks that need the MCU’s attention, you can argue that this approach is OK. In general, querying an I/O device that probably does not provide useful information most all of the time, lowers the overall through put of the MCU.
- 2) **Response Time:** The notion of the interrupt is that the code that the MCU is processing is “interrupted” so that the MCU can execute a special subroutine. This means that the code in the special subroutine executes with what we consider a higher priority than the code not in that special subroutine. Having code run at different priorities such as this is a way to reduce the “response time” of your system. If your system relies 100% on polling, the response time of your system and/or the system complexity increases exponentially as you add more tasks that your MCU needs to handle.

Imagine a complex digital system that contains many I/O devices. In such a system, polling each of these devices would usually be a bad option because it may take a long time for a given device to get the service it needs. The better option would be for all the devices to request service from the microcontroller only when they need it. Real-time systems become quite interesting as the systems become more complex as such issues of interrupt priority, interrupt latency, specialized interrupt hardware, and other real-time concepts become more important. Don’t worry, most of these concepts are beyond the scope of this text. We’ll cover the more important issues later in this text.

13.4 RISC-V Interrupt Architecture for Programmers

The overall notion of interrupts is relatively simple due to their similarity with subroutines. Stated as simply as possible, an interrupt is basically a subroutine call that is initiated by the hardware. In contrast, executing a `call` instruction initiates a subroutine in a program. We describe the mechanism that the hardware uses to initiate the special subroutine in a later chapter. For now, all the hardware the pure programmer needs to know is that the

⁵ Low power hardware and subsequent firmware design are indescribably important. This text leaves those issues for another course. You’ll find that off-the-shelf MCUs have many ways to adapt to your particular design such that you can lower the overall power and processing needs of your design.

⁶ The notion here is that if the processor truly has nothing to do, it can turn itself off and wait for a signal to turn itself back on. This is a classic low-power mode that most off-the-shelf MCUs have. In other words, the MCU does not always need to be running if there is no hope that it will need to do anything for a while.

hardware designers of the system that contains the MCU they are programming set up the system such that an external device can cause the hardware to initiate the execution of the special subroutine. In short, when the RISC-V MCU hardware detects a request for service from an external device, the underlying hardware initiates a sequence of events to switch to the execution of the special subroutine.

13.4.1 Real-Time Programmer Responsibilities

Although the interrupt architecture on the hardware level is somewhat complex, the pure programmer only has a few responsibilities when writing interrupt-driven programs. We list these responsibilities below with a brief description, then delve into them deeper in later subsections.

- 1) The Overall Program Structure: Real-time programs have a special structure that is different from non-real-time programming. You'll learn in later discussions that there are things our programs need to do and special places in the code where they do those things. We divide the code into three sections: 1) initialization (both of interrupts and the program in general), 2) the background task, and, 3) the interrupt service routine.
- 2) Interrupt Initialization: There is special hardware in the RISC-V MCU that is dedicated to interrupt implementation. This hardware requires programmers to initialize it in various ways in order to make the program work properly. This initialization is generally part of the overall program initialization.
- 3) The Interrupt Service Routine: The special subroutine that we previously mentioned as a specific name: the *interrupt service routine*, or (ISR). It's truly a subroutine, but there are several approaches to using the ISR in an optimal manner.

13.4.1.1 Real-Time Program Structure

Figure 13.3 shows a basic interrupt-driven assembly language program. This code actually does something if you can imagine that the MCU can control a single LED. This code does in fact contain the three items listed in the previous section. Some of these items are not apparent, so we provide a few pertinent comments below. Note that the code below depends on the fact that some external device has a pin that connected to the RISC-V MCU's interrupt input, and is occasionally generating interrupts.

- The initialization code spans lines (09-19), and comprises of interrupt-related and general initializations. Line (09) stores the output port address. Line (14) initializes a register to use as a flag. Lines (15-16) put the output LED into a known state. The code on lines (11-12) and lines (18-19) are part of the interrupt initialization code that we'll discuss later.
- The main code, or what we'll often refer to as the *background task*, is on lines (21-22). This code is always running, waiting for an interrupt to happen. This code is a polling loop, but that's OK for this example as the code only has one task to perform, which is blinking an LED. We use the name main code and background task interchangeably; we sometimes refer to the code as the *task code*.
- The interrupt service routine is on lines (39-40), after we introduce it with a nice descriptive banner. The ISR represents the foreground task. Program execution exits the background task each time the MCU acts on an interrupt and commences executing the foreground task.

```

(00) #-----
(01) # Example Interrupt Driven Program.
(02) #
(03) # Description: The program blinks an LED. Each time the program
(04) # receives an interrupt the code changes the state of the LED. We
(05) # assume some external device has configured the hardware such that
(06) # the MCU can receive an interrupt signal from an external device.
(07) #-----
(08) My_Prog:
(09) init:   li    x15,0x1100C004 # put output address into register
(10)
(11)        la    x6,ISR          # load address of ISR into x6
(12)        csrrw x0,mtvec,x6    # store address as interrupt vector CSR[mtvec]
(13)
(14)        mv    x8,x0          # clear x8; use as flag
(15)        mv    x20,x0         # keep track of current output value
(16)        sw    x20,0(x15)     # put LEDs in known state
(17)
(18)        li    x10,1          # set value in x10
(19)        csrrw x0,mie,x10    # enable interrupts
(20)
(21) main:   nop                 # do nothing (easier to see in simulator)
(22)        beq   x8,x0,main     # wait for interrupt
(23)
(24)        xori  x20,x20,1      # toggle current LED value
(25)        sw    x20,0(x15)     # output LED value
(26)
(27)        mv    x8,x0          # clear flag
(28)        csrrw x0,mie,x10    # enable interrupt
(29)        j     loop           # return to loopville
(30) #-----
(31)
(32) #-----
(33) # The ISR:
(34) #
(35) # Description: This ISR puts a non-zero value into x8.
(36) #
(37) # Tweaked Registers: x8
(38) #-----
(39) ISR:   li    x8,1           # set flag to non-zero
(40)        mret                   # return from interrupt
(41) #-----

```

Figure 13.3: An example interrupt-driven program.

13.4.1.2 Interrupt Initialization

Interrupt driven programs requires two forms of initialization programs, which are items programmers must be aware of: the vector address and the interrupt enable. Both of these items are values that the RISC-V MCU hardware stores in special register, thus there are instructions in the RISC-V ISA that access these registers.

Recalling that the ISR is a subroutine “called” by hardware. Acting on an interrupt causes the transfer of program control to the ISR. In a subroutine call, the assembler encodes the information to know where to jump to (the address of the first instruction in the subroutine) as part of the instruction. Because ISRs don’t have a program-related calling mechanism, the address of the ISR must be stored as part of the interrupt initialization code. Typical MCU vernacular refers to the mechanism as the *vector address*, with the idea that program execution “vectors” to that address when the MCU acts on an interrupt.

The vector address is stored in the **mtvec** register, which is one of three registers involved in interrupt processing. The vector address is stored with a **csrrw** instruction. Table 13.1 gives details of the **csrrw** instruction. Here are the important items to know about this instruction.

- The instruction mnemonic states for “control and state register read write”. There are three CSR registers that programmers can write to; one of them is the **mtvec** register. This instruction allows you to simultaneously read the current contents of CSR[mtvec] and store that value in a register,

and write a new value to CSR[**mtvec**]. The CSR[**mtvec**] register is one that we typically only write to once in a given program.

- The RISC-V MCU stores the interrupt vector address in CSR[**mtvec**]. To do this, programmers need to first issue a **la** (load address) instruction to obtain the value of the ISR label (it's an address), then use the **csrrw** instruction to save that address in CSR[**mtvec**]. Thus, CSR[**mtvec**] contains the address of the first instruction in the ISR. When the MCU acts on an ISR, the underlying hardware ensures that the instruction at this address is the next one executed.
- We typically don't need to know what the CSR[**mtvec**] value is, so we use **x0** as the destination register in the **csrrw** instruction.
- The **mtvec** value in the instruction assumes the assembler knows a value for **mtvec** to use for **mtvec**. The **mtvec** value is actually an address of a particular register in the hardware, so **csrrw** is simply a number.
- **csrrw** is a base instruction. Its underlying bit format is unique so we don't consider it as having an instruction "type".

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
none	csrrw rd,csr,rs1	rd ← CSR[csr] CSR[csr] ← rs1	csrrw x0,mtvec,x8	Simultaneous read and write of the mtvec CSR register

Table 13.1: The csrrw instruction with other information.

The other form of initialization that programmers must do for all interrupt driven programs is to "enable interrupts". Programmers can enable or disable interrupt under program control using the **csrrw** instruction by writing to the CSR[**mie**] register. Note that the "ie" in mie stands for "interrupt enable", which is a comment acronym in MCU-related lingo. This register is only one-bit wide. Writing a '1' to this register enables the interrupts; writing a '0' to this register disables the interrupts. When the interrupts are disabled, the MCU effectively ignores any pending interrupts; when interrupts are enables and an external device connected to the RISC-V MCU generates an interrupt, the MCU processes that interrupt, which include calling the ISR. Table 13.2 shows instruction usage to disable/enable interrupts by writing CSR[**mie**].

Instruction Usage	Comment
<pre> mv x5,x0 # clear x5 csrrw x0,mie,x5 # write CSR[mie] </pre>	Disable interrupts (prevent interrupt processing)
<pre> li x5,1 # set LSB in x5 csrrw x0,mie,x5 # write CSR[mie] </pre>	Enable interrupts (allow interrupt processing)

Table 13.2: The csrrw instruction usage for enabling/disabling interrupts.

When working with interrupts, MCUs typically use a special vernacular to indicate whether interrupts are enabled or not. If interrupts are disabled, we can say that they are *masked*. Conversely, if interrupts are *unmasked*, we know that interrupts are enabled. The act of masking the interrupt means that we are disabling it, which is a term we use quite often. In addition, if we unmask an interrupt, we are enabling it. We sometimes refer to bits such as CSR[**mie**] as the *interrupt mask bit*.

Unlike working with the vector address where we generally on need to write it once during the initialization part of our program, we typically need to write CSR[**mie**] more often. The reason is that when the RISC-V MCU acts on an interrupt, part of the interrupt architecture dictates that the hardware automatically disables future interrupts. The hardware disables the interrupts so that the MCU can execute instructions without risking

receiving another interrupt and having the MCU act on it. If the hardware did not automatically disable the interrupts, the first instruction in the ISR would cause another interrupt to be processed, even if that instruction attempted to disable the interrupts. xxxxAAs we'll discuss later, although an ISR is similar to a subroutine, it's different because the RISC-V OTTER hardware does not currently have the capability to *nest interrupts*.

Automatically disabling the disabling interrupts has two ramifications to the program. First, the program has time do whatever processing required in the ISR (or associated with the interrupt) without risking acting on another interrupt. Second, the programmer must re-enable the interrupts under program control using the `csrrw` instruction outlined in Table 13.2. Where exactly to place the instruction can be tricky. Don't try to put the code at the end of the ISR because if there is a pending interrupt, the MCU will act on the interrupt before the program can exit the ISR, which would represent the deadly "nested interrupt".

Good programmers always know the state of the interrupts relative to the code they're writing. You always must ensure the interrupts are masked if you're program is executing important code. The most important code for us now is the initialization code. We hope that hardware designer provided a way to ensure that our interrupts powered-up in the disabled state, but we generally don't take chances. In all embedded systems programming, it's better to do what you can as a programmer to ensure the integrity of your system. In this case, that means probably the first instructions in any program you write should be to mask the interrupts.

13.4.1.3 The Interrupt Service Routine

Implementing Interrupt service routines have the same guidelines are implementing subroutines. The only major difference is that they have use different instructions to transfer program control (the return statement). We'll discuss returning from ISRs in another section. Similar to subroutines, ISRs should save the operating context of the MCU when it received the interrupt.

There is one other obvious difference between ISRs and subroutines. The code in the ISR necessarily runs with a higher priority than the code in any subroutine, which is because the external event (the interrupt) causes the program from to switch from whatever it may be doing to the ISR code. When writing ISRs, you should keep in this in mind. There is one important ISR guideline here as a result of the higher running priority of the ISR code. When you're executing ISR code, the interrupts are disabled, which means the program may be missing some important event while processing the interrupt. You can't get around this issue by simply re-enabling the interrupts in the ISR, which would cause the interrupts to nest, and your program to die. The general approach solution here is to strive to keep you interrupts are short as possible. Notice we say, "you should" as a general approach, but this is not always desirable and/or feasible.

13.4.1.4 Saving the Context

Various MCUs out there have many different context saving mechanisms, which is yet another reason why examining the interrupt architecture is always one of the first things you do when working with a new MCU. The notion of saving the context is important because by definition, when we act on an interrupt, we're temporarily suspending the part of code we're currently executing and then start executing the ISR. This implies that we may be using registers in the background task, so we want to ensure that the ISR does not permanently change those registers. What you ideally want the MCU to do is stop the code that it is currently executing, execute the ISR code to completion, and then go back to the code that the MCU was executing when the MCU received the interrupt. The idea here is that if you must can "save the state" of the MCU before you execute the ISR, then you can "restore" that state once the ISR completes execution and before you start executing the code you were executing when MCU received the interrupt.

Many MCUs store the context automatically in hardware, but the RISC-V MCU has no such mechanism. All context saving in the RISC-V MCU is done in firmware and uses the same approach as saving the context when you call a subroutine. Recall that we saved context by pushing the registers used in that subroutine onto the stack at the beginning of the subroutine and popping them off the stack before the subroutine terminates. Recall that ISRs are essentially subroutines that the hardware can "call", so it's no surprise subroutines and ISRs share the same characteristics. To summarize, the RISC-V MCU has no automatic context saving mechanism; context saving in ISRs is done by the programmer in firmware in the same way the program saves context in subroutine calls.

One could argue that the RISC-V does have some type of automatic context saving mechanism. Usually, when we speak of context, we primarily refer to register values (from the register file). We could easily stretch this definition to include other registers, such as the program counter. In truth, part of the automatic context saving mechanism in the RISC-V MCU is to save the address of the instruction following the instruction that was executing when the MCU received the interrupt to a CSR register, **mepc**. Because the hardware automatically does this, we won't delve deeply into the subject until the hardware portion of this text.

13.4.1.5 Returning From ISRs

As you probably would guess, returning from ISRs is similar to returning from subroutines. When a subroutine is called (when a program executes a **call** instruction), the RISC-V hardware automatically saves the return address (the address of the instruction after the call instruction) in a register (typically **x1**, or **ra**). When the subroutine exits (the MCU executes a **ret** instruction), it loads the value in that register into the PC, which makes it the next instruction executed after the **ret** instruction.

Returning from an interrupt is similar, except that it loads the return address from another CSR register rather than from the **ra** register. When the MCU acts on an interrupt, the MCU's hardware places the address of the instruction after the instruction that was being executed when it received the interrupt into CSR[**mepc**], which is one of the three registers in the CSR. The hardware automatically controls the loading of this hardware so the programmer does not need to do anything. Additionally, the programmer would rarely have a reason to ever load a value into CSR[**mepc**], though they could do so with the **csrrw** instruction.

Because the hardware uses a different return address when returning from a subroutine, programmers must use a different instruction. The instruction in this case is **mret**. Table 13.3 shows various helpful information associated the **mret** instruction. Similar to **ret**, **mret** has no operands. Additionally, **mret** is a base instruction.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
none	mret	$PC \leftarrow CSR[mepc]$	mret	Return instruction for returning form ISR

Table 13.3: The **csrrw** instruction with other information.

13.4.2 Basic Interrupt Example Program

We initially presented an interrupt driven example problem but provided very little information describing the operation of the program. We'll once again show this program, but this time explain it in a painful amount of detail. The disclaimer for this section is this: this is a simple example that shows the basic structure/requirements of an interrupt driven RISC-V interrupt driven program. This is not necessarily a good example, because a "good" interrupt driven program example would be more complex, which would allow programmers to do things "more intelligently". The point is that there are many approaches to writing real-time programs; if you learn the basics associated with simple programs such as the one in this section, you'll have no trouble writing your own "good" interrupt driven programs. Figure 13.4 shows the same program we previously provided; here's all the good stuff to realize about this program:

- The program assumes that some external device attached to the RISC-V MCU generates interrupts. Each time the MCU receives an interrupt, the code changes the state of an external peripheral, which in this case is an LED. Someone has generously provided you the programmer with the correct output port address associated with the LED.
- The first part of the program is initialization, as indicated with "init" label. The first instruction places the output port address into a register for later use.
- The instructions on lines (11-12) is initialization of the interrupts. The instruction on line (11) loads the address of the subroutine into a register; this value is the address of the first instruction of the ISR, which is the interrupt vector address. The following instruction stores that address value in CSR[**mtvec**]; the underlying hardware loads this value into the PC when the MCU acts on an interrupt.

- We use a register as a “flag”, which is the assembly language approach of using a Boolean value. The program places either a ‘1’ or ‘0’ into the flag register; the program interprets these two values as positive logic where ‘1’ means something happened and ‘0’ means otherwise. The code on line (14) uses x8 as the flag register and sets the flag to an initial value of zero.
- The program blinks an LED; we use a register to hold the current LED value. We initialize that register to zero (LED initially off) on line (15). We follow that line with an instruction to write that register value to the output on line (16).
- The second portion of the interrupt initialization code resides on lines (18-19). At this point in the code, we’ve completed all the other required initializations; we purposely saved this code until last. The purpose of this code is to enable (unmask) the interrupts, which we do by writing a ‘1’ to the CSR[mie] register using the **csrrw** instruction on line (19).
- The background task is the loop on lines (21-22). The program gets stuck in this code for what seems like forever, because this code keeps monitoring (yes, this is a polling loop) the state of x8, which we are using as a flag. We initialized this flag to zero, and we keep loop so long as it remains zero. The only way x8 can become a non-zero value is when the MCU receives an interrupt and executes the interrupt service routine.
- When the program receives an interrupt, the program transfers control to the ISR, which starts on line (39). Be sure to note the nice banner for the subroutine, very similar to standard subroutine banners. The ISR comprises of two instructions: line (39) change the value of the flag to be non-zero, and line (40) returns from the subroutine. Note that we use an **mret** instruction rather than a **ret** instruction because we are returning from an interrupt and not a normal subroutine.
- The program was executing the instruction on line (21) or line (22) when it received the interrupt. Interrupts are external to the MCU and can thus happen at any time. When the ISR exits, it returns to one of these instructions. The difference now is that x8 is no longer zero, which causes the conditional branch on line (22) to fail and program control to drop through to the instruction on line (24).
- The program has one task to do as part of the main code: toggle the LED. It does this by first toggling a bit in the register storing the LED value on line (24), then outputting that value to the LED on line (25). This code effectively makes the LED blink.
- Once the LED blinking completes, we need to prepare for the next interrupt. We first clear the flag register (x8), which allows us to stay in the polling loop on lines (21-22). We must use this approach based on the way we structured our code. We then need to unmask the interrupts, which we do by write a ‘0’ to the CSR[mie] register on lines (27-28).
- The final part of the code is to jump back to the main loop, which we do on line (29).


```

(00) #-----
(01) # Example Interrupt Driven Program.
(02) #
(03) # Description: The program blinks an LED. Each time the program
(04) # receives an interrupt the code changes the state of the LED. We
(05) # assume some external device has configured the hardware such that
(06) # the MCU can receive an interrupt signal from an external device.
(07) #-----
(08) My_Prog:
(09) init:    li    x15,0x1100C000 # put output address into register
(10)
(11)         la    x6,ISR          # load address of ISR into x6
(12)         csrrw x0,mvec,x6     # store address as interrupt vector CSR[mvec]
(13)
(14)         mv    x8,x0          # clear x8; use as flag
(15)         mv    x20,x0         # keep track of current output value
(16)         sw    x20,0(x15)     # put LEDs in known state
(17)
(18)         li    x10,1          # set value in x10
(19)         csrrw x0,mie,x10    # enable interrupts
(20)
(21) main:    nop                # do nothing (easier to see in simulator)
(22)         beq   x8,x0,main     # wait for interrupt
(23)
(24)         xori   x20,x20,1     # toggle current LED value
(25)         sw    x20,0(x15)     # output LED value
(26)
(27) admin:   mv    x8,x0         # clear flag
(28)         csrrw x0,mie,x10    # enable interrupt
(29)         j     main          # return to loopville
(30) #-----
(31)
(32) #-----
(33) # The ISR:
(34) #
(35) # Description: This ISR puts a non-zero value into x8.
(36) #
(37) # Tweaked Registers: x8
(38) #-----
(39) ISR:     li    x8,1          # set flag to non-zero
(40)         mret   # return from interrupt
(41) #-----

```

Figure 13.4: An example interrupt-driven program.

13.4.3 Real-Time Programming Considerations

Real-time programming is an art form. There are so many issues involved with even relatively simple real-time problem that any rules as associated with designing and programming such systems become questionable. If there were a set of rules to follow to ensure that any real-time program you wrote was going to solve the given problem 100% of the time, then people would not be paying you the big bucks to be embedded systems designers and programmers. There are a few guidelines you should consider following, particularly if you're new at real-time programming. Here they are:

Keep your ISR as short as possible: The issue here is that we want to keep the response time as short as possible. The problem is that the hardware automatically masks the interrupts the MCU acts on an interrupt, and can only be unmasked under program control. This generally means that if your ISRs are long, that may cause you to delay or completely miss another interrupt. They delayed interrupt may cause an obnoxious delay that would make people think less likely to purchase your product (or hire you); missing the interrupt altogether could never good outcome.

Nested ISRs: The current interrupt architecture does not allow for nested interrupts. Because there is currently only one register to store the return address (CSR[mepc]) when the MCU acts on an interrupt, your code would return to the incorrect place in the code if you acted on a second interrupt. Note that the only way you could get a nested interrupt is if you re-enable interrupts while you are in an ISR. This being the case, nested interrupts are easily avoidable.

Calling subroutines from ISRs: There is nothing inherently wrong with calling subroutines from interrupts. Your code can even nest subroutine calls if there is a need if you follow the standard rules for writing and nesting subroutines. In truth, the only two special things about your interrupt code the fact that 1) the ISR uses a different return instruction from subroutines, and, 2) the interrupts are probably disabled while executing the ISR code. The one possible drawback of calling subroutines in the ISR code is the fact that it extends the amount of time interrupts are disabled, which may cause problems with response time issues with acting on other pending interrupts.

13.5 Real-Time Programming Example Problems

This section provides a few interrupt-driven example problems. These problems are similar to the example presented earlier in this chapter, but do show a few more tricks and expose a few more issues.

Example 13.1: Another Blinking LED

Write a RISC-V assembly language program that counts the number of interrupts the MCU receives. The count range is [0,255] and rolls over from 255 to 0. When the count is less than 128, the program turns on the right-most LED; otherwise it leaves it off. The LED address is 0x1100C000.

Solution: The first thing to note about the solution is that it is very similar to the first interrupt driven program we worked with. That being the case, we essentially copied much of that code. As you'll see, the interrupt initialization is always the same; the program initialization, not so much so. Here are the other cool things to note about this solution.

- The program places the LED address into a register for later use by output instructions on line (09).
- The program writes the address of the ISR to the CSR[mtvec] register on lines (11-12); this is where program flow vectors to when the MCU acts on an interrupt.
- This program uses x8 as a flag, so we clear it on line (14). We use this to have the ISR signal that the MCU received an interrupt.
- We also clear our LED counter value and write that value to the outputs on lines (15-16). We always want to put external items in a known state as part of the initialization sequence.
- We enable (unmask) the interrupts on lines (18-19) by using the csrwr instruction to write to the CSR[mie] register.
- The main code (background task) starts on line (21) with a polling loop. The loop is checking the value if the flag register x8; the code stays in this loop until x8 contains a non-zero value.
- When the MCU receives an interrupt, program control transfers to the foreground task, which is the first instruction in the ISR on line (42); this instruction places a non-zero value into x8 before returning program flow control to the background code.
- The first thing we do outside of the polling loop is to increment the counter on line (23). We then massage the counter by clearing all but the lower byte, which we do because the problem states that our count range is [0,255].
- At this point, we could do some type of compare operation, but we instead take advantage of the fact that if the 8th bit from the left is set, then the count is greater than 127, and we thus want to turn off the LED. We first mask the count on line (25) to isolate the eighth bit, then shift it right to the LSB position on line (26). This bit does not have the correct logic level, so we toggle it on line (27) before outputting it to the LEDs on line (28).
- Once we complete handling the stuff the problem description wanted, we need to recover from the interrupt and prepare to receive another interrupt, which we do starting at the line with the admin

label. We first clear the flag register `x8` on line (30), which was made non-zero in the ISR. We then use a `csrrw` instruction on line (31) to set `CSR[mie]`, which unmask the interrupts. Recall that when we receive an interrupt, the RISC-V hardware automatically masks the interrupts.

- We are now ready to receive another interrupt, so we transfer program control back to the main loop on line (32).

```

(00) #-----
(01) # Example Interrupt Driven Program.
(02) #
(03) # Description: The program counts the number of interrupts using the
(04) # count range of [0,255]. When the count is less than 128, the program
(05) # turns on LED. Assume some external device configured the hardware so
(06) # the MCU can receive an interrupt signal from an external device.
(07) #-----
(08) My_program:
(09) init:    li      x15,0x1100C000 # put output address into register
(10)
(11)         la      x6,ISR          # load address of ISR into x6
(12)         csrrw  x0,mtvec,x6     # store address as interrupt vector CSR[mtvec]
(13)
(14)         mv      x8,x0          # clear x8; use as flag
(15)         mv      x20,x0         # keep track of current count
(16)         sw      x20,0(x15)     # put LED in known state
(17)
(18)         li      x10,1          # set value in x10
(19)         csrrw  x0,mie,x10     # enable interrupts
(20)
(21) main:    beq     x8,x0,main     # wait for interrupt
(22)
(23)         addi   x20,x20,1        # increment counter
(24)         andi   x20,x20,0xFF    # clear all but lower byte
(25)         andi   x21,x20,0x80    # mask the 2^7 bit (8th from right)
(26)         srli   x21,x21,7       # shift to LSB position
(27)         xori   x21,x21,1       # toggle LSB to agree with problem
(28)         sw      x21,0(x15)     # output LED value
(29)
(30) admin:   mv      x8,x0          # clear flag
(31)         csrrw  x0,mie,x10     # enable interrupt
(32)         j      loop           # return to loopville
(33) #-----
(34)
(35) #-----
(36) # The ISR:
(37) #
(38) # Description: This ISR puts a non-zero value into x8.
(39) #
(40) # Tweaked Registers: x8
(41) #-----
(42) ISR:     li      x8,1          # set flag to non-zero
(43)         mret                    # return from interrupt
(44) #-----

```

Figure 13.5: An example interrupt-driven program.

Example 13.2

Modify the code in example problem solution shown in Figure 13.4 such that the ISR handles all the LED blinking activity.

Solution: The previous example used a register as a flag, which allowed the foreground task (the ISR) to signal to the background task (the main code) that an interrupt had occurred. There was nothing special about this approach other than to show that we often use registers (and sometime memory locations) as flags. The previous solution actually made the code slightly more complicated. Here are some items to note about the solution to this example in Figure 13.6.

- The first thing to note is that the code is shorter than the previous solution, so this solution is more space efficient.
- The code is first initializing all the registers it uses on lines (09-10), then writes the interrupt vector on line (12-13), then puts the LEDs in a known state (off) on line (15-16).
- The main code consists of unmasking the interrupts. The issue here is that we continually unmask the interrupts, but this is the only way we can do this in a simple problem such as this one. The tendency is to unmask the interrupts before leaving the ISR, but that's a horrible idea because a pending interrupt will cause the interrupts to nest.
- This solution does all the blinking work in the ISR, which comprises of toggling the state of the LED using an XOR instruction on line (29), and then outputting the result on line (30).
- Overall, this program is functionally equivalent to the previous solution; the only notable difference is that this solution does more in the ISR, which means the interrupts are disabled for longer compared to the previous solution.

```

(00) #-----
(01) # Example Interrupt Driven Program.
(02) #
(03) # Description: The program blinks an LED. Each time the program
(04) # receives an interrupt the code changes the state of the LED. We
(05) # assume some external device has configured the hardware such that
(06) # the MCU can receive an interrupt signal from an external device.
(07) #-----
(08) My_Prog:
(09)   init:   li      x15,0x1100C004 # put output address into register
(10)         li      x10,1         # set value in x10
(11)
(12)         la      x6,ISR        # load address of ISR into x6
(13)         csrrw  x0,mtvec,x6   # store address as interrupt vector CSR[mtvec]
(14)
(15)         mv      x20,x0        # keep track of current output value
(16)         sw      x20,0(x15)    # put LEDs in known state
(17)
(18)   main:   csrrw  x0,mie,x10   # enable interrupt
(19)         j      main         # return to main loop
(20) #-----
(21)
(22) #-----
(23) # The ISR:
(24) #
(25) # Description: This ISR toggle the LSB of x20 and outputs it
(26) #
(27) # Tweaked Registers: x20
(28) #-----
(29)   ISR:   xori   x20,x20,1     # toggle current LED value (LSB)
(30)         sw      x20,0(x15)    # output LED value
(31)         mret                    # return from interrupt
(32) #-----

```

Figure 13.6: An example interrupt-driven program.

Example 13.3

Write a RISC-V MCU interrupt-driven assembly language program that blinks a single LED. The LED is in the LSB position of the output port with the address 0x1100C004. The LED toggles each time the system receives an interrupt; assume the hardware is configured such that an external peripheral can generate an interrupt on the RISC-V MCU. The blinking action only occurs if the switch in the LSB position is on; otherwise, the LED turns off and does not blink. The port address of the switch input is 0x11008000. Keep the ISR as short as possible.

Solution: The example seems similar to the previous example, but this example has an extra control input that partially determines the how the LED operates. Because much of the code in this example is similar to the previous example, we'll only describe the main differences. Figure 13.7 shows the solution to this example along with this other fun stuff to note:

- There are many possible solutions to this example; Figure 13.7 show just one of them, and not necessarily the best solution, but certainly a working solution.
- The main difference with this solution is in the structure of the code. The previous example unconditionally blinked the LED when the program received an interrupt. This program now blinks the interrupt conditionally based on the value of a switch. Additionally, the program must ensure the LED is off if the switch is not actuated. These differences make the program structure quite different.
- There is a polling loops starting on line (25). The interrupts were never masked, so the this polling loop checks to see if the switch is on. If the switch is on, the code drops out of the polling loop; otherwise the program continues to poll the switch.
- With the switch activated, the code exits the polling loop and first enables the interrupts on line (29). The program checks the status of the flag on line (30), and toggles the LED if the flag is set starting with the code on line (32). Note that if program needs toggle the LED, it also needs to unmask the interrupt, which the program does on line (35).
- If the interrupt-received flag is not set, the program checks for to see if the switch is still on lines (38-40). This code is a repeat of the polling loop on line (25), but the code on lines (38-40) has an if/else structure. If the switch is off, we jump to line (20) to turn off the LED and disable the interrupts. If the switch is on, the program branches to check the status of the interrupt-received flag on line (30).

```

(00) #-----
(01) # Example Interrupt Driven Program LED Blinking Program.
(02) #
(03) # Description: The program blinks an LED only when a given switch is off
(04) # The LED address port is 0x1100C004; the switch input port address is
(05) # 0x11008000. If the right-most switch is one, the program toggles the LED
(06) # each time it receives an interrupt. Assume the hardware is configured
(07) # such that some external device that the MCU can receive an interrupt
(08) # signal from device
(09) #-----
(10) My_Prog:
(11) init:  li    x15,0x1100C004 # LED port address (output)
(12)        li    x16,0x11008000 # switch port address (input)
(13)
(14)        la    x6,ISR        # load address of ISR into x6
(15)        csrrw x0,mtvec,x6   # store address as interrupt vector CSR[mtvec]
(16)
(17)        mv    x8,x0        # clear x8; use as flag
(18)        li    x10,1        # set value in x10
(19)
(20) sw_off: mv    x20,x0       # clear LED
(21)        csrrw x0,mie,x0    # disable interrupts
(22)        sw    x20,0(x15)   # put LEDs in known state
(23)
(24) sw_off_loop:
(25) main:  lw    x25,0(x16)    # input switch data
(26)        andi  x25,x25,1    # mask lsb
(27)        beq  x25,x25,main  # branch if switch off
(28)
(29)        csrrw x0,mie,x10   # enable interrupts
(30) sw_on:  beq  x8,x0,chk_sw  # check for flag
(31)
(32) togl:  xori  x20,x20,1    # toggle current LED value
(33)        sw    x20,0(x15)   # output LED value
(34)        mv    x8,x0        # clear flag
(35)        csrrw x0,mie,x10   # enable interrupts
(36)
(37) sw_on_loop:
(38) chk_sw: lw    x25,0(x16)    # input switch data
(39)        andi  x25,x25,1    # mask lsb
(40)        beq  x25,x25,sw_off # branch if switch off
(41)
(42)        j    sw_on         # return to loopville
(43) #-----
(44)
(45) #-----
(46) # The ISR:
(47) #
(48) # Description: This ISR places a non-zero value into x8.
(49) #
(50) # Tweaked Registers: x8
(51) #-----
(52) ISR:   li    x8,1         # set flag to non-zero
(53)        mret                # return from interrupt
(54) #-----

```

Figure 13.7: An example interrupt-driven program.

Example 13.4

Write a RISC-V MCU interrupt-driven assembly language program that blinks a single LED. The LED is in the LSB position of the output port with the address 0x1100C004. The LED toggles each time the system receives an interrupt; assume the hardware is configured such that an external peripheral can generate an interrupt on the RISC-V MCU. The LED can blink at two different frequencies based on the state of the switch in the LSB position. If the switch is on, then the LED toggles every two received interrupt; otherwise the LED toggles on every received interrupt. The port address of the switch in put is 0x11008000. Keep the ISR as short as possible.

Solution: The example is somewhat similar to the previous examples, but with some slight twists. Once again, we'll only describe the significant differences, particularly the structural differences in the program, because there are many similarities with previous solutions. Figure 13.8 shows the solution to this example: here is the description of fun stuff contained within:

- Lines (11-12) include initialization instructions for both the interrupts and other standard items.
- The code next falls into a one-line polling loop on line (25). When the program breaks out of this polling loop, it then increments a counter. Using a counter is a standard way of track on/off event. In this program we're sometimes interested when things happen, and other times, when things happen every other time. When something happens and the LSB of the counter is '1', we know something has happened every other time. In this program, we always toggle the LED if the switch is on; otherwise we toggle the LED if the switch is off and the LSB of the counter is '1'. Somewhat tricky, but hey, it's assembly language.
- After the program increments the interrupt counter on line (26), we input the switch data to determine the LED blink frequency, which we do on lines (28-30). Note that its structure is an if/else construct. The if part of the construct jumps over the code that toggles the LED to the code at the "done" label. The if code essentially jumps over the else code. The else code is the code that toggles the LED and resides on lines (35-36). The else code is on lines (32-33).
- The code at the "done" label does tasks that always need to be done including resetting the flag and unmasking the interrupt on line (38-39). The program then transfer control back to the loop that polls for the interrupt flag on line (25).

```

(00) #-----
(01) # Example Interrupt Driven Program LED Blinking Program.
(02) #
(03) # Description: The program blinks an LED only when a given switch is off
(04) # The LED address port is 0x1100C004; the switch input port address is
(05) # 0x11008000. If the right-most switch is one, the program toggles the LED
(06) # each time it receives an interrupt. Assume the hardware is configured
(07) # such that some external device that the MCU can receive an interrupt
(08) # signal from device
(09) #-----
(10) My_Prog:
(11)   init:   li     x15,0x1100C004 # LED port address (output)
(12)         li     x16,0x11008000 # switch port address (input)
(13)
(14)         la     x6,ISR          # load address of ISR into x6
(15)         csrrw  x0,mtvec,x6    # store as interrupt vector CSR[mtvec]
(16)
(17)         mv     x8,x0          # clear x8; use as flag
(18)         mv     x9,x0          # use as interrupt counter
(19)         li     x10,1         # set value in x10
(20)
(21)   sw_off: mv     x20,x0        # clear LED - 344 -ource- 344 -y
(22)         sw     x20,0(x15)    # put LEDs in known state
(23)         csrrw  x0,mie,x10    # unmask interrupts
(24)
(25)   main:   beq     x8,x0,main   # branch if switch off
(26)         addi   x9,x9,1        # increment interrupt counter
(27)
(28)   get_sw: lw     x15,0(x16)    # get switch data
(29)         andi   x15,x15,1      # mask LSB of counter
(30)         beq   x15,x0,togl     #
(31)
(32)   slow:   andi   x15,x9,1      # mask LSB
(33)         beq   x15,x15,done    # branch to done if zero
(34)
(35)   togl:   xori   x20,x20,1     # toggle current LED value
(36)         sw     x20,0(x15)    # output LED value
(37)
(38)   done:   mv     x8,x0        # clear flag
(39)         csrrw  x0,mie,x10    # enable interrupts
(40)         j      main         # return to loopville
(41) #-----
(42)
(43) #-----
(44) # ISR Description: This ISR places a non-zero value into x8.
(45) #
(46) # Tweaked Registers: x8
(47) #-----
(48)   ISR:   li     x8,1          # set flag to non-zero
(49)         mret                    # return from interrupt
(50) #-----

```

Figure 13.8: An example interrupt-driven program.

13.6 Chapter Summary

- Interrupts are a signal from the world outside of the MCU connected to a dedicated pin on the MCU.
 - Interrupts provide a method for external hardware to execute a special subroutine typically referred to as the interrupt service routine (ISR). Much of the functionality associated with interrupts is the responsibility of the underlying hardware. Interrupt driven programs form the basis of embedded systems programming.
 - The interrupt architecture is a term we use to describe all the hardware and hardware-induced operations associated with the processing interrupts. The interrupt architecture is one of the first things you should examine when dealing with a new MCU or CPU, as interrupt driven programs have many distinct advantages over programs that are not interrupt driven.
 - There are three main types of interrupts: 1) internal, 2) external, and, 3) software-based. These types are based upon which device and the location of that device in the system.
 - If some device requires service, there are two ways to make this need known to the MCU: 1) polling, or 2) interrupt driven. Polling refers to the MCU expending instructions to see if a device requires service. Interrupt driven systems allow the particular device to tell the MCU when and if it requires service.
 - The main problem with polling is that it is done under program control, which means it takes time, and can effectively prevent your MCU from processing doing any other processing. Polling a device typically creates low throughput because the act of asking a device if it needs service wastes time in the case where the device does not require service. While polling in itself sounds bad, it is actually only bad if the act of polling prevents the MCU from performing a more meaningful task. Often time in MCU-based digital design, there are times when there is “nothing” that the MCU needs to do; these times are ideal for polling because polling is most often “nothing”.
 - Programmers have several responsibilities when writing interrupt-driven programs. First, they must write an Interrupt Service Routine (ISR), which is similar to a subroutine. Second, they must store the interrupt vector address (the address of the first instruction in the ISR) in the CSR[mtvec] register. Third, they must control the interrupt enable (CSR[mie]), which controls whether the MCU acts on interrupts or not. Interrupts must be enabled (unmasked) before the MCU can act on an interrupt, and must also be unmasked after the MCU receives an interrupt because the interrupt architecture automatically disables interrupts as part of initial interrupt processing.
 - Programmers should strive to keep ISRs as short a possible because we typically process ISRs with the interrupts masked. Any time the interrupts are masked, the system could experience a delay because the MCU is not able to react to an interrupt. Making and unmaking of interrupts is done under program control, though interrupts are masked in hardware after an interrupt is acted on by the MCU.
 - The current RISC-V architecture does not have the capability to nest interrupts, so programmers must be careful to not enable interrupts within the ISR if there is any chance of receiving another interrupt.
-

13.7 Chapter Exercises

- 1) List and briefly describe the three main types of interrupts.
 - 2) Briefly describe why does the current approach to the RISC-V MCU interrupt architecture somewhat limited?
 - 3) In your own words, describe the notion of polling.
 - 4) Briefly describe in your own words why is polling usually a bad idea.
 - 5) Briefly describe in your own words when polling is not a totally bad idea.
 - 6) Briefly describe what we mean by an *interrupt service routine*
 - 7) Briefly describe what is exactly is being interrupted in the context of interrupts.
 - 8) Briefly describe why it is important to always examine the interrupt architecture for each new MCU you work with.
 - 9) List and briefly describe the two main reasons to use a real-time system to solve your given problem.
 - 10) List and briefly describe the two forms of initialization require by an interrupt driven program.
 - 11) Masking interrupts sounds very much like bit masking. Briefly comment if there any meaningful relation.
 - 12) Briefly describe the two ramifications that automatically disabling interrupts in hardware has for programmers.
 - 13) Briefly describe what we mean by “foreground” and “background” tasks in the context of RISC-V MCU assembly language programs.
 - 14) Briefly explain why the code in an interrupt service routine is considered “higher priority code” compared to code that is not part of an interrupt service routine.
 - 15) Briefly describe why masking interrupts should be one of the first tasks in any assembly language program.
 - 16) Briefly describe the general use of a “flag” variable or register.
 - 17) Briefly describe the functional differences (not the RTL) between **mret** and **ret** type instructions.
 - 18) Briefly describe why it is not possible to nest interrupts on the RISC-V OTTER MCU.
 - 19) Briefly describe whether programmers can call subroutines from ISRs.
 - 20) Briefly describe whether programmers can call subroutines that call other subroutines from ISRs.
 - 21) Briefly describe how it would be possible to receive an interrupt while in the interrupt service routine knowing that the hardware automatically masks the interrupts upon entry to the ISR.
 - 22) Briefly describe whether it would possible to act on another interrupt while in the interrupt service routine.
 - 23) Briefly describe why it is a good idea to keep ISRs are short as possible.
 - 24) Briefly describe the ramifications of jumping out of an ISR rather than returning from it with an **mret** instruction.
 - 25) Briefly describe what would happen if you returned from an interrupt using a **ret** pseudoinstruction rather than a **mret** instruction.
 - 26) I designed my interrupt service routine such that it called one subroutine. Briefly explain whether I need to save the return address register (ra) before I call the subroutine in the ISR.
 - 27) Briefly describe whether it is possible to nest subroutine calls in an interrupt service routine.
-

13.8 Chapter Programming Problems

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code
 - Provide a banner for all subroutines
 - Keep ISRs as short as possible
 - Assume there are 16 switches (port address = 0x11008000) and 16 LEDs (port address = 0x1100C004).
- 1) Write a RISC-V OTTER interrupt driven assembly language program that does the following. Assume that some external hardware can assert a signal connected to the RISC-V MCU interrupt input. Each time the MCU receives an interrupt, the program inputs a value from the switches and outputs that value to the LEDs. After receiving ten interrupts, the program stops acting on interrupts until it detects that only the right-most button is pressed (active high), at which point, the program continues processing interrupts in the ten interrupt sequence. The MCU continues doing this process-wait pattern for an eternity.
 - 2) Write a RISC-V MCU interrupt-driven assembly language program that outputs a 16-bit binary count to port address 0x11003008. Each time the program receives an interrupt, the program outputs advances the count value then outputs it. If the switch in the LSB position is on (on=1), then the program adds two to the count; otherwise the program adds three to the count before outputting. For this problem, don't worry about overflow in the counter. Don't perform any I/O in the ISR.
 - 3) Write a RISC-V MCU interrupt-driven assembly language program that does the following each time it receives an interrupt. The interrupt indicates that the program must transfer data starting at the address in x20 to the output port address 0x11005500, one byte at a time. The number of bytes of data to output is given by the switch data, which forms a binary value that is never greater than 255. Don't Don't perform any I/O in the ISR.
 - 4) Write a RISC-V MCU interrupt-driven assembly language program that does the following: it keeps a decimal count of the number of interrupts. The count starts at zero; the 1's, 10's, and 100's digits are stored in registers, x10,x11,x12, respectively. Each time the count changes, the three values are output to port address 0x11009990, 0x11009991, and 0x11009992, respectively. The count should roll over from 9999 to 000. Don't Don't perform any I/O in the ISR.
 - 5) Repeat the previous problem with the following modifications. When the program will increment or decrement depending upon the value input from port address 0x110000F0; if the input value is zero, the count increments; otherwise it decrements. Don't allow the count value to exceed 999 or go below 000, meaning when it hits those values, the count does not increment for 999 and does not decrement for 000.
 - 6) Write a RISC-V MCU interrupt-driven assembly language program that does the following: each time it receives an interrupt, it reads a unsigned byte from port address 0x11002200. This value can be in the range [0,32], and is used to light the same number of LEDs in a stoneage unary type manner (light LEDs starting from right and filling to the left). The value is output to the LED port address of 0x1100C000. Don't Don't perform any I/O in the ISR. This is an example of a digital level meter.
-

14 Important Supporting Topics

14.1 Introduction

There are a few topics regarding the programming side of the RISC-V MCU that didn't fit into other chapters. These topics are important so we group them all into this chapter before we go onto other amazing stuff.

Main Chapter Topics

- **MEMORY SEGMENTATION:** This chapter provides a description of the RISC-V MCU's segmented memory model including an overview of the utilized segments.
- **RISC-V MCU ASSEMBLERS:** This chapter provides an overview of the currently available RISC-V assemblers and the basic functionality such as assembler directives.
- **PROGRAMMING EFFICIENCY ISSUES:** This chapter provides an overview of concepts and terminology dealing with programming efficiency in the context of standard programming constructs.

Why This Chapter is Important

This chapter is important because it describes many important support topics associated with programming the RISC-V MCU.

14.2 Memory Segmentation

The RISC-V MCU has one memory, which we commonly refer to as main memory. This memory provides storage for both the program and data. We further divide the data portion of memory into special areas for particular uses such as the stack. We typically refer to the various areas of memory by the notion of “segments”. Segmenting memory is a common term when working with MCUs, which is why the memory map associated with the MCU is so important. Figure 14.1 once again shows the memory map for the RISC-V MCU, which clearly shows the various segments in main memory.

Keep in mind that the notion of segmenting memory is an approach to help humans better understand and work with system resources. In the end, it's all just memory; but particular portions of that memory serve different purposes so we give those portions special names associated with the word “segment”. The astute programmer can change many but not necessarily all of the segment boundaries and addresses because they are most likely arbitrary. Figure 14.1 essentially represents a set of starting guidelines. Keep in mind that physical memory on the RISC-V Otter consists of bytes in the range [0x00000000,0x0000FFFF].

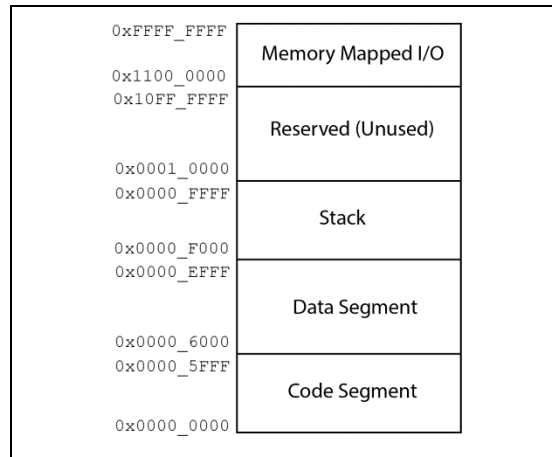


Figure 14.1: The RISC-V MCU memory map.

14.2.1 Memory Address Space

The term memory address space is a common term in computerland. As you can probably see by now, the notion of memory in a computer system is very important, as even simple computers base their operation on accessing different types of memory in the system. This being the case, the instructions in the computer’s instruction set have a heavy focus on “addressing” in order to efficiently work with that memory. The result of this is that we need to become familiar with exactly how much memory the computer can access.

The notion of accessing memory can be misleading. People who design generic computers must deal with a trade-off regarding the overall physical size of the computer and the basic functionality of the computer. This notion becomes obvious with the notion of memory accessing. Every computer has a maximum amount of memory it can directly address; the value is based on the width of the bundle that the system uses to address memory. We refer to the amount of memory a CPU can address as the memory address space, which is based on the physical size (data width) of the address lines. The trade-off computer designers must deal with is that the wider the larger the address space, the larger the computer is going to be. The problem is that some applications won’t need all that memory space. The best example of this is with the RISC-V Otter MCU. The address space is 32 bits, but the current OTTER implementation only uses 16-bits of that. While we could redesign the RISC-V hardware to limit the address space to 16 bits, this would require extra time and effort, and it may not be what we want somewhere down the line.

The point here is that the address lines in the RISC-V can sometime address physical memory and sometimes it addresses “other things” such as input and output ports. The physical memory in the RISC-V Otter MCU is [0x00000000,0x0000FFFF]. Note that the amount of memory is constant, but the address of the memory, or the placement of this memory in the memory map of Figure 14.1 is arbitrary; we placed it starting at address zero for convenience. In the same way, most of the memory map is arbitrary; but someone needs to map this stuff out. If you’re simply a programmer, someone needs to provide these details for you; but if you’re the system designer, you’ll need to provide these details for the people working on and/or programming your system.

14.2.2 Code Segment

As the name implies, the RISC-V MCU uses the code segment to store programs. That being the case, we often refer to the code segment part of memory as program memory. Typical MCU lingo uses different and often confusing names for the code segment, and the RISC-V MCU is no different. The RISC-V MCU often refers to the code segment as the “text segment” for some unknown reason.

When you write an assembly language program, the assembler translates your assembly code into machine code and stores it in the designated code segment portion of memory. Programmers should always specify that their code goes into the code segment using the “.text” assembler directive, which we’ll discuss in the next section. We’ve used this quite often in our programming examples up to now; it becomes more important when we discuss look-up-tables in another chapter. The assembler actually assumes your instructions go into the code

segment if your program does not specify a data segment (we cover data segments in the following section), but you should always use the `.text` assembler directive in order to provide clarity for humans reading your code.

14.2.3 Data Segment

The data segment is where the RISC-V MCU stores intermediate data. The notion here is that the RISC-V only has 32 general-purpose registers, which is simply not enough data storage for many applications. The solution is to store intermediate values in main memory to free up registers to use in general data crunching. Additionally, the designers of the RISC-V created instructions that easily and efficiently access data memory for look-up-tables.

We've previously dealt with this topic when discussing load and store instructions and various issues regarding the stack. Both the stack segment and the data segment represent area for generic data storage, but they differ by how they are accessed. In general, programmers access the data segment using load and store instructions, but do so in two different ways. The two ways differ in how the program provides the physical address. For generic data access, the physical address can be anything; for stack operations, program stores the main part (base address) of the address in a register we refer to as the stack pointer. Generic data memory accesses specify the memory address of interest using an address specified in the associated load and store instruction, while access to the stack uses a reference to the stack pointer to access data.

Not all programs need to specify a data segment. We generally only specify a data segment when we need to reserve specific places for memory, such as a look-up-table. Data accesses such as stack operations rely on the notion of stack pointer stored in a register. Other data operations access area of memory that fall into two categories: memory with pre-initialized values and memory that is simply *reserved* but not initialized. In cases where our programs need special access to memory (reserved memory) or pre-initialized memory (look-up-tables), we need to explicitly specify a data segment, which we do with a `“.data”` assembler directive, a topic we cover in the next section with a greater amount of detail.

14.2.4 Stack

The stack segment is another area of data memory. Because the access to the stack segment is conceptually different from access to the data segment, we consider the stack and code segment to be separate independent areas of main memory. We extensively described stack operations in a previous section, so we'll not describe it again here. There are a few items to keep in mind regarding the stack segment.

- There is no assembler directive specifying the stack as there was for the data segment.
- We don't need to use an assembler directive to specify memory dedicated to the stack; we just know the location of the top of the stack and know that the stack grows in the negative direction regarding memory addressing.
- There is no magic uncrossable boundary between any parts of main memory including the data and stack segment. It is once again up to the programmer to write code that respects these boundaries, as it would be easy for stack operations to corrupt the data segment and data access to corrupt the stack segment with stack overflow/underflow.

14.2.5 Memory Mapped I/O Segment

The memory mapped I/O segment is not actually a physical segment in the RISC-V MCU as were the code, data, and stack segments. We include the memory mapped I/O in the memory map because it provides useful information to programmers regarding I/O operations. What programmers need to know is that the underlying hardware considers a memory access instruction to be an I/O instruction based on the value of the memory address specified by the load or store instruction.

As the memory map in Figure 14.1 indicates, the hardware interprets all memory accesses with addresses above a certain value (as specified in Figure 14.1) as I/O. The underlying hardware handles all the required details, so programmers don't need to worry too much about this. Recall that the person who designed/configured the hardware must inform potential programmers of the “memory address” associated with I/O. The memory mapped I/O segment does not require any special assembler directives because memory mapped I/O is based in the associated hardware configuration.

14.3 The RISC-V Assemblers

As of this writing, there are three RISC-V assemblers that you can use to assemble your programs. They each have their pro & cons, which we list in Table 14.1. Note that all assemblers are available at no cost.

Assembler	Brief Description	Pros	Cons
Venus	Web-based	<ul style="list-style-type: none"> • very simple • includes graphical-based simulator/debugger • includes error message reporting 	<ul style="list-style-type: none"> • does not recognize interrupt-based instructions • does not simulate interrupts • fixed data segment address
RARS	Downloadable Java-based	<ul style="list-style-type: none"> • simple • includes graphical-based simulator/debugger • better error message reporting 	<ul style="list-style-type: none"> • fixed data segment address • does not simulate interrupts
gcc	gcc-based	<ul style="list-style-type: none"> • very complete • very versatile • various debuggers available 	<ul style="list-style-type: none"> • requires Unix-based environment • steeper learning curve

Table 14.1: Description of various RISC-V assemblers.

All three assemblers have the ability to act as simulators/debuggers. Not only can the assembler assemble your programs (convert your programs to machine code), they also allow you to debug/simulate your code. This means that you can step through your code one instruction at a time and watch the various RISC-V memory elements change including register, memory, and program counter. The act of stepping through your assembly code line-by-line in this manner and watch the changes occur in the various RISC-V MCU memory elements makes the software a simulator. When your code does something incorrectly, you can fix your code, thus making the software a debugger.

For the current form of this course, we'll be using both the Venus and RARS assemblers. The gcc assembler and associated development tools are by far the best tools available. The problem with gcc-based tools are that they require a Unix-based environment, which has a steeper learning curve based on the notion that most people taking this course have little experience working in a Unix-based environment. We intend to switch to the gcc assembler in future offerings of the course, which will happen once we make the switch to presenting this course using a complete Unix-based environment. Your particular instructor may have you work with gcc, but most instructors do not.

14.3.1 Assembler Directives

Recall that assembler directives are messages from the human programmer to the assembler. In general, the use of assembler directives in your program allows you some level of control as to how the assembler handles your program. The RISC-V assembler has many available directives in order to provide the programmer with more versatility in overall program design. We only cover the more commonly used directives in this section, as some of the directives are typically associated with advanced assembly language topics and large programs.

We can classify the directives as being one of two types: required by the program, or, 2) helpful to make your programs more readable to humans. Table 14.2 shows the list of directives we'll use in the program presented in this text; the following sections show usage information for these directives. As you'll see in the following code examples, all directives begin with a period and we place them in the left-most column of text in your source code.

Directive	Short Description	Comment
.text	Indicates following information is in text segment	Required if using .data directive
.data	Indicates following information is in data segment	Required if initializing or reserving data
.space	Allocates a given number of bytes of memory (data segment)	Not required
.byte	Allocates and assigns 1 byte of memory (data segment)	Required if initializing data
.half	Allocates and assigns 2 bytes of memory (data segment)	Required if initializing data
.word	Allocates and assigns 4 bytes of memory (data segment)	Required if initializing data
.equ	Substitutes a label for a value	Not required, but potentially helpful

Table 14.2: The short list of RISC-V assembler directives.

14.3.1.1 Instruction-Related Directives

Table 14.3 shows a summary of the two code-related directives. Probably the best way to present these two directives is with simple programs. Up to this point, the code we've written looked like the code in Figure 14.2. While this code is OK, we prefer to use the code in Figure 14.3, which is 100% equivalent, though it does appear different.

Directive	Usage	Comments
.text	.text	Takes no arguments; only instructions can follow directive
.equ	.equ lab, new_lab	Takes two comma-separated arguments, assembler replaces all instances of lab with new_lab

Table 14.3: The summary of code-type directives.

(00)	#-----			
(01)	init:	li	x10,0x11008000	# switch input port address
(02)		li	x11,0x1100C000	# LED output port address
(03)				
(04)	main:			
(05)	in_data:	lw	x20,0(x10)	# input switch data
(06)	out_data:	sw	x20,0(x11)	# write data to LEDs
(07)		j	main	# rinse, repeat
(08)	#-----			

Figure 14.2: A simple example problem.

There are several advantages to writing the program using the style in Figure 14.3. Here is the happy bulleted list of those advantages.

- We use the .equ directive on lines (01-02) to specify the input and output port address. These values are essentially constants, which is why we specify them using all capital letters. This is a common programming practice that all good programmers follow, so you should too.
- We put all the .equ directive in one area of the program and delineate nicely with the long dashed commenting style. You can spread these directives throughout your program if you choose, but that is really bad programming practice.
- Using the .equ directives is good for a few reasons. First, it makes the code somewhat self-commenting. Second, it makes the program more generic. The I/O addresses are generally fixed for a given hardware platform, but if they change, we want to have the changes in one spot only,

which allows us to only change the directive rather than values that may be spread through the program.

- The code also uses a `.text` directive. The program does not require this directive because the program is not defining a data segment, but we include it in order to provide more information to the human reader of the code.
- Although the `.equ` and `.text` directives increase the file size of your source code, they do not increase the size of program memory. And because they make the program more readable to humans, you should strive to use these.

```

(00) #-----
(01) .equ      SWITCHES,0x11008000    # port address of switches
(02) .equ      LEDES,0x1100C000     # port address of LEDs
(03) #-----
(04)
(05) .text                               # specify code segment
(06)
(07) init:     li      x10,SWITCHES  # switch input port address
(08)          li      x11,LEDES     # LED output port address
(09)
(10) main:
(11) in_data:  lw      x20,0(x10)    # input switch data
(12) out_data: sw      x20,0(x11)    # write data to LEDs
(13)          j       main          # rinse, repeat
(14) #-----

```

Figure 14.3: An clearer alternative to the simple example problem.

NOTE: for some unknown reason, the RARS assembler uses “.eqv” instead of “.equ” for that directive.

14.3.1.2 Data-Type Directives

We use data-type directives exclusively in the data segment. We use the `.data` directive to signify all that follows that directive is associated with the data segment. We use the data segment for intermediate storage of data and we typically want control of exactly how much data we are using and where in the data segment we are placing that data. Table 14.4 shows the list of data-type directives and an example of their usage. As with the code-type directives, the usage makes more sense when you see them in actual code. What is missing from Table 14.4 and the Venus and RARS assemblers in general is a directive to control where the assembler places the data in the data segment. This means we need to have various work-arounds to ascertain where exactly the data resides.

The `.space` directive allow programmers to “reserve” a specified number of “uninitialized” bytes of data. The other three directives allow programmers to both reserve and initialize that data in three different sizes: bytes, halfwords, and words, using the `.byte`, `.half`, and `.word` directives. Note: for some unknown reason, the Venus assembler does not recognize the `.half` directive.

Directive	Usage	Comments
<code>.data</code>	<code>.data</code>	no arguments; only data-type directives can follow directive
<code>.space</code>	<code>.space 10</code>	Reserve space for 10 bytes (uninitialized)
<code>.byte</code>	<code>.byte 4, -1, 0xFA</code>	Defines and initializes the list of data in 1 byte format
<code>.half</code>	<code>.half 0xFF00, 0xFA, -5</code>	Defines and initializes the list of data in 2 byte format
<code>.word</code>	<code>.word 0xFA780001, -1</code>	Defines and initializes the list of data in 4 byte format

Table 14.4: The summary of code-type directives.

Figure 14.4 shows an example using the `.data` and `.text` directives. Recall that using the `.text` directive is optional when the program does not need to specify a data segment. Here is the useful information regarding the code fragment in Figure 14.4:

- By convention, we list the data segment at the top of the program and before the code segment. We can break up the data and code segment as long as we correctly identify them using the `.data` and `.text` directives, but that is not good programming practice. We use delineation comments to clearly show the different segments.
- We reserve 20 bytes of “space” in memory using the `.space` directive on line (02). We don’t initialize the data in this area. The `.space` directive specifically reserves the number of bytes specified by the argument to the directive, but programmers can use data in this area to store byte, halfwords, or words of data.
- The `.space` directive on line (02) is not in the first column because we prefaced it with a label. The number associated with the “empty” label is thus the address of the first byte of 20 bytes of storage. We don’t at this point know where the assembler places that data in memory, but we’ll figure that out later and be able to work with the data at this location.
- We specify three bytes using the `.byte` directive on line (04). We use the “my_bytes” label so we can later access the location of the first byte of the three bytes of data. This data is initialized to the provided comma-separated values. The assembler stores negative values in an 8-bit 2’s complement format.
- We specify three halfwords using the `.half` directive on line (05). We use the “my_halfs” label to locate the data for future reference of all data specified by this directive. The assembler initializes the memory associated with the provided comma-separated values. The assembler stores negative values in a 16-bit 2’s complement format.
- We specify four words using the `.word` directive on line (06). The “my_words” label locates the data for future reference. The assembler initializes the memory associated with the provided comma-separated values. The assembler stores negative values in a 32-bit 2’s complement format.
- Because this program uses a data segment, we must explicitly specify a code segment using the `.text` directive on line (10). Only instructions and directive can follow the `.text` directive.
- We use the `la` pseudoinstruction on line (12) to retrieve the address of the data starting at the “empty” label. We then can store data at that address; we store a byte of data at that address using the `sb` instruction on line (14). The byte that we store is `0xB7`, which is the data we loaded into `x11` on line (13). We could have also stored halfwords or words in this memory location.
- We get the first byte in memory starting at the data associated with the “my_bytes” label using the `lbu` instruction on line (17). Note that we use “4” in the address offset for the `lbu` instruction, which points the address to the third halfword. We load the value “34” into `x11`.

- We get the third halfword in memory using the **lbu** instruction on line (20). Note that we use “4” in the address offset for the **lbu** instruction, which points the address to the third halfword. We load the value “0xFFE9” into x11.
- We get the fourth word in memory using the **lw** instruction on line (23). In this case, we use an address offset of “12” to point at the fourth word specified at this location. We load the value “0x00002355” into x11.
- We still have the address of the word data in x10, so we use that address to clear the first word at that address using the **sw** instruction on line (25).

```

(00) #-----
(01) .data                                     # data segment
(02) empty:      .space    20
(03)
(04) my_bytes:   .byte     34,-1,0xFA
(05) my_halfs:   .half     -4,0x4FAD,-0x23
(06) my_words:   .word     0x1100C000, -2, 0x34F, 9045
(07) #-----
(08)
(09) #-----
(10) .text                                           # code segment
(11)
(12) main:      la      x10,empty                    # load address of empty
(13)           li      x11,0xB7                     # place value in x11
(14)           sb      x11,0(x10)                   # store value in memory
(15)
(16) get_b:     la      x10,my_bytes                 # load address of my_bytes
(17)           lbu     x11,0(x10)                   # get data from memory
(18)
(19) get_h:     la      x10,my_halfs                 # load address of my_halfs
(20)           lhu     x11,4(x10)                   # get third half from address
(21)
(22) get_w:     la      x10,my_words                 # load data of my_words
(23)           lw      x11,12(x10)                  # get fourth word from address
(24)
(25) stor_w:    sw      x0,0(x10)                   # clear data at addr my_words
(26)
(27)           j      main                          # repeat pointless program
(28) #-----

```

Figure 14.4: A program using code and data-type directives.

Example 14.1: Address of Data

Answer the following questions using the code fragment that follows. For this problem, assume the value of `my_words` is 0x50.

- | | | |
|-----------------------------------|-------------------|-----------------|
| a) Value of <code>my_bytes</code> | d) Address of 6 | g) Value in x11 |
| b) Value of <code>my_halfs</code> | e) Address of -24 | h) Value in x12 |
| c) Value of <code>my_extra</code> | f) Address of 89 | i) Value in x13 |

```

.data      # data segment directive
my_words:  .word  4,5,6,7,9
my_bytes:  .byte  0x23,-24,46,-33
my_halfs:  .half  88,99,456
my_extra:  .word  988, 89

.text      # text segment directive
        la    x11,my_bytes
        la    x12,my_halfs
        la    x13,my_extra
stop:    j     stop

```

Solution: This is a classic problem type, which has somewhat of an issue. What this problem requires you to do in order to solve it is count in 1's, 2's, or 4's based on the whether you're counting byte, halfword, or word data respectively. The issue with this is that it is tedious and error prone. The reason it is error prone that you never need to do it in real life; you instead allow the assembler to handle the details.

The first thing to note in this problem that there are four labels associated with 12 pieces of data. There is nothing preventing programmers from using a unique label for each piece of data; doing so would not make the program any less space efficient (meaning it would require the same amount of data memory). Generally, you group data on the same line if you have some easier way to access the data on that line, which is the case for *look-up tables*, which we discuss later in this chapter. The other issue is that when we're programming and require data from the data segment, we generally don't know the actual address of that data, we only know how to easily access that data. Being able to access data is almost always more important than knowing exactly where that data lives.

- `my_bytes` is five words past the `my_words` label the precedes it. We only know the value of the `my_words` label (from the program description), so we work from there. The value of `my_words` is 0x50; the value of `my_bytes` is 5 (number of words on `my_words` line) * 4 (size of a word) greater than `my_words`, which is 0x64 (0x50 + 0x14).
- The value of `my_halfs` is four bytes of space beyond `my_bytes`, which is 0x64 + 0x4, or 0x68.
- The value of `my_extra` is three halfwords of space beyond `my_halfs`, which is 0x68 + 0x6, or 0x6E.
- The address of 6 is two words greater (because it's the third value on the word list) than the value of `my_words`, which is 0x58.
- The address of -24 is one byte greater (because it's the second on the byte list) than the value of `my_bytes`, which is 0x65.
- The address of 89 is one word greater (because it's the second on the word list) than the value of `my_extra`, which is 0x72.
- The code loads the value associated with `my_bytes` into x11, so x11 contains 0x64.
- The code loads the value associated with `my_halfs` into x12, so x12 contains 0x68.
- The code loads the value associated with `my_extra` into x13, so x13 contains 0x6E.

Example 14.2

Write a RISC-V assembly language program that counts the number of set bits in a range of words in memory. The starting point of the range and the quantity of numbers in the range are passed to the subroutine `x8` & `x10` respectively. The result is passed back to the calling routine in `x25`.

Solution: This solution is somewhat special because we include some test code for the solution. Anytime you write code, you should do your very best to test it before you “show it to anyone”, which mostly means before you submit it as part of an assignment. This example accesses memory, which means you have to be able to set values in memory to test the code, a task that is not always easy based on the assembler you’re working with. We’ll describe the problem in more detail once we’re done describing the solution.

First thing we need to do in all problems like this is to devise the steps that will lead us to the glory of a solution. There are two tasks in the problem: 1) grab words from memory, and 2) count the number of bits that are set in that word. The number of words to grab from memory is given, which could be zero, so we’ll control that with a while loop. The number of bits set in the word from memory can also be zero, so we’ll put that in a while loop also. This problem thus has two loops, one is on the interior of the other loop, which is a common situation we run into in all programming. Thus, the outer loop is the “get words from memory” and the inner loop is “count the number of bits in a word”. Check for that in the code below. Figure 14.5 shows the solution for this example including the test code, and lots of other stuff to check for.

- The subroutine has a header describing what the subroutine does, what values are sent to and returned from the subroutine, what registers the subroutine changes. Always do this.
- We clear the accumulator, which is our return value on line (22).
- The outer while loop starts on (24), where we check the count variable and exit the loop if it is zero.
- We then load a word of data from memory on line (25); `x20` now has the data we want to count the bits in.
- The inner while loop starts on line (27) where we count the data, which we do by masking the LSB on line (28), and adding the result of the making operation to the accumulator on line (29).
- The inner loop admin is on line (30) which is to shift the value loaded from memory to the right one bit position, and then jump to “`in_loop`” to repeat the inner loop. If the value is zero, we’re done. Recall that the shift right operation inserts a ‘0’ into the left-most bit position when it does a right shift.
- The outer loop administration starts on line (33), where we first decrement the counter, then advance the address pointer on line (34), before jumping to another iteration of the outer loop on line (35).
- The test code is on lines (13-18). The problem is that RISC-V assemblers do funny things with declared data, meaning the programmer has little control were the assembler places the data segment. To work around this peculiarity, we provide the data with a label on line (14); “junk” may not be the best label ever, but it works. We then use the `la` pseudoinstruction on line (18) to load the number associated with “junk” into `x8`, which effectively put the address of the first piece of data into `x8`. The code on line (17) places a 2 in `x10`, which is the quantity of data in the test code. The subroutine is now ready to test.
- Recall that if we don’t include a data segment, anything you write in the program defaults to the text segment. If our program needs to declare data, we must do so in the data segment, which we

do on line (13). Once we do this, everything we write is in the data segment until we declare a text segment, which we do on line (16).

```

(00) #-----
(01) # Subroutine: Count_bits:
(02) #
(03) # This subroutine counts the number of set bits in a range of words in
(04) # memory passed to the subroutine in x10 an starting at address passed to
(05) # the subroutine in x8. The result is passed back to the calling code in
(06) # x25.
(07) #
(08) # Passed values: x8 & x10
(09) # Returned values: x25
(10) # Tweaked Registers: x8,x10,x20,x21
(11) #-----
(12) # ---- test code -----
(13) .data
(14) junk:    .word    0xF,0x3          # assign some data
(15)
(16) .text
(17)         li      x10,2          # assign a count
(18)         la      x8,junk        # assign an address
(19) # ---- test code -----
(20)
(21) Count_bits:
(22) init:    mv      x25,x0          # clear counter
(23)
(24) out_loop: beq     x10,x0,done     # check loop count
(25)         lw      x20,0(x8)       # get data
(26)
(27) in_loop:  beq     x20,x0,admin     # exit inner loop
(28)         andi   x21,x20,1        # mask LSB of data
(29)         add    x25,x25,x21      # add LSB to counter
(30)         srli   x20,x20,1        # shift right one position
(31)         j      in_loop         # do it again
(32)
(33) admin:   addi   x10,x10,-1       # decrement loop count
(34)         addi   x8,x8,4          # increment address pointer
(35)         j      out_loop        # do it again
(36)
(37) done:    ret                    # take it home jimmie

```

Figure 14.5: A solution for this example.

14.4 Programming Efficiency Issues

Out there in computerland, there are always many different approaches to performing the same task. This is also true for assembly language programming. Although there are many different ways to do the same thing and obtain the same result, there are generally underlying differences in the code that affect how the code executes. This section describes a few of the more obvious issues, which we group into the notion of “programming efficiency”.

The term “programming efficiency” certainly sounds good. Suppose you tell your boss that the code you wrote is very efficient. If your boss is actually not just sitting there taking up space, she will ask you, “What makes your code efficient?”. The issue here is that there are different forms of efficiency. The two forms we discuss in this section are run-time efficiency and program memory space efficiency, which are generally the two most important issues in assembly language programming.

I always think of the example of knowing an algorithm that I can use to save the world. Sounds good, right? What if the algorithm requires too much program memory space to actually implement? What if you could implement the algorithm in a reasonable amount of code space, but it takes 5000 years to run the code? In these cases, your algorithm is useless, no matter how good it sounds. Conversely, if I had a program that ran “pretty fast” but required a bajillionquadrillion lines of code (thus too much memory to actually store somewhere), the

algorithm would be equally as useless. While these are extreme examples, they nicely describe issues that good programmers face every time they write code.

14.4.1 Iterative Construct Overhead

The underlying problem with iterative constructs is that they have associated “overhead”, which we refer to as loop overhead. This means that loop constructs contain instructions that don’t do anything useful other than maintain the iterative operational integrity of the construct. The instructions we refer to are the loop administration instructions (such as incrementing loop counts) and program flow instructions associated with the loop. This creates a well-known trade-off in coding: fast code vs. less code.

The idea behind fast code is that the code gets the task done faster; the idea behind less code is that the code itself takes up less space in the program memory. These two issues are always of great concern when writing programs, particularly in environments such as embedded systems, which are generally resource constrained. While we all want our programs to run super-fast, we can’t always do that if we’re writing code for an environment that has limited program memory. The best way to see this is in an example.

Example 14.3: Byte-Based Parity Generation

Write a RISC-V assembly language subroutine that calculates the parity of a byte in register x20. If x20 has even parity, it returns a ‘0’ in x20. Otherwise, it returns a ‘1’

Solution: There are a few ways to calculate parity using firmware; the approach in this problem the notion of counting the individual bits by masking, accumulating, and shifting. We won’t go over the programming details in this solution as we are more interested in the runtime and space efficiencies of the solutions.

Figure 14.6 and Figure 14.7 show two subroutines that solve the given program; these solutions are functionally equivalent but perform the task in different ways. The code in Figure 14.7 uses an iterative construct while the code in Figure 14.6 doesn’t use an iterative construct. The solution Figure 14.7 obviously has fewer instructions, but it must execute more instructions to arrive at the answer compared to Figure 14.6.

Your first look at these subroutines shows that there are fewer instructions for the code that uses an iterative construct (10 instructions vs. 27 instructions). This means that the code for the iterative construct requires less space in program memory. The execution of these programs tells another store. While the non-iterative subroutine requires 27 instructions to complete, the iterative subroutine requires 46 instructions. Thus, the subroutine with less about 1/3 less instructions requires almost twice as much time to execute.

The moral of the store is that the non-iterative version of the subroutines requires about twice as much program memory space, but runs twice as fast as the iterative version. This trade-off is something you always need to think about while programming in assembly language. The most easily applied issue associated with this is that you should never use an iterative construct that you know will iterate less than three times¹. Keep in mind that this is only a suggestion, and you should always have your brain engaged when programming. For example, if you had to iterate twice, don’t use a loop. However, if the code associated with the task you need to do twice requires 100 instructions, use a loop construct². One thing to consider here is that it is generally a good ideas to keep your iterative constructs as “single purpose” as possible.

¹ If you need to do a lot of work in your iterative construct, iteration counts of two are acceptable as it does save program memory space.

² Obviously, it’s really hard to make a black/white rule on this. Using your brain is always a better option than looking for rules to follow.

Subroutine	Number of instructions	Number of executed instructions
Get_par1	27	27
Get_par2	10	46

Table 14.5: A summary of efficiency statistics for both subroutines.

```

(00) #-----
(01) # Subroutine: Get_par1
(02) #
(03) # This subroutine determines the parity of the byte in x20. Parity
(04) # is returned in x20 where 1 and 0 equal odd and even parity, respectively.
(05) # The byte question is in the lower 8-bits of x20.
(06) #
(07) # Passed values: x20
(08) #
(09) # Tweaked registers: x20,x10,x11
(10) #-----
(11) Get_par1:
(12) init:      mv      x10,x0      # clear accumulator
(13)
(14) one:      andi    x11,x20,1    # mask LSB
(15)           add    x10,x10,x11   # accumulate bit
(16)           srli   x20,x20,1    # shift value
(17) two:      andi    x11,x20,1    # do 7 more times
(18)           add    x10,x10,x11
(19)           srli   x20,x20,1
(20) thr:      andi    x11,x20,1    # 3
(21)           add    x10,x10,x11
(22)           srli   x20,x20,1
(23) for:      andi    x11,x20,1    # 4
(24)           add    x10,x10,x11
(25)           srli   x20,x20,1
(26) fiv:      andi    x11,x20,1    # 5
(27)           add    x10,x10,x11
(28)           srli   x20,x20,1
(29) six:      andi    x11,x20,1    # 6
(30)           add    x10,x10,x11
(31)           srli   x20,x20,1
(32) sev:      andi    x11,x20,1    # 7
(33)           add    x10,x10,x11
(34)           srli   x20,x20,1
(35) eig:      andi    x11,x20,1    # 8
(36)           add    x10,x10,x11
(37)
(38) done:     andi    x10,x10,1    # mask LSB
(39)           mv     x20,x10      # transfer to x20
(40)           ret     # bring it on home
(41) #-----

```

Figure 14.6: A runtime efficient solution to this example.


```

(00) #-----
(01) # Subroutine: Get_par2
(02) #
(03) # This subroutine determines the parity of the byte in x20. Parity
(04) # is returned in x20 where 1 and 0 equal odd and even parity, respectively.
(05) # The byte question is in the lower 8-bits of x20.
(06) #
(07) # Passed values: x20
(08) #
(09) # Tweaked registers: x20,x10,x8
(10) #-----
(11) Get_par2:
(12)   init:      mv      x10,x0      # clear accumulator
(13)             li      x8,8        # load iterative count
(14)
(15)   loop:      andi   x20,x10,1    # mask LSB
(16)             add    x10,x10,x20  # accumulate
(17)
(18)   admin:     srli   x10,x10,1    # shift right one bit
(19)             addi   x8,x8,-1     # decrement loop count
(20)             j      loop        # rinse, repeat
(21)
(22)   done:      andi   x10,x10,1    # mask LSB
(23)             mv     x20,x10     # transfer to x20
(24)             ret     # bring it on home
(25) %-----

```

Figure 14.7: A codespace efficient solution to this example.

14.4.2 Subroutine Overhead Issues

We consider subroutines to have “overhead”, which means there are instructions associated with subroutines that we consider as doing “nothing useful”. In this case, the **call** and **ret** instructions essentially do nothing except handle the administrative tasks of the program flow control associated with calling and returning from subroutines. This means that anytime you call a subroutine, there are at two instructions worth of “doing nothing useful”. But wait, it gets worse. There are potentially two other forms of overheads associated with subroutines.

- 1) Subroutines typically save the operating context upon entering the subroutine, which generally comprises of pushing registers onto the stack. Additionally, once you push registers on the stack, you then need to pop them off the stack. Both pushing and popping operations are essentially instructions that don’t do anything useful but take time to execute in the process.
- 2) If you’re particular subroutine calls another subroutine, you need to push the return address onto the stack before the nested subroutine call and then pop it off afterwards. Yet more instructions that don’t do anything.

The issue of subroutine overhead is always something programmers need to consider. While we typically push programmers to write modular code, if your subroutines have a lot of overhead and don’t do that much “work”, your modular code won’t have runtime efficiency³. There’s an art to writing good subroutines that are part of a carefully architected program. We mention few items at the end of this chapter, but it primarily something that comes with experience and a lot of conscientious coding. Here is a somewhat meaningful example.

Figure 14.8 and Figure 14.9 show two code fragments that perform the exact same task. The code in Figure 14.8 adds a number to a register four times. The code in Figure 14.9 performs the same task, but does so by using a subroutine call. Yes, this is an overly simplified example, but it proves the point.

The code in Figure 14.8 performs the given task in using four instructions. The code in Figure 14.9 performs the same task, but requires a total of 12 instructions, thus requiring three times as much time to perform the same task. The difference in running times of these code fragments has to do with the subroutine call/return overhead. Specifically, each add operation has an associated **call** and **ret** instruction. These are the instructions that don’t do anything except perform administrative issues for the subroutine. Additionally, the code in Figure 14.8

³ This is a nerdy way of saying your program will be relatively slow.

requires less program memory space; it requires four instructions compared to the six instructions of Figure 14.9. The moral of the story is that you should strive to prevent the structure of your program from adding extra running time to your programs.

```
(00) #~~~~~ program fragment ~~~~~
(01)
(02)     addi    x8,x8,0x4           # add some value
(03)     addi    x8,x8,0x4           # etc.
(04)     addi    x8,x8,0x4           #
(05)     addi    x8,x8,0x4           #
(06)                                           ;
(07) #~~~~~ program fragment ~~~~~
```

Figure 14.8: Program fragment of some meaningless task.

```
(00) #~~~~~ program fragment ~~~~~
(01)
(02)     call    Add_four           # do something
(03)     call    Add_four
(04)     call    Add_four
(05)     ca;;   Add_four
(06)                                           ;
(07) #~~~~~ program fragment ~~~~~
(08)
(09) #-----
(10) #- Subroutine: Add_four -
(11) #
(12) #
(13) # Near meaningless subroutine, but serving as an excellent example.
(14) #-
(15) # Passed value: x8
(16) #
(17) # Tweaked registers: x8
(18) #-----
(19) Add_four:
(20)     addi    x8,x8,0x4           # change x8
(21)     ret     # bring it on home
(22) ;-----
```

Figure 14.9: A functionally equivalent fragment.

Example 14.4

What percentage of the code in Figure 14.10 would we classify as overhead?

Solution: This code is the declared bad code from a previous solution, but we'll continue working with it. Here's the big summary:

- The code has a total of 16 instructions. We'll call it 17 instructions because we'll in the **call** instruction from the calling code.
- The four instructions on lines (09-12) represent saving the current context; these instructions do nothing useful because we have to undo them later.
- The four instructions on lines (24-27) restore the context after the body of the subroutine executes. These instructions undo the context saving, so they do nothing useful either.
- The subroutine also has a **ret** instruction that does nothing useful.

In the end 10 out of the subroutine's 17 instructions (approximately 59%) do nothing. That means that over 50% of the time associated with the execution of this subroutine is dedicated to subroutine overhead. The

ramifications of these are that if your program only rarely calls this subroutine, you might as well not make it into a subroutine. What this means is that to recoup your losses from subroutine overhead, your program must call this subroutine often and particularly from different parts of the code. Note that if your program called this subroutine often but from inside the same loop, it would probably once again be better to not use a subroutine.

```

(00) #-----
(01) # Subroutine name: Swap_mem_ws
(02) #
(03) # This subroutine swaps two word values in memory. The address of the
(04) # values to swap is found in register x6 & x7.
(05) #
(06) # Tweaked Registers: none
(07) #-----
(08) Swap_mem_ws:
(09) init:      addi  sp,sp,-12      # make room on stack for storage
(10)           sw    x10,0(sp)      # push 3 items on stack
(11)           sw    x11,4(sp)
(12)           sw    x12,8(sp)
(13)
(14)           lw    x10,0(x6)      # get data to swap
(15)           lw    x11,0(x7)
(16)
(17)           mv    x12,x10        # copy data in x10 to working register
(18)           mv    x10,x11        # copy data from x11 to x10
(19)           mv    x11,x12        # copy working data to x11
(20)
(21)           sw    x10,0(x6)      # store swapped values
(22)           sw    x11,0(x7)
(23)
(24) restore:  lw    x10,0(sp)      # pop data into register
(25)           lw    x11,4(sp)
(26)           lw    x12,8(sp)
(27)           addi  sp,sp,12       # unadjust the stack pointer
(28)
(29)           ret    # transfer program control back
(30) #-----

```

Figure 14.10: A subroutine that uses data passed by address.

14.4.3 Program Space vs. Bullet-Proof Code Issues

As you have probably figured out by now, there are always many approaches to performing the same task when programming computers. You the programmer always face many subtle but important design decisions when writing your code. This section examines another subtle issue, yet clever opportunity for you to write “appropriate” code.

Example 14.5

Write a subroutine that multiplies the two halfword values stored in x10 & x10 together, and stores the result in x15.

Solution: We provide two different solutions to this example. The first solution in Figure 14.11 shows the barebones dumb-dood solution, while we refer to the solution in Figure 14.12 as “bullet proof”. If we’re speaking roughly, we can refer to these two subroutines as functionally equivalent, but only speaking roughly.

The solution in Figure 14.11 obviously has few instructions than the solution in Figure 14.12, but we don’t want to think that fewer instructions is somehow better. The truth is that the code in Figure 14.11 probably works “most” of the time and runs faster than the code in Figure 14.12 based on the number of instructions alone. But are you as a programmer satisfied with your code working most of the time. If you answered “yes”, then there are many job openings for academic administrators with your name on them. The problem with the code in Figure 14.11 is that it fails horribly in some common cases.

The code in Figure 14.11 has one initialization instruction followed by a do-while loop and its associated loop administration stuff. This code fails (does not provide the proper result) in two main areas. First, if the multiplier value is zero, the do-while performs one calculation including decrementing the multiplier. If the multiplier is 0, it becomes -1 (32 1's) after the decrement on line (15), and thus stay in the loop for a long time. Second, sending values other than halfwords to the subroutine also causes the subroutine to fail, as there may be values in the two upper bytes of x10 and x11, which is probably not what we want. Additionally, this subroutine changes several register values, which makes using this subroutine “troublesome” to use and reuse.

The code Figure 14.12 solves the problematic issues present in the previous solution. Here are the ways we resolve those issues:

- 1) We save the operating context by pushing the registers the subroutine uses on lines (14-17).
- 2) We check to see if the values are greater than 0x0000FFFF on lines (19-22), which indicates passed values are not halfwords. If we detect a non-valid value, we exit out of the subroutine, which is an arbitrary choice. In this case, it may be better to indicate an error condition in another register or memory location, which represents an even greater level of error detection that we don't want to deal with in this example.
- 3) We then check both operands for zero on lines (24-25), which serves two purposes. First, it ensures our do-while loop is valid in that there is no chance of decrementing a zero count. Second, it allows the subroutine to end faster in the case that one of the operand is zero, taking advantage of the fact if one of the operands is zero, the result is zero. If you know for sure that neither operand would ever be zero, you could not include this code.

The moral of this story is that there is a trade-off here. We wrote the code in Figure 14.12 so that it would always work and always work as efficiently as possible. The cost of doing this was that the subroutine required more code space and required more time to run. In all honesty, as the number of times the loop iterates becomes larger, the overhead associated with the extra code becomes less significant. So what is the best approach? Only you, the astute and knowledgeable programmer knows for sure. Note that you have to be knowledgeable of basic programming techniques and the system you're writing the code for.

```

(00) #-----
(01) # Subroutine: Mult_nums
(02) #
(03) # This subroutines multiples the two halfword values x10 & x11 and
(04) # stores the result in x15. The result is limited to 32 bits.
(05) #
(06) # Passed values: x10, x11
(07) #
(08) # Tweaked registers: x11, x15
(09) #-----
(10) Mult_nums:
(11) init:      mv      x15,x0      # clear accumulator
(12)
(13) loop:      add     x15,x15,x10  # accumulate result
(14)
(15) admin:     addi    x11,x11,-1   # decrement multiplier
(16)           beq     x11,x0,loop   # branch if not done
(17)
(18) done:      ret
(19) ;-----

```

Figure 14.11: A runtime efficient version of the Mult_nums subroutine.

```

(00) #-----
(01) # Subroutine: Mult_nums
(02) #
(03) # This subroutines multiples the two halfword values x10 & x11 and
(04) # stores the result in x15. The result is limited to 32 bits.
(05) #
(06) # Passed values: x10, x11
(07) #
(08) # Tweaked registers: x15
(09) #-----
(10) Mult_nums:
(11) init:      mv      x15,x0      # clear accumulator
(12)           li      x20,0xFFFF0000 # upper half mask value
(13)
(14) store:     addi    sp,sp,-12    # adjust stack pointer
(15)           sw      x11,0(sp)    # push: store context
(16)           sw      x12,4(sp)
(17)           sw      x20,8(sp)
(18)
(19) chk_size:  and     x12,x10,x20    # mask multiplicand: verify half
(20)           bne    x12,x0,done    # error condition
(21)           and     x12,x11,x20    # mask multiplier: verify half
(22)           bne    x12,x0,done    # error condition
(23)
(24) chk_zero:  beq     x10,x0,restore # check multiplicand for zero
(25)           beq     x11,x0,restore # check multiplier for zero
(26)
(27) loop:      add     x15,x15,x10  # accumulate result
(28)
(29) admin:     addi    x11,x11,-1   # decrement multiplier
(30)           beq     x11,x0,loop   # branch if not done
(31)
(32) restore:   lw      x11,0(sp)    # pop: restore context
(33)           lw      x12,4(sp)
(34)           lw      x20,8(sp)
(35)           addi    sp,sp,-12    # adjust stack pointer
(36)
(37) done:      ret
(38) #-----

```

Figure 14.12: A “bullet-proof” version of the Mult_nums subroutine.

14.5 Look-Up Tables (LUTs)

Generally speaking, anytime you can use a LUT in your hardware or software, you do so. In hardware, we can use LUTs to implement Boolean functions, which is really handy when the equations become knarly. In software/firmware, we can use LUTs to reduce the size of programs (and thus they run faster) by not repeating calculations or by not having to conduct long if/else clause (or case statements) in our code.

LUTs should be nothing new to you at this point, or at least the concepts behind LUTs. This is because LUTs are analogous to arrays in higher-level programming languages⁴. An array is a structure that holds data; we access this data using the base address of the array (the address of the first piece of data in the array) plus some offset. We refer to the offset we provide the array as the “index”; when we retrieve data from the array, we say we are indexing into the array.

LUTs in assembly language are a true mix of software and firmware techniques. We need to store the data in memory to make it accessible to the program. The assembler provides instruction so place data into the array using assembler directives. The associated ISA provides instructions to access that data as needed. We store the LUT somewhere in the data segment; we access the LUT using the standard set of load and store instructions.

There are many advantages to using a LUT, particularly in firmware applications such as display multiplexing. The use of LUTs in computer programming is typically well supported by the underlying assembly languages, which certainly underscores their usefulness. The RISC-V MCU ISA supports LUTs without any type of special instructions; the use of assembler directives and the load & store instructions are adequate.

Using a LUT on the RISC-V MCU requires three steps. 1) generate the data that goes into the LUT, 2) store the data in an accessible area of the data segment, and 3) access the LUT using the RISC-V MCU’s instruction set. The best way to present this information is with an example.

Example 14.6: LUT-Based Parity Subroutine 4-Bit Version

Write a RISC-V assembly language subroutine that determines the parity of the value in x10. If x10 has even parity, it returns a ‘0’ in x20. Otherwise, it returns a ‘1’

Solution: There are a few ways to calculate parity using firmware; the approach in this problem uses a LUT because it runs faster than other version. Then again, it requires more data memory than other versions, which is a tradeoff that we’ll discuss later. This solution uses the LUT to determine the parity of a nibble; to complete solution requires that we determine the parity of each of the eight nibbles in the 32-bit register. The algorithm accumulates the number of set bits; the LSB of the accumulated value is then the parity. Figure 14.13 shows the complete solution; here are the details with an emphasis on the LUT portions of the program:

- We are defining a LUT, which is a section in data memory, so we start off by working in the data segment as noted by our use of the `.data` assembler directive on line (01).
- The first step in using a LUT is to define the data that goes into the LUT. We sort of did this in our heads for this program, but make sure you understand what the data means before you continue on in this solution.
- The second step in using a LUT is to put the data into memory. The two `.byte` directives tell the assembler to place the data into memory; we don’t know where exactly the assembler is putting the data, but we’ll be able to access it because we included the “`par_val`” label. The two `.byte` directives define 16 values; these values correspond to the number of bits that are set in a 4-bit number ranging from [0,15]; 0 through 15 are the decimal equivalents of each possible 4-bit value. For example, the eighth value in the line on (04) is “3”, which corresponds to the fact that the eighth value in the [0,15] is “7”, or “0111”. Because the value “7” has three bits set, we placed a “3” at this data location.

⁴ Where there is a possibility that you have not used a real LUT in your previous programming experience, you absolutely should have used an array of some type. Or at least I hope you did. Consider having a talk with your programming instructor if you did not use a LUT or especially an array.

- We used two `.byte` directives for clarity and neatness; we could have used only one.
- The initialization sequence of the subroutine includes loading the iteration count with 8 (for 8 nibbles) on line (24), clearing an accumulator register on line (25), and most importantly, loaded the address of the LUT into a register using the `la` pseudoinstruction on line (26). `x30` now contains an address, which is the address of the first piece of data in the LUT.
- We used a while loop for the body of the code and check the loop variable on line (28).
- The next task is step 3) in using a LUT: accessing the LUT data. We then need to mask all but the right-most nibble to use as an index into the LUT, which we do on line (29). We don't know what that nibble value is, but we use that value as an index into the LUT, which we do by adding the nibble (offset) the base address of the LUT to form the absolute address of the data we're looking for in the LUT. We do the calculation on line (30) and the actual LUT access (look-up) on line (31). We accumulate the value we "looked up" on line (32).
- The administrative part of the loop is to shift right the data by a nibble on line (34) and then decrement the loop counter on line (35).
- When the code breaks out of the while loop, the value in `x15` contains the number of bits that were set in `x10`. We can use the LSB as the parity, but first we must mask all but the LSB, which we do on line (39).

```

(00) #-----
(01) .data # define data segment
(02) # num of bits set in each nibble (range: [0,15])
(03)
(04) par_val: .byte 0,1,1,2,1,2,2,3 # values 0 -> 7
(05) .byte 1,2,2,3,2,3,3,4 # values 8 -> 15
(06) #-----
(07)
(08) #-----
(09) # Subroutine name: Par_32b
(10) #
(11) # This subroutine uses a LUT-based approach to calculate parity of x10
(12) # by adding the parity values of the 8 underlying nibbles. The LUT
(13) # thus holds the parity values for each of the 16 possible number that
(14) # the nibble can represent. The 8 look-ups are added and the LSB is
(15) # the parity value.
(16) #
(17) # Passed values: x10
(18) #
(19) # Tweaked values: x10, x15, x20, x30, x21, x22
(20) #-----
(21) .text
(22)
(23) Par_32b:
(24) init: li x20,8 # loop count
(25) mv x15,x0 # bit count
(26) la x30,par_val # get address of LUT
(27)
(28) loop: beq x20,x0,done # done yet?
(29) andi x21,x10,0xF # calc table offset
(30) add x22,x30,x21 # calc index
(31) lbu x22,0(x22) # table look-up
(32) add x15,x15,x22 # accumulate
(33)
(34) admin: srli x10,x10,4 # shift right one nibble
(35) addi x20,x20,-1 # decr loop count
(36) j loop # rinse, repeat
(37)
(38) done: mv x10,x15 # load count to x10
(39) andi x10,x10,1 # mask LSB
(40) ret # take it on home
(41) #-----

```

Figure 14.13: A program using code and data-type directives.

The beauty of this approach for accessing data maybe can only be appreciated by those people who have written functionally equivalent code not using a LUT. The cool thing about this code is that we never had to figure out what the value of the nibble was (meaning we did not have to use a bunch of if/else constructs to figure it out); we instead simply used that value as an index, or offset, into the LUT. The moral of this story is that with any program you write, you should always ask yourself: “How am I going to use a LUT to make this program easier to write and more efficient?” Seriously, asking this question of yourself should be automatic in any program you write⁵.

Example 14.7: LUT-based Parity Subroutine 8-Bit Version

Don’t actually do it, but describe how you would repeat the previous problem using a LUT with 256 entries. Discuss the obvious space and run time efficiencies involved.

⁵ Similarly, when you’re designing digital hardware, you should always be asking yourself how you can use a generic decoder to simplify your circuit.

Solution: It's the same problem, but this time we'll use a LUT with 256 entries, which would represent the number of set bits for each of the 256 unique values that an 8-bit number can represent. For a problem such as this, you'd for sure want to use some other software to write a program that generated the LUT code for you; you certainly would not want to count all the bits and type it all in. The while-loop in the code would now only need to be iterated four times. The main result here is that the subroutine runs about twice as fast, but the required data memory increased from 16 bytes to 256 bytes. You, the astute programmer, would need to decide if that was worth it.

14.5.1 LUTs Revisited

LUT implementations are a tradeoff between space and run-time efficiencies. The underlying details are that it is computationally more efficient to “look something up” than it is to search for or calculate it. LUTs can make code run faster, but it comes at the prices of requiring extra space in memory to store the LUT. While LUTs can be quite helpful, the larger they are, the more memory space they consume. In both cases, you must make sure your computational savings of using a LUT justifies the memory space required to represent that LUT in memory.

- The act of searching for something in this context means that you're iteratively searching for a particular value associated with a given value, which typically implies you encode this search with an if/else or case structure. If the given value could be one of many different values, then the supporting search structure could be very large and subsequently very slow.
- The act of calculating a result associated with a value can require a significant amount of computing resources. In this case, it would make sense to use a LUT under the condition that you have to perform the calculation relatively often in your code.

14.6 Chapter Summary

- Memory is important to computers, which is why instructions set generally have different ways of addressing memory. The term *address space* refers to the total amount of memory an instruction set can access, but not all of this memory is physical memory; it includes other memory space items such as I/O port addresses.
 - Computer memory is typically divided into segments; the segments used in the RISC-V include:
 - Code segment: stores program memory (physical memory)
 - Data segment: stores LUTs and other data (physical memory)
 - Stack Segment: store data used by program (physical memory)
 - Memory mapped I/O: used to differentiate different computer peripherals (not physical memory)
 - There are currently three RISC-V assemblers available: Venus, RARS, and gcc. Each of them has their good and bad points as listed in this chapter.
 - Different assemblers have a different set of *assembler directives*, which allow programmers to control certain aspects of the assembler. Assembler directive as essentially messages from the programmer to the assembler. It's good to know which directives an assembler supports before using that assembler.
 - There are two main types of directives: 1) those that support instructions, and 2) those that support code.
 - Writing efficient assembly language programs is more of an art form than a science. Many aspects of assembly language programs have efficiency issues including. There is always a tradeoff between runtime and program space efficiencies, for example, larger programs (more program memory) often run faster than functionally equivalent smaller programs. Areas where programming efficiencies are an issues include
 - Iterative loops: instructions that check loop conditions and/or handle program flow control (branching) don't do meaningful work
 - Subroutines: calling/returning from subroutines are program flow control instructions that don't do meaningful work. Saving context and saving return addresses (for nested subroutines) also do not do anything.
 - We can write "bullet proof" subroutines that work no matter when and where you call them, which includes checking all iteration counts and saving/restoring context.
 - There are two main efficiency issues in assembly language programming: run-time efficiencies and program memory space efficiencies. The programmer needs to be aware of this trade-off and program their computer appropriately.
-

14.7 Chapter Exercises

- 1) Briefly describe the difference between *address space* and the *physical address space* of a memory access instruction.
- 2) Briefly describe what or who decides how much physical address space is available in a given system.
- 3) Briefly describe what or who delineates the boundaries between the code and data segments in the RISC-V OTTER MCU.
- 4) Briefly describe how stack segment can encroach on other segments.
- 5) Briefly describe if there is actually a physical boundary between any segments in physical memory space.
- 6) Briefly describe whether it would be possible for a programmer to write programs without understanding and being familiar with the memory map.
- 7) Where do you typically find most errors in assembly language programs?
- 8) What is the question that programmers should always be asking themselves when they're writing source code?
- 9) What are the three steps required in order to use a LUT on the RISC-V MCU?
- 10) Describe how LUTs can help programmers create efficient code.
- 11) What's the general rule to using an iterative construct or not in programming?
- 12) Briefly describe the overhead associated with iterative loops.
- 13) Briefly describe the two types of overhead associated with subroutines.
- 14) Describe the difference between accessing a LUT located at address 0xF0000010 and a LUT located at 0xF0000020. For this problem, assume each LUT has ten locations.
- 15) Describe the two types of programming efficiencies in the RISC-V MCU assembly language.
- 16) Describe why assembly code that has more instructions can have a shorter running time than code that has fewer instructions.
- 17) Briefly describe the relationship between the number of times a loop iterates, the amount of non-overhead code in the loop, and the overall efficiency of the loop.
- 18) Answer the following questions using the code fragment that follows. For this problem, assume the value of **xwords** is 0x00000F00.
 - d) Value of **xbytes**
 - e) Value of **xwords2**
 - f) Value of **xmore**
 - g) Address of -33
 - h) Address of 459
 - i) Address of 0xDD0
 - j) Value in x21
 - k) Value in x22
 - l) Value in x23

```

.data      # data segment directive
xwords:   .word   0xAF0,0xBF0,0xCF0,0xDD0,0xFF0, 0x3E0
xbytes:   .byte   0xAA,-0x32,58, 23,-33,-121
xwords2:  .word   88,99,459
xmore:    .half   344,456

.text      # text segment directive
          la      x21,xbytes
          la      x22,xwords2
          la      x23,xmore
stop:     j      stop

```

19) Answer the following questions using the code fragment that follows. For this problem, assume the value of **bwords** is 0x000010F0.

- | | | |
|---------------------------|----------------------|----------------|
| g) Value of chalfs | j) Address of 0xFFCC | m) Value in x7 |
| h) Value of dbytes | k) Address of 48 | n) Value in x8 |
| i) Value of ewords | l) Address of 34605 | o) Value in x9 |

```
.data    # data segment directive
bwords: .word  0xFF03,0xAB30,0xFD00,0xEE00,0xF3DE
chalfs:  .half  0xFFFE,0xFFCC,0xFFAA,0xFF11
dbytes:  .byte  0x4,-0x6,48,123,-93,128
ewords:  .word  0x4555,34605,-0x8958

.text    # text segment directive
        la     x7,chalfs
        la     x8,dbytes
        la     x9,ewords
        addi   x7,x7,3
        addi   x8,x8,4
        addi   x9,x9,5

kill:   j      kill
```

20) The following two subroutines generates are described by their headers.

- What percent of instruction in the subroutine are considered subroutine overhead?
- What percentage of the instructions executed by the subroutine are subroutine overhead?

```
#-----
# Subroutine: bcd_to_bin
#
# Converts a 3-digit decimal number represented in the lowest three nibbles
# of x10 to the equivalent unsigned binary value and places the result in x20.
#
# Tweaked Registers: x20
#-----
bcd_to_bin:
init:
store:   addi   sp,sp,-12      # adjust sp to save 3 regs
        sw    x21,0(sp)      # save x21
        sw    x15,4(sp)      # save x15
        sw    x10,8(sp)     # save x10

        li    x21,0x00000F00 # 100's bit mask
        mv    x20,x0        # zero accumulator

t_100:  and    x15,x15,x21     # mask 100's nibble
        srli  x15,x15,8      # shift to lowest position
loop1:  beqz   x15,t_10        # go to tens if zero
        addi  x20,x20,100    # accumulate 100s
        addi  x15,x15,-1     # decrement loop count
        j     loop1         # do it again

t_10:   lw    x15,8(sp)      # load original value
        srli  x21,x21,4      # shift the mask value to next nibble
        and   x15,x15,x21    # mask 10's nibble
        srli  x15,x15,4      # shift left to right-most position
loop2:  beqz   x15,t_1        # move on if it's zero
        addi  x20,x20,10     # accumulate 10 values
        addi  x15,x15,-1     # decrement loop count
        j     loop2         # do it again

t_1:    mv    x15,8(sp)      # load original value
        srli  x21,x21,4      # shift the mask value to next nibble
        and   x15,x15,x21    # mask bits
        add   x20,x20,x15    # add value to accumulator
```

```

restore:  lw    x21,0(sp)      # restore x20
          lw    x15,4(sp)     # restore x20
          lw    x10,8(sp)    # restore x20
          addi  sp,sp,12     # adjust sp after restoring 3 regs

done:    ret                  # take it on home

```

```

#-----
# Subroutine: gen_fib_16
#
# Generates the first 16 Fibonacci numbers (starting with 1,1,...) and
# stores the numbers as halfwords starting at address stored in x25.
#
# Tweaked registers: none
#-----
gen_fib_16:
store:   addi  sp,sp,-20     # adjust sp to save 5 regs
          sw   x20,0(sp)     # save x20
          sw   x21,4(sp)     # save x21
          sw   x25,8(sp)     # save x25
          sw   x15,12(sp)    # save x15
          sw   x16,16(sp)    # save x16

init:    li    x20,14        # load loop count
          li    x21,1        # load initial fib number

          sh   x21,0(x25)    # store first two fib numbers
          sh   x21,2(x25)
          addi x25,x25,4     # adjust the pointer forward

loop:    beq   x20,x0,done   # done yet?
          lhu  x15,-4(x25)   # get two previous values
          lhu  x16,-2(x25)
          add  x15,x15,x16   # add two previous value
          sh   x15,0(x25)   # store result of addition
          addi x20,x20,-1    # loop admin: decrement loop count
          addi x25,x25,2     # increment pointer forward
          j    loop         # repeat, rinse

restore: lw    x20,0(sp)     # restore x20
          lw    x21,4(sp)   # restore x21
          lw    x25,8(sp)   # restore x25
          lw    x15,12(sp)  # restore x15
          lw    x16,16(sp)  # restore x16
          addi  sp,sp,20    # adjust sp to back to original value

done:    ret

```

- 21) Rewrite the following to subroutines and make them “bullet proof”, in other words, safe to call in all circumstances.

```

#-----
# Subroutine: parity
#
# Determines the parity of the value in x10; returns '0' in x20 if parity
# is even, otherwise returns '1'.
#
# Tweaked Registers: x10, x15, x20
#-----
parity:

init:    mv    x15,x0        # clear an accumulator

loop:    beq   x10,x0,done   # check to see if were done
          andi  x15,x10,1    # mask LSB
          add  x20,x20,x15   # increment bit count

```

```

        srl  x10,x10,1    # shift value right 1 bit
        j    loop        # rinse, repeat

done:   andi  x20,x20,1   # clear all but LSB
        ret           # take it home jimie

```

22) Briefly describe under what condition will the following subroutine work as described?

```

#-----
# Subroutine: abs_mem
#
# This subroutine multiplies takes the absolute value of signed bytes
# in memory starting at the address in x10, and does this for the number
# of values represented by the count in x11.
#
# Tweaked registers: none
#-----

abs_mem:
store:   addi  sp,sp,-12   # room on stack
        lw    x10,0(sp)   # push regs
        lw    x11,4(sp)
        lw    x12,8(sp)

init:    # nothing to init

loop:    lb    x20,0(x10)  # load value
        bge   x20,x0,write # br if > 0

write:   neg   x20,x20     # change sign
        sb    x20,0(x10)  # store value

admin:   addi  x10,x10,1   # incr addr
        addi  x11,x11,-1  # decr loop count
        j    loop        # do again

done:
rstore:  lw    x10,0(sp)   # pop regs
        lw    x11,4(sp)
        lw    x12,8(sp)
        addi  sp,sp,12    # adjust sp

        ret           # bring it home

```

14.8 Chapter Programming Exercises

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code, including proper headers for subroutines
- 1) Write a RISC-V assembly language subroutine that determines how many if all eight nibbles of register x8 have the same parity. The subroutine returns a non-zero value in x8 if all the nibbles have the same parity; otherwise the subroutine returns zero in x8. Use a LUT for this subroutine. Don't permanently change any registers other than x8.
 - 2) Write a RISC-V assembly language subroutine that determines if all eight nibbles of register x8 the same number of bits that are set. The subroutine returns a non-zero value in x8 if all the nibbles have the same number of bits set; otherwise the subroutine returns zero in x8. Use a LUT for this subroutine. Don't permanently change any registers other than x8.
 - 3) Write a RISC-V assembly language subroutine that determines if all eight nibbles of register x8 the same number of bits that are set and have even parity. The subroutine returns a non-zero value in x8 if all the nibbles have the same number of bits set and are even parity; otherwise the subroutine returns zero in x8. Use a LUT for this subroutine. Don't permanently change any registers other than x8.
 - 4) Write a RISC-V assembly language subroutine that determines whether the value in x10 is a prime number or not. The value in x10 always falls into the following range: [2,25]. If the number in x10 is prime, the subroutine returns a non-zero value in x10; otherwise it returns zero in x10. Use a LUT for this subroutine. Don't permanently change any registers other than x8.
 - 5)
-

15 RISC-V Solved Programming Problems

15.1 Introduction

The only way to learn about assembly language programming is to actually do some assembly language programming. The previous chapters spoke about the basic mechanics of assembly language programs, but only provided a few basic examples. This chapter presents nothing new, but presents all of the older assembly language programming ideas in the context of example problems. The problems start out easy and become more challenging as the chapter progresses. The idea here is that if you understand all the example programs in this chapter, then you'll know about all the tricks associated with assembly language program.

Keep in mind that one of my theories of assembly language programming that if you see and understand a trick once, you can put that trick in your bag of tricks and then be prepared to whip it out whenever you need it. Remember, assembly language programming is the same instructions and constructs arranged in different orders such that your program solves the problem at hand.

Main Chapter Topics

- **NO NEW TOPICS:** This chapter presents all previously presented stuff in the context of actual example assembly language programs.
- **C PROGRAMMING CONVERSIONS:** This chapter show how common C programming constructs translate into RISC-V assembly language code.

Why This Chapter is Important

This chapter is important because it shows how to solve a wide set of problems by writing RISC-V assembly language programs.

15.2 Introductory RISC-V Programming Problems

Here are a few introductory RISC-V programming problems. Each solution contains pertinent highlights as well as the well-commented source code.

Example 15.1: Continuous I/O

Write a RISC-V OTTER assembly language program that continually reads data from the input port associated with the switches, complements that data, and outputs the data to the port associated with the LEDs. Consider the address of the switches and LEDs to be `0xC000_0004` and `0xC000_000A`, respectively.

Solution: This program tries to do something meaningful in that we're reading in data from the outside world, tweaking it, and then writing back out to the outside world. The actual topic of I/O is really important, but relatively simple on the programming level. We'll get into more details later, but for now, just go with it.

Here's the quick I/O overview. The RISC-V MCU uses "memory-mapped" I/O, which is one of several common approaches to performing I/O on a computer system. The notion here is that I/O and reading/writing memory use the same instructions. Recall that the RISC-V MCU has a memory that programmers can use to

store intermediate values. What makes this work is that the computers configure the hardware such that when it sees a particular address, it knows that it needs to perform I/O rather than performing a memory read or write. This being the case, when we load a value from that particular address in memory (using a load instruction), the instruction accesses the data from the outside world and places it in the specified source register. When we store a value at a that particular address in memory (using a store instruction), the instruction takes the data from a general purpose register and makes it available to the outside world to do something with. Lots more on this later; for now: *input = load; output = store.*

```

(00) #-----
(01) # Read in data from the port connected to the switches, compliment
(02) # the data, then output the data to the port connected to the LEDs.
(03) # The port addresses for the I/O is listed in the code.
(04) #-----
(05) .text                               # we're in the text segment
(06)
(07) init:    li      x10,0xC0000004      # input port for switches
(08)         li      x11,0xC000000A     # output port for LEDs
(09)
(10) main:    lw      x15,0(x10)         # input data from switches port to x15
(11)         xori   x15,x15,-1         # compliment data
(12)         sw     x15,0(x11)         # output data to LED port
(13)         j      main              # do it again

```

Figure 15.1: Solution to this example problem.

Solution Notes: Fun stuff embedded in the solution.

- The first five lines provide a nice explanation of what is going on. All good programs include a very neat header such as this.
- Line (05) uses an assembler directive to specify that we're in the "text" segment. All code goes in the text segment; this will make more sense later when we talk more about memory segmentation in the MCU.
- The most straight forward way to get data from the outside world is to place the address associated with the switches and LEDs into a register. The **li** instruction stands for "load immediate"; it loads the immediate data specified in the instructions in lines (07-08) into the listed registers. The x10 & x11 registers are arbitrary. Note that someone needs to give you the programmer these addresses. Some hardware person configured them; that person needs to state how to access I/O in the hardware.
- The actual input instruction is the **lw** instruction on line (10). The first operand specifies which register is written with the external data; the second operand is the address. The "0(x10)" notation specifies the port address, which is officially zero added to the value in x10. A previous instruction put the address value into x10.
- The input data is then complimented using an **xori** instruction, which stands for "exclusive OR immediate". The instruction reads the value from the source operand (the right-most x15), does a bit-wise exclusive OR with -1, and stores the value into x15. The assembler represents negative numbers using 2's compliment notation, so -1 is encoded as all 1's (0xFFFFFFFF). The instruction takes the data from x15, operates on it, then stores the data back into x15.
- Line (12) is the output operation. Note that it uses a "**sw**" instruction, which stands for "store word". The **sw** instruction takes the data from register x15 and makes it available to the outside world.
- The problem specified to do this operation over and over again, so line (13) directs program control back to the line (10), which is the instruction that performs an input. Note that we don't go back to line (07) because that data is already in the registers and no instruction changed it.

- All the instructions and comments are nicely aligned.
- All labels start in the left-most column.
- All assembler directives start in the left-most column
- We used two pseudoinstructions in this code: **li** & **j**. We could have used the **not** pseudoinstruction in place of the **xor** instruction. The code in below shows the equivalent pseudoinstruction.

```

(00) #-----
(01) # Read in data from the port connected to the switches, compliment
(02) # the data, then output the data to the port connected to the LEDs.
(03) # The port addresses for the I/O is listed in the code.
(04) #-----
(05) .text                                # we're in the text segment
(06)
(07) init:    li        x10,0xC0000004    # input port for switches
(08)         li        x11,0xC000000A    # output port for LEDs
(09)
(10) main:    lw        x15,0(x10)        # input data from switches port to x15
(11)         not       x15,x15           # compliment data
(12)         sw        x15,0(x11)        # output data to LED port
(13)         j         main              # do it again

```

Figure 15.2: An alternate solution to this example problem.

Example 15.2: Continuous Output Sequence

Write a RISC-V OTTER assembly language program that continually outputs the following sequence to the output port specified by address 0xC000_00D0. Don't use more than two registers in your design.

```
{...0x1, 0x2, 0x4, 0x8, 0x4, 0x2, 0x1, 0x2...}
```

Solution Notes: Fun stuff embedded in the solution. There are definitely better approaches to this problem, but you don't have those items in your toolset as of yet.

- The first five lines provide a nice looking header. Nice a judgment call, but I'm practicing to be an administrator so I consider all my work in the nice to great range.
- The first thing to note when doing this problem is that the sequence is a single bit in the LSB of a number that moves to the left and back to the right until you want to hurl. This reminds us of a shifting left and shifting right operations that we loved so much from our days working with shift registers. Lucky for us programmers there are instructions in the RISC-V instruction set that shift left and right. These instructions are the **slli** and **srli** instructions, which are mnemonics for "shift left logical immediate" and "shift right logical immediate".
- There is always an issue of what the processor stuffs in the right side of that data when it shifts left (and vice-versa with the shift right). Does it stick in a '1' or a '0'? There is no magic to this, you need to check the RISC-V spec to find out. After you do that, you'll be relieved to know that the instructions shove in '0's', so the instructions are perfect for this problem.
- There is also plain shift left and shift right instructions (**sll** & **srl**), which use the lowest five bits of a register location as the number of bits to shift. We can't use these for this

problem because the problem nefariously stated we could only change two registers. Be sure to enhance your excitement by checking out these instructions in the RISC-V spec.

- The set of shift lefts and rights synthesize the required values; the values are then output to the specified output port.
- To repeat the sequence forever, jump back to the instruction associated with the **main** label to allow the fun to continue.

```

(00) #-----
(01) # Outputs the following sequence to output port 0xC00000D0:
(02) #
(03) # {...0x1, 0x2, 0x4, 0x8, 0x4, 0x2, 0x1, 0x2...}
(04) #-----
(05) .text
(06)
(07) init:      li    x10,0xC00000D0    # store address in register
(08)          li    x20,0x01          # initial value of sequence
(09)
(10) main:     sw    x20,0(x10)        # output first value in sequence (1)
(11)          slli  x20,x20,1         # shift left 1 spot
(12)          sw    x20,0(x10)        # output second value in sequence (2)
(13)          slli  x20,x20,1         # shift left 1 spot
(14)          sw    x20,0(x10)        # output second value in sequence (4)
(15)          slli  x20,x20,1         # shift left 1 spot
(16)          sw    x20,0(x10)        # output second value in sequence (8)
(17)
(18)          srli  x20,x20,1         # shift right 1 spot
(19)          sw    x20,0(x10)        # output second value in sequence (4)
(20)          srli  x20,x20,1         # shift right 1 spot
(21)          sw    x20,0(x10)        # output second value in sequence (2)
(22)          srli  x20,x20,1         # shift right 1 spot
(23)          j     main              # jump to first output ad nasuem

```

Figure 15.3: Solution to this example problem.

Example 15.3: Half Swap

Write a RISC-V assembly language subroutine that swaps the upper two bytes in x10 with the lower two bytes in x10.

Solution Notes: Fun stuff embedded in the solution. There are arguably better ways to do to this problem; we'll take the most straightforward approach.

- This is a subroutine, so we give it a nice header. The header includes the name of the subroutine, a brief description of the subroutine, and a list of what registers the subroutine changes. All this information is massively important to anyone who may want to your subroutine. The notion here is that the registers are shared by all the code, so if the subroutine changes a register that the calling code is working with, that is really ungood. There are ways to prevent this, but that is a “stack” issue; we'll deal with that later.
- The subroutine name is on line (08); it is a simple label, which means there is a value associated with it, and that value is the location in program memory of the first instruction in the subroutine. For what it's worth, the labels “init” and “Big_swap” have the same numerical value; we include both so as not to confuse the human readers of the code.
- The code has three initialization related instructions on lines (10-12). The two **li** instructions are to place the mask values into registers. We need to do that because the **andi** instruction has a limited amount of space for mask values, so we opt to put the mask value in a register and

use the **and** instruction for the masking operations. The code on line (12) makes a copy of the register with the data we need to swap.

- The swapping action is this: we clear the upper two bytes of one register and shift the result left 16 places, clear the lower two byte of the other register and shift the result right by 16 placed, then combine the results. The **slli** and **srlr** instructions handle the shifting operations.
- The results of the two shifting operations are combined using an OR instruction on line (20). We could have combined them with an **add** instruction or an **or** instruction; the choice is arbitrary.
- We conclude the subroutine with a **ret** instruction, which stands for “return”. This is a pseudoinstruction, but it works rather nicely if we’ve called the instruction using the **call** pseudoinstruction (more on that later).
- The “done” label is never called; it serves as a comment to human readers of the code.

```

(00) #-----
(01) # Subroutine: Big_swap
(02) #
(03) # This subroutine swaps the upper two bytes with the lower two bytes
(04) # in register x20.
(05) #
(06) # Tweaked Registers: x10, x15, x20, x21
(07) #-----
(08) Big_swap:
(09)
(10) init:    li    x20,0x0000FFFF    # lower bit mask
(11)         li    x21,0xFFFF0000    # upper bit mask
(12)         mv    x15,x10           # make a copy
(13)
(14) upper:  slli  x15,x15,16        # move lower 2 bytes 16 bits to left
(15)         and  x15,x15,x21        # clear lower 16 bits
(16)
(17) lower:  srlr  x10,x10,16        # move upper 2 bits 16 bits to right
(18)         and  x10,x10,x20        # clear upper 16 bits
(19)
(20) glue:   or    x10,x15,x10       # tack two results together
(21)
(22) done:   ret                      # take it on home

```

Figure 15.4: The solution to this example.

But of course, there is a better approach. We presented the previous solution to show an example of bit masking. The reality is that the shift left and shift right instructions stuff in 0’s to the register when they shift (the spec describes this characteristic). That means we do not need to use masks in this problem. This problem also provides an alternate solution to this example. Here are some comments.

- The code is shorter because we removed the instructions that initialized the masks, and the instructions that do the actual masking.
- We kept the labels, as they are forms of commenting: the make the code easier to understand but do not increase the code length.
- There is less code in this subroutine so it executes faster than the previous code, but provide the exact same result.

```

(00) #-----
(01) # Subroutine: Big_swap
(02) #
(03) # This subroutine swaps the upper two bytes with the lower two bytes
(04) # in register x20.
(05) #
(06) # Tweaked Registers: x10, x15
(07) #-----
(08) Big_swap:
(09)
(10) init:    mv    x15,x10        # make a copy
(11)
(12) upper:   slli  x15,x15,16     # move lower 2 bytes 16 bits to left
(13)
(14) lower:   srli  x10,x10,16     # move upper 2 bits 16 bits to right
(15)
(16) glue:    or    x10,x15,x10    # tack two results together
(17)
(18) done:    ret

```

Figure 15.5: An alternative solution to this example.

Example 15.4: Conditional Operations

Write a RISC-V assembly language subroutine that does the following based on the value in x20.

- If the value in x20 = 64
 - Divide value in x20 by 8
- If the value in x20 = 128
 - Divide the value in x20 by 32
- Otherwise, make x20=0

Solution Notes: Fun stuff embedded in the solution. This demonstrates a classic case construct, which of course is a special form of an if/else construct.

- The subroutine has a header that include pertinent information that other programmers who are reading the code can learn from, particularly the list of registers that the subroutine changes.
- The subroutine name is on a line by itself, which makes it clear to human readers; the subroutine name is a label that the assembler uses to transfer program control to the subroutine when it is called from another section of the program.
- Line (13) loads one of the values to compare into a register. The comparison is done on the next line. If the value in x20 is not equal to 64, the branch on line (14) sends the code to the next test, which is on line (18). If the value is equal to 64, the value is divided by 8 by shifting the value right by 3 using the **srli** instruction. Because the code found a match in the values, it's not going to find another match, so the program flow control jumps to the **ret** instruction.
- The same general approach is taken looking for the second match starting at the **check_128** label. Note that the code shifts right 5 places to divide by 32.
- The **check_64** and **check_128** labels and the **default** label for the cases for the case statement that this code uses. It smells like C code to me; maybe it's the same in less useful languages.

- The two blocks of code at starting at **check_64** and **check_128** labels are very similar. Quite often when you're writing code, you repeat the same functionality. This sort of means you can do it more generically, but it for sure means you can copy your code. Don't try this if you're a computer science major, because copying your own code is plagiarism and could get you expelled, killed, or worse.

```

(00) #-----
(01) # Subroutine: case_construct
(02) #
(03) # This subroutine is an example of a case statement. If the value in x20
(04) # is equal to 64, then the value is divided by 8. If the value in x20 is
(05) # equal to 128, then the value is divided by 32. If neither of those
(06) # tests are true, then the subroutine sets the value of x20 to 0. The
(07) # does the same for the other test.
(08) #
(09) # Tweaked Registers: x10, x20
(10) #-----
(11) case_construct:
(12)
(13) check_64:    li    x10,64           # load value to compare
(14)             bne   x10,x20,check_128 # check to see if equal
(15)             srli  x20,x20,3       # divide by 8 (2^3)
(16)             j     done           # done
(17)
(18) check_128:  li    x10,128          # load value to compare
(19)             bne   x10,x20,default # check to see if equal
(20)             srli  x20,x20,5       # divide by 32 (2^5)
(21)             j     done           # done
(22)
(23) default:    mv    x20,x0          # clear register
(24) done:      ret

```

Figure 15.6: A solution to this example problem.

15.3 More Advanced RISC-V Programming Problems

This section continues with more advanced programming problems. The previous sections provided a basis for many of the standard RISC-V MCU programming structures. We sincerely hope that after staring at these problems, you did not find them too complicated. Recall that the nice thing about assembly language programming is that nothing can really become too complicated based on the inherently simplistic nature of the assembly language programming.

Example 15.5: BCD to Binary Conversion

Write a RISC-V assembly language subroutine that converts a BCD to binary conversion on x10. Consider the three least significant nibbles in x10 to represent a 3-digit decimal number. Place converted binary number in x20. Note that a nibble is half a byte, or 4-bits. Recall that we use BCD to represent decimal numbers, which requires a minimum of 4 bits

Solution Notes: Fun stuff embedded in the solution. This is a very common and useful conversion. There are cleaner ways to do this, but this is good enough for now. This is the most meaningful and useful program up until now, but it is still “missing” some stuff; we’ll talk about the missing stuff later.

- Once again, nice header providing useful information regarding the subroutine.
- This is a non-trivial subroutine, so we need to do some initialization. We use the **init** label to note the code that falls into the category of initialization.

- We first need to save a copy of the data we need to convert, which we do with the `mv` pseudoinstruction on line (11). We often refer to this as making a “working copy” of the data. This subroutine is going to tweak the data, so we need to ensure that we don’t lose the original data before we’re done with it.
- We stuff our nibble masks in registers starting at line (12). Yes, definitely better ways to do this in case you’re thinking this is klunky. We’ll definitely become cleverer with our coding once we get more assembly language programming skills in our bag of tricks.
- This is a classic “accumulator” problem. As with most accumulator problems, we need to start the accumulator at 0, which is what the `mv` instruction on line (15) does.
- The BCD values provide the count of the number of 100’s, 10’s, and 1’s. The general approach of this code is to keep adding one of those values for each value in the nibble location. This means there is a loop to accumulate 100’s, followed by a loop to calculate 10’s, and then we simply add the 1’s as the 1’s has no associated weighting as the 100’s and 10’s does.
- The 100’s loop starts at line (17) where we see a label which is there for commenting purposes (no code ever jumps to it. We first mask the bits we’re interested in with the `and` instruction on line (17). We then shift the resulting 100’s nibble to the four LSB positions of the register. At that point, the code on lines (19-22) form a while loop that adds 100 to the accumulator each iteration. The BCD value is effectively the loop count, so we must decrement it each iteration. We then jump to the comparison instruction on line (19). In this case, we do use the `loop1` label as the place to `j` instruction on line (22) passes program control to.
- The code for the 10’s loop is similar to the code for the 100’s loop, so we’ll skip the painful detail. The one interesting thing to note is that the first thing we need to do is restore the original value on line (24), which we originally saved on line (11).
- The 100’s and 10’s BCD nibbles have weights associated with the counts, but the 1’s nibble does not. This being the case, we don’t require a loop as we did with the 100’s and 10’s, we simply add the 1’s nibble to the accumulator. We do have to mask the higher-order nibble before we accumulate the 1’s, which we do on line (33).

```

(00) #-----
(01) # Subroutine: bcd_to_bin
(02) #
(03) # Converts a 3-digit decimal number represented in the lowest three nibbles
(04) # of x10 to the equivalent unsigned binary value and places the result
(05) # in x20.
(06) #
(07) # Tweaked Registers: x15, x20, x21, x22, x23
(08) #-----
(09) bcd_to_bin:
(10)
(11) init:      mv      x15,x10      # save a copy
(12)          li      x21,0x00000F00 # 100's bit mask
(13)          li      x22,0x000000F0 # 10's bit mask
(14)          li      x23,0x0000000F # 1's bit mask
(15)          mv      x20,x0        # zero accumulator
(16)
(17) t_100:    and     x15,x15,x21   # mask 100's nibble
(18)          srli   x15,x15,8     # shift to lowest position
(19) loop1:    beqz   x15,t_10      # go to tens if zero
(20)          addi   x20,x20,100    # accumulate 100s
(21)          addi   x15,x15,-1    # decrement loop count
(22)          j      loop1        # do it again
(23)
(24) t_10:     mv      x15,x10      # restore original value
(25)          and     x15,x15,x22   # mask 10's nibble
(26)          srli   x15,x15,4     # shift left to right-most position
(27) loop2:    beqz   x15,t_1      # move on if it's zero
(28)          addi   x20,x20,10    # accumulate 10 values
(29)          addi   x15,x15,-1    # decrement loop count
(30)          j      loop2        # do it again
(31)
(32) t_1:      mv      x15,x10      # get original value
(33)          and     x15,x15,x23   # mask bits
(34)          add     x20,x20,x15   # add value to accumulator
(35)
(36) done:     ret

```

Figure 15.7: A solution to this example problem.

Figure 15.8 provides an alternate solution for this example. What we did was use one register for the mask value rather than three registers. The length of the code is the same and it has the same running time, but the subroutine is more “space efficient” because it uses less registers. This becomes a more important issue when we start writing our subroutines “more better”. More on that later.

- Lines (23) and (32) shift the single mask right by four bits. This creates the same mask as before but it uses less registers. Pretty clever. I wish I had thought of that the first time.


```

(00) #-----
(01) # Subroutine: bcd_to_bin
(02) #
(03) # Converts a 3-digit decimal number represented in the lowest three nibbles
(04) # of x10 to the equivalent unsigned binary value and places the result
(05) # in x20.
(06) #
(07) # Tweaked Registers: x15, x20, x21
(08) #-----
(09) bcd_to_bin:
(10)
(11) init:    mv    x15,x10      # save a copy
(12)         li    x21,0x00000F00 # 100's bit mask
(13)         mv    x20,x0      # zero accumulator
(14)
(15) t_100:   and    x15,x15,x21 # mask 100's nibble
(16)         srli  x15,x15,8    # shift to lowest position
(17) loop1:   beqz  x15,t_10     # go to tens if zero
(18)         addi  x20,x20,100   # accumulate 100s
(19)         addi  x15,x15,-1    # decrement loop count
(20)         j     loop1        # do it again
(21)
(22) t_10:    mv    x15,x10     # restore original value
(23)         srli  x21,x21,4    # shift the mask value to next nibble
(24)         and   x15,x15,x21  # mask 10's nibble
(25)         srli  x15,x15,4    # shift left to right-most position
(26) loop2:   beqz  x15,t_1     # move on if it's zero
(27)         addi  x20,x20,10   # accumulate 10 values
(28)         addi  x15,x15,-1   # decrement loop count
(29)         j     loop2        # do it again
(30)
(31) t_1:     mv    x15,x10     # get original value
(32)         srli  x21,x21,4    # shift the mask value to next nibble
(33)         and   x15,x15,x21  # mask bits
(34)         add   x20,x20,x15  # add value to accumulator
(35)
(36) done:    ret                    # take it on home

```

Figure 15.8: An alternate solution to this example problem.

Example 15.6: Parity Determination

Write a RISC-V assembly language subroutine that determines the parity of the value in x10. If x10 has even parity, it returns a '0' in x20. Otherwise, it returns a '1'.

Solution Notes: Fun stuff embedded in the solution. This is another handy function.

The overall algorithm is this: mask LSB, increment count with LSB, shift right original value until the original value is zero. The value in the LSB of the count is the desired parity value returned to the calling code. There are many approaches to calculating parity, this is one of the easier.

- Subroutine initialization is only a matter of clearing a register that the subroutine uses as an accumulator, which occurs on line (10).
- The main body of the loop is a while loop. The while loop counts the number of bits that are set in x10. We implement the while loop as a loop with an unknown number of iterations; the idea here is that we'll keep counting bits in x10 so long as x10 is non-zero. We could have modeled this as a loop with a known count (namely, 32), but the way we modeled it ensures it will run faster in the average case. Note that this approach only works because we know the RISC-V shifting operations shift '0's into the register that is being shifted.

- Line (13) masks the LSB of x10 and stores the result in x15. Line (14) uses the result of the masking operation to increment a count variable stored in x20.
- Line (15) adjust the original value by shifting it to the right by one bit, then jumps to the check instruction on line (12).
- The done label on line (18) clears the upper 31 bits, thus leaving the required parity bit in x20.

```

(00) #-----
(01) # Subroutine: parity
(02) #
(03) # Determines the parity of the value in x10; returns '0' in x20 if parity
(04) # is even, otherwise returns '1'.
(05) #
(06) # Tweaked Registers: x10, x15, x20
(07) #-----
(08) parity:
(09)
(10) init:   mv    x15,x0        # clear an accumulator
(11)
(12) loop:  beq   x10,x0,done    # check to see if were done
(13)        andi  x15,x10,1     # mask LSB
(14)        add   x20,x20,x15   # increment bit count
(15)        srli  x10,x10,1     # shift value right 1 bit
(16)        j     loop         # rinse, repeat
(17)
(18) done:  andi  x20,x20,1     # clear all but LSB
(19)        ret   x20          # take it home jimie

```

Figure 15.9: A solution to this example problem.

Example 15.7: Rotate Left Implementation

Write a RISC-V assembly language subroutine that rotates the value in x10 left by the value in x11. Assume the value in x11 is going to be [0,32].

Solution Notes: Fun stuff embedded in the solution. This is another handy function. The RISC-V ISA currently does not include a rotate instruction. To combat this injustice, we need to implement a rotate in code. Yes, very clever algorithm. It seems to work despite its admitted cleverness.

- The general approach of the subroutine is to use a shift left to do most of the rotate, but to catch the bits that we normally shift off into a register that we later add back at the other end of the original data.
- The instruction on line (10) places the length of the RISC-V registers into a register. We'll use this value to find out how many bits we need to shift in the right direction. The actual calculation is done on line (11). The result of this subtraction essentially finds the complement of the value to shift based on 32. The result is the number of bits to shift in the other direction.
- We need to operate on two different registers, for we make a working copy on line (12).
- Line (14) left shifts off the number of bits in the register passed to the subroutine (x12). We already saved the original number in x30, so we shift the value that number in the other direction, leaving the bits that were shifted off in line (14) in the lower x11 bits of x30.
- We have both the values remaining in the sent value after the shift (x10) and the bits we shifted off at the lower end of x30. We then complete the subroutine by gluing these values together with the `or` instruction on line (17).

```

(00) #-----
(01) # Subroutine: rot_left
(02) #
(03) # This routine performs a rotate left on the value in x10 by rotating
(04) # x10 by the value provided x11. The value in x11 must be [0,32].
(05) #
(06) # Tweaked Registers: x25, x30, x10
(07) #-----
(08) rot_left:
(09)
(10) init:    li    x25,32      # 32 is the length of registers
(11)         sub   x25,x25,x11 # get the complement of 32
(12)         mv    x30,x10    # copy x10 to working register
(13)
(14)         sll   x10,x10,x11 # shift the lower bits left
(15)         srl   x30,x30,x25 # shift the upper bits right
(16)
(17)         or    x10,x30,x10 # glue the results together
(18)
(19) done:    ret                # go home, all the way home

```

Figure 15.10: A solution to this example problem.

Example 15.8: Classic LED Bouncer

Write a RISC-V assembly language program lights one LED at a time. The program makes it appear as if the lighted LED moves left through 16 LEDs, the back to the right, then back to the left, etc. Assume there are 16 LEDs and the output port address of those 16 LEDs is 0xC0000080.

Solution Notes: Fun stuff embedded in the solution. This is a fun problem: it's the bouncing LED problem, which has some classic programming constructs. Many approaches to doing this problem. Be sure to check out the solution on the simulator of your choice.

- On lines (08) – (10), we place the output port address in a register so we can later use a store instruction for output of data to the LEDs. We output 32 bits to 16 LEDs, for all but one of the LEDs is off. We use x10 for the one LED that is on so we initialize it to '1' (only right-most LED on). We finish up with initializing register x15 to 15, which we later use to discern when we're done with one left-to-right or right-to-left cycle. Note that we use the `init` label to indicate that chunk of code is initialization code.
- The algorithm officially starts on line (12) where we initialize a local loop count to 0. This is truly an initialization, but we place the code near one of the loops because we jump back to this instruction after we finish a right-to-left and a left-to-right cycle.
- Line (13) shows the first output; we output using a `sw` (store word) instruction. The data we output is the LED value; we output it to the output port address.
- After the output, we do some administrative tasks. We first shift the register holding the LED value to the left on line (14). We then increment the loop count on line (15). The final loop administration operation is to jump if the count is not equal to the maximum shift value, which we do on line (16).
- If the branch on line 16 is not taken, then we drop through to the other block of code, which shifts the lit LED from the left to the right. We start this process on line (18) by resetting the loop counter.

- The second loop is similar to the first loop except we are shift right one bit at a time. When program controls falls through the second loop on line (22), we jump back to initialize the first loop.
- Note that the overall form of the program is an initialization followed by two do-while loops.

```

(00) #-----
(01) # Program: led bouncer
(02) #
(03) # The program moves a single LED back and forth on an assumed 16 LED
(04) # display. The LED output port is 0xC0000080
(05) #
(06) # Tweaked Registers: x10, x15, x16, x30
(07) #-----
(08) init:  li   x30,0xC0000080  # put output port address in register
(09)        li   x10,1          # initialize LED bounce register
(10)        li   x15,15         # x15 used as counter register
(11)
(12) loop:  mv   x16,x0          # clear counter register
(13) left:  sw   x10,0(x30)      # output current LED value
(14)        slli x10,x10,1      # shift LED left one bit position
(15)        addi x16,x16,1      # increment bounce count
(16)        bne x16,x15,left    # check loop (goes 15) times
(17)
(18)        mv   x16,x0          # init bounce count
(19) right: sw   x10,0(x30)      # output current LED value
(20)        srli x10,x10,1      # shift right one bit position
(21)        addi x16,x16,1      # increment loop count
(22)        bne x16,x15,right    # branch to inner loop if loop not 15
(23)        j    loop           # done with right, go to left

```

Figure 15.11: A solution to this example problem.

Example 15.9: Conditional LED Display

Write a RISC-V assembly language program lights monitors the 16 switches connected to the RISC-V MCU, such as the ones on the development board. Consider the 5 right-most switches to form a 5-bit digital number. The program outputs to the LEDs continuously according to the following:

- Switches = 0: turn on right-most four LEDs
- Switches = 1: turn on the second to right-most four LEDs
- Switches = 2: turn on the second to left-most four LEDs
- Switches = 3: turn on left-most four LEDs
- Otherwise, turn off all LEDs

Consider the address of the switch input port and LED output port to be 0xC0000040 and 0xC000008, respectively.

Solution Notes: Fun stuff embedded in the solution. This is a classic case statement problem, that does not do too much exciting as do some of the previous problems up to this point.

- On lines (12) – (15), we do a bunch of initialization. First we place the port addresses registers. Then we initialize a mask for input data, which clears all but the first three bits. Lastly, we clear x20, which we later use as a count register.

- The code starting at `main` is essentially more initialization code, but it inits each iteration, which means it reinitializes items that were first initialized in the `init` code, but the values were changed by the body of the program.
- Line (20-21) inputs the data and masks all but the three LSBs. Recall that we only need to look for 0-3 for this case statement.
- The `chk_x` labels delineate the separate parts of the program where we're looking for values 0-3. These sections of code form the cases we're looking for. The output instruction (`sw`) on line (43) represents the default condition where we turn off all LEDs.
- We use `bne` instructions to compare the input to the desired value. We compare the input to a counter that we increment in each `chk_x` checking section of code.
- We need to output a 4-bit chunk of LEDs that are on, so we keep a register with the desired output value. With each checking section, we use the `slli` instruction to shift that chunk of data to the correct position for outputting.

```

(00) #-----
(01) # Program: switch input monitor
(02) #
(03) # The program moves monitors the switches associated with the system. There
(04) # are 16 switches, which the program reads and then outs a value to the LEDs
(05) # according to: value 0, 1, 2, 3, none; the corresponding outputs are
(06) # a set of four LEDs starting from the right and moving to the left. The c
(07) # default value is all LEDs off.
(08) #
(09) # Tweaked Registers: x15, x16, x20, x30, x31
(10) #-----
(11)
(12) init:      li      x30,0xc0000040 # put switch input port address in reg
(13)          li      x31,0xc0000080 # put LED output port address in reg
(14)          li      x15,0x00000007 # mask for switches
(15)          mv      x20,x0        # clear counter register for input compares
(16)
(17)
(18) main:      li      x16,0x0000000F # start value to output
(19)          mv      x20,x0        # clear counter register for input compares
(20)          lw      x10,0(x30)     # get input data
(21)          and     x10,x10,x15    # mask off lower 3 bits
(22)
(23) chk_0:     bne     x10,x20,chk_1 # check input for 0
(24)          sw      x16,0(x31)     # output to LEDS
(25)          j       main
(26)
(27) chk_1:     addi    x20,x20,1     # increment check count
(28)          bne     x10,x20,chk_2 # check input for 1
(29)          slli    x16,x16,4     # value to output
(30)          sw      x16,0(x31)     # output to LEDS
(31)          j       main
(32)
(33) chk_2:     addi    x20,x20,1     # increment check count
(34)          bne     x10,x20,chk_3 # check input for 2
(35)          slli    x16,x16,8     # value to output
(36)          sw      x16,0(x31)     # output to LEDS
(37)          j       main
(38)
(39) chk_3:     addi    x20,x20,1     # increment check count
(40)          bne     x10,x20,default # check input for 3
(41)          slli    x16,x16,12    # value to output
(42)          sw      x16,0(x31)     # output to LEDS
(43)          j       main
(44)
(45) default:  sw      x0,0(x31)
(46)          j       main

```

Figure 15.12: A solution to this example problem.

As with all code you write, there’s probably a “better” way to write the code. Keep in mind that the notion of “better” has many definitions. The thing to note for this solution is that the various case clauses look very similar. Anytime you see this in the code, you can often time structure you code to use loop constructs rather than the straight through code in the original solution. The code in Figure 15.13 is an attempt to structure the code to be more space efficient. This solution obviously requires less instructions, but... it requires more register (one more). The funny thing in programming is that there are always tradeoffs. The other trade off in the solution of Figure 15.13 is that it runs a bit slower than the first solution, which is because there is always some overhead associated with loops, which means there are more instructions that say: “go somewhere” rather than say: “do something”.

```

(00) #-----
(01) # Program: switch input monitor
(02) #
(03) # The program moves monitors the switches associated with the system. There
(04) # are 16 switches, which the program reads and then outs a value to the LEDs
(05) # according to: value 0, 1, 2, 3, none; the corresponding outputs are
(06) # a set of four LEDs starting from the right and moving to the left. The c
(07) # default value is all LEDs off.
(08) #
(09) # Tweaked Registers: x15, x16, x20, x21, x30, x31
(10) #-----
(11)
(12) init:      li      x30,0xc0000040 # put switch output port address in register
(13)          li      x31,0xc0000080 # put LED output port address in register
(14)          li      x15,0x00000007 # mask for switches
(15)          mv      x20,x0        # clear counter register for input compares
(16)          li      x21,4         # the end count
(17)
(18)
(19) main:      li      x16,0x0000000F # start value to output
(20)          mv      x20,x0        # clear counter register for input compares
(21)          lw      x10,0(x30)    # get input data
(22)          and     x10,x10,x15   # mask off lower 3 bits
(23)
(24) check:    bne     x10,x20,not_eq # check input for 0
(25)          sw      x16,0(x31)    # output to LEDS
(26)          j       main
(27)
(28) not_eq:    addi    x20,x20,1    # increment compare count
(29)          beq     x20,x21,default # do default if count is 4
(30)          slli   x16,x16,4      # count != 4, shift output val
(31)          j       check        # look for next number
(32)
(33) default:  sw      x0,0(x31)
(34)          j       main

```

Figure 15.13: A solution to this example problem.

Example 15.10: Memory Data Swap

Write a RISC-V assembly language subroutine that swaps the data in two memory locations. Consider the data to be words (4-bytes) that reside at the addresses given by the values in x20 and x21.

Solution Notes: Fun stuff embedded in the solution. Note that this is a classic case of using what intelligent & useful higher-level languages such as C refer to as pointers. This provides classic genericity when work with data sets that you can’t obtain from working with registers alone.

- After the great subroutine banner, we include an init label in the program. We do this mostly out of habit, as you could argue that we really don't need to initialize anything.
- Lines (11-12) show the loading of data from the addresses given in the x20 and x21 registers into two working registers x30 & x31.
- Lines (14-16) show the classic XOR in-place swap trick, showcasing the magic of the XOR function.
- Lines (18-19) uses the address still store in x20 & x21 to store the data back in memory.

Post Mortem: note that the general structure of the program was to transfer something from memory to registers, tweak with the value in registers, then transfer the values back to memory. The idea here is that we the only operations we can do with memory is loading and storing; all the interesting bit crunching takes place using registers.

```

(00) #-----
(01) # Subroutine: mem_word_swap
(02) #
(03) # Swap data in the memory location specified by the contents of
(04) # registers x20 & x21 (x20 & x21 thus contain memory address values).
(05) #
(06) # Tweaked registers: x30,x31
(07) #-----
(08) .text
(09)
(10) mem_word_swap:
(11) init:    lw    x30,0(x20)      # get data from memory
(12)         lw    x31,0(x21)
(13)
(14)         xor   x30,x30,x31     # the classic xor in-place data swap
(15)         xor   x31,x31,x30
(16)         xor   x30,x30,x31
(17)
(18)         sw    x30,0(x20)     # store the data back at the addresses
(19)         sw    x31,0(x21)     # the data was obtained from
(20)
(21)         ret                    # take on home

```

Figure 15.14: Solution to this example problem.

Example 15.11: Memory Data Averaging

Write a RISC-V assembly language subroutine that calculates the average of 32 values (words, so 4 bytes) in memory. The starting address of the data is passed to the subroutine in register x10; pass the average back to the calling routine by placing the calculated average in x20.

Solution Notes: Fun stuff embedded in the solution. All we know about this problem is the starting address of the data to average, and the number of items to average. Sending data to the subroutine is often referred to as passing data to the subroutine; returning data from the subroutine is often referred to as returning data from the subroutine. Exciting stuff.

- This is a subroutine that requires accumulating values, which means we first must clear the accumulator. We use x20 as the accumulator and clear it on line (09).
- This is a problem where we iterate a given number of times (32), so we use x15 to hold that count and initialize x15 on line (10).

- Although the problem does not state it, we've decided to save the first address value in x10 by copying it to another register, which we do on line (11).
- The main loop starts on line (13) with a conditional branch (**beq**). We're modeling this solution using a while loop; since we know we'll always be adding 32 items; we could have easily used a do-while loop for this solution.
- We first get the data from memory using a lw instruction on line (14), we then accumulate the loaded value on line (15). These two lines form the body of the loop; all the other stuff in the loop is what we refer to as loop administration.
- For loop administration, we first decrement the loop count on line (16), then advance the address of data we're loading from memory on line (17). The next line of loop admin is the unconditional branch on line (18). The final line of loop admin is the conditional branch on line (13).
- When the branch condition evaluates are true, we branch to the instruction associated with the done label. This **srli** instruction performs the divide by 32. Very handy; we must be thankful that the person who created this problem made the divide easy, as a divide by 32 is simply a barrel shift right five bit locations.
- The restore label is used to indicate we're restoring some registers to the values they had when the subroutine started. There are better ways to do this, but this works for now. Please don't tell Jeffrey.
- Note that since we "restored" the original value of x10, we don't include it in the tweaked register list.

```

(00) #-----
(01) # Subroutine: avg_32
(02) #
(03) # Averages 32 words in memory starting at the address in x10. The
(04) # result is stored in x20.
(05) #
(06) # Tweaked registers: x20,x15,x16,x11
(07) #-----
(08) avg_32:
(09) init:      mv    x20,x0          # clear accumulator
(10)          li    x15,32         # number to sum
(11)          mv    x16,x10        # copy original address
(12)
(13) loop:     beq   x15,x0,done    # leave if finished
(14)          lw    x11,0(x10)     # get value from memory
(15)          add  x20,x20,x11     # accumulate
(16)          addi x15,x15,-1     # decrement loop count
(17)          addi x10,x10,4      # advance addr to next data
(18)          j    loop           # done with iteration, do again
(19)
(20) done:     srli  x20,x20,5     # divide by 32
(21) restore:  mv    x10,x16      # restore original x10 address
(22)
(23)          ret                    # come on up to the house

```

Figure 15.15: Solution to this example problem.

Example 15.12: Fibonacci Sequence Generator

Write a RISC-V assembly language subroutine that generates the first 16 Fibonacci numbers (starting with 1,1) and stores those values as unsigned halfwords starting at address x25 in memory. Don't allow this subroutine to permanently change any register value.

Solution Notes: Fun stuff embedded in the solution. We all know how much students love solving problems having to do with Fibonacci numbers. So here's the solution in RISC-V assembly language.

- I've of course plopped the solution down; the truth is that I first generated a flowchart before I wrote the code. After that, I wrote the code in two main phases. I first wrote code to solve the problem. I then added the code I that saved the context of the MCU when the subroutine was called. This is essentially a fancy way of saying the subroutine saved all the register that are changed in the body of the subroutine on the stack before executing doing anything having to do with the Fibonacci sequence. Moreover, once we completed what the problem was asking for, we restored the registers we used in the subroutine back to their original values.
- The first part of any subroutine is the initialization, which we casually label with "init". For most of these problems, initialization includes several phases. First we save any registers we're using in the subroutine. Then we save the return address **ra** (x1) if the subroutine calls other subroutines. Then we initialize important things in the code such as loop counters, accumulators, etc. This subroutine doesn't call another subroutine so we don't have to save the **ra** register.
- The subroutine uses five registers, so we make room on the stack so we can safely add these registers to the stack. We back the **sp** up 20 bytes so that we can store 5 registers (recall that each register comprises of four bytes) on line (10). We then proceed to save the five registers (in no particular order) onto the stack, which we do on lines (11-15).
- The next part of the initialization is to put the loop count into a register, which we do on line (17). We only place the count at 14 even though we intend to generate of Fibonacci sequence comprising of 16 values because we hardcode the first two values in the sequence.
- The first two numbers in the Fibonacci sequence are one, so we opt to place '1' in a register for easy later access. This makes sense because we need the value we want to write to memory to be in a register.
- We store the first two values in the Fibonacci sequence on lines (20-21). We follow that by adjusting the address pointer **x25** by 4, which represents to halfwords We could have done these three instructions in different orders using different offsets; there's nothing magical about the approach I took in these lines.
- After we've initialized the first two values in the Fibonacci sequence, we're ready to enter the "algorithmic" portion of the subroutine. This is because unlike the first two values in the sequence that we assigned, we're not ready to calculate the remaining Fibonacci values from previous Fibonacci values. We of course do this all this in a loop, and we find it easiest to use a while loop. We first check to see if we're done with the loop starting on line (24).
- The first part of the algorithm requires up to get the last two values in the sequence from memory, which we do on lines (25-26) we grab unsigned halfwords (**lhu**) as the problem specifies. The address was already updated, so we use the offset of these two instructions to reach back before the current value of the address (we use negative offsets to reach back).
- Once we have the two previous values in the Fibonacci sequence, we then add these values to form the next value in the sequence on line (27) and store the result in memory on line (28) using the **sh** instruction.
- At this point, we're done with the body of the loop and we need to perform some loop administration, which include decrementing the loop count on line (29) and then advancing the

address value we're keeping in x25 by two. Note that we add two because we're working with halfwords.

- After we've complete the loop administration we branch unconditionally back to the start of the loop.
- When we eventually complete the loop, we must restore the registers we're using in the subroutine to the values they were at before the subroutine changed them. We do this with five **lw** instructions starting on line (34). Once we've restored all the registers, we then adjust the stack point by 20, which essentially undoes the operation on line (10).

```

(00) #-----
(01) # Subroutine: gen_fib_16
(02) #
(03) # Generates the first 16 Fibonacci numbers (starting with 1,1,...) and
(04) # stores the numbers as halfwords in memory starting at the address
(05) # stored in x25.
(06) #
(07) # Tweaked registers: none
(08) #-----
(09) gen_fib_16:
(10) store:  addi  sp,sp,-20    # adjust sp to save 5 regs
(11)         sw   x20,0(sp)    # save x20
(12)         sw   x21,4(sp)    # save x21
(13)         sw   x25,8(sp)    # save x25
(14)         sw   x15,12(sp)   # save x15
(15)         sw   x16,16(sp)   # save x16
(16)
(17) init:   li    x20,14      # load loop count
(18)         li    x21,1       # load initial fib number
(19)
(20)         sh   x21,0(x25)    # store first two fib numbers
(21)         sh   x21,2(x25)
(22)         addi x25,x25,4    # adjust the pointer forward
(23)
(24) loop:   beq   x20,x0,done  # done yet?
(25)         lhu  x15,-4(x25)   # get two previous values
(26)         lhu  x16,-2(x25)
(27)         add  x15,x15,x16   # add two previous value
(28)         sh   x15,0(x25)   # store result of addition
(29)         addi x20,x20,-1    # loop admin: decrement loop count
(30)         addi x25,x25,2    # increment pointer forward
(31)         j    loop        # repeat, rinse
(32)
(33) restore:
(34)         lw   x20,0(sp)    # restore x20
(35)         lw   x21,4(sp)    # restore x21
(36)         lw   x25,8(sp)    # restore x25
(37)         lw   x15,12(sp)   # restore x15
(38)         lw   x16,16(sp)   # restore x16
(39)         addi sp,sp,20     # adjust sp to back to original value
(40)
(41) done:   ret

```

Figure 15.16: Solution to this example problem.

Example 15.13: Largest Value Finder

Write a RISC-V assembly language subroutine that finds the largest value in the five unsigned bytes starting at the address stored in x10. Return the largest value to the calling code in x10. Don't allow the subroutine to permanently change any registers other than x10.

Solution Notes: Fun stuff embedded in the solution. Yet another problem that involves generic access to memory. There are many ways structure problems such as this one; we always choose the most generic approach. For this problem, that means that we want to make the algorithm in one phase rather than two. For problems such as this, it's always tempting to compare the first two values, then compare all the result to that first result. Yep, it works, but it's easier to keep in simple by making it one phase only.

- Keep in mind, I first wrote the body of this loop, then went back and saved/restored context. This is of course because I don't know what registers need saving until I'm done doing the required work.
- Saving/restoring context only included registers; since this subroutine did not call other subroutines, there was no reason to include **ra** in context saving/restoring. This subroutine used three registers, so we saved 12 bytes of space on the stack on line (10). We then stored the registers in no specific order and at no specific addresses.
- The code starting at the **init** label was for the loop. We need to check five value so we initialized a register on line (15). We then wanted to keep our algorithm generic so we initialized a register with zero on line (16). Note that zero is the smallest possible value, so the MCU stores any value greater than that as the largest value in the loop's iterations.
- The body of the loop first loads some data from memory on line (18); the problem stated unsigned bytes so we use the **lbu** instruction. We then compare the loaded data with our current largest value on line (19). If we find a new large value, we replace the current larger value on line (21).
- Whether the branch is taken on line (19) or not, we always perform loop administration. For this algorithm, that include incrementing the address value on line (23) and decrementing the loop counter on line (24). Note that we only increase the address by one because in this problem we are dealing with bytes.
- We model this loop as a do-while loop because we know that we'll always have to go through the loop five times, which of course means we'll always have to do it once. There are many ways to do this but this is probably the most efficient.
- When we drop out of the loop, we know that x20 has the largest value. We then need to put that value in x10 as requested by the original problem, which we do on line (27).
- Context restoration is performed on lines (29-31). We adjust the stack pointer (**sp**) on line (32). Note that the amount we adjust the stack pointer on line (32) is the opposite of how much we adjusted the stack pointer on line (10).

```

(00) #-----
(01) # Subroutine: find_big_5
(02) #
(03) # This subroutine finds the largest of 5 continuous unsigned bytes in
(04) # memory starting at the address store in x10.
(05) #
(06) #
(07) # Tweaked registers: x10
(08) #-----
(09) find_big_5:
(10) store:   addi sp,sp,-12      # room for 3 regs
(11)         sw  x15,0(sp)      # push regs
(12)         sw  x20,4(sp)
(13)         sw  x21,8(sp)
(14)
(15) init:   li   x15,5
(16)         mv  x20,x0        # smallest possible
(17)
(18) loop:   lbu  x21,0(x10)    # get data
(19)         blt x21,x20,Admin # jump if x16>x17
(20)
(21)         mv  x20,x21      # store new large
(22)
(23) Admin:  addi x10,x10,1    # incr addr
(24)         addi x15,x15,-1  # decr loop count
(25)         bne x15,x0,loop  # do it again
(26)
(27) xfer:   mv   x10,x20     # x15 is largest
(28)
(29) restore: lw  x15,0(sp)   # pop regs
(30)         lw  x20,4(sp)
(31)         lw  x21,8(sp)
(32)         addi sp,sp,12   # readjust stack pointer
(33)
(34) done:   ret              # bring it home

```

Figure 15.17: Solution to this example problem.

When I write these problems, I always test them first on one of the RISC-V simulators. To test problems that include memory, I have to first put the data in memory. I do this by using the `.byte` directive in the data segment. In general, we put data in the data segment and code in the `.text` segment. This means that you have to use the `.data` and `.text` directives to have the data placed in the correct places. I included the full program using both segments in the following figure. Here is some stuff to chew on.

- Line (09) has the `.data` segment directive. From there we can now specify some data.
- Line (10) has some data, which is conveniently give pieces of data. We specify this data as being bytes by using the `.byte` directive, which means the data we specify is stored as bytes in memory.
- We gave the data a label on line (10) also. The issue here is that the assembler can place the data anywhere in data memory, we don't yet have easy control of that. We use a label here because we can figure out where that data actually is by using the `la` instruction on line (13). This instruction stands for “load address” and is one of our load instructions that don't actually have anything to do with memory. What this instruction does is put the value associated with the “junk” label into register x10. The key to understanding this is the value associated with the junk label is the address in memory where the “2” is stored. The value of 45 is one byte beyond where the “2” is stored.
- Before we start writing actual code, we first must change from the data segment to the text segment. We do that by using the `.text` directive on line (12). If we did not do this, the assembler would grumble, and no one likes a grumbling assembler.
- The remainder of the algorithm does not change so we'll not bore you again with the details.

```

(00) #-----
(01) # Subroutine: find_big_5
(02) #
(03) # This subroutine finds the largest of 5 continuous unsigned bytes in
(04) # memory starting at the address store in x10.
(05) #
(06) #
(07) # Tweaked registers: x10
(08) #-----
(09) .data                                # declare data segment
(10) junk: .byte 2,45,4,5,6                # make up some data
(11)
(12) .text
(13)     la    x10,junk                    # load address of data
(14)
(15) Find_Big_5:
(16) store: addi sp,sp,-12                 # room for 3 regs
(17)         sw  x15,0(sp)                 # push regs
(18)         sw  x20,4(sp)
(19)         sw  x21,8(sp)
(20)
(21) init:  li   x15,5
(22)         mv  x20,x0                    # smallest possible
(23)
(24) loop:  lbu  x21,0(x10)                # get data
(25)         blt x21,x20,Admin            # jump if we don't find new large
(26)
(27)         mv  x20,x21                    # store new large
(28)
(29) Admin: addi x10,x10,1                 # incr addr
(30)         addi x15,x15,-1                # decr loop count
(31)         bne x15,x0,loop                # do it again
(32)
(33) xfer:  mv   x10,x20                    # x15 is largest
(34)
(35) restore: lw  x15,0(sp)                 # pop regs
(36)         lw  x20,4(sp)
(37)         lw  x21,8(sp)
(38)         addi sp,sp,12                 # readjust stack pointer
(39)
(40) done:  ret                             # bring it home

```

Figure 15.18: Solution to this example problem.

Example 15.14: Largest Value Finder

Write a RISC-V assembly language subroutine that finds the largest value in a set of unsigned halfwords. The starting address of the data is passed to the subroutine in x10; the length of the data is passed to the subroutine in x11. Return the largest value to the calling code in x15. Don't allow the subroutine to permanently change any registers other than x15.

Solution Notes: Fun stuff embedded in the solution. This problem is similar to the previous problem, so we won't include another painful description. The issue with the previous problem was that it was not generic; although the subroutine could check data anywhere, it was hardcoded to inspecting five pieces of data. The subroutine in this problem differs in that the calling code passes the number of values to check. You'll be sure to note that there are not big changes from the previous problem.

- Since we're passing a new value to the subroutine, which is the count value in a register, we need to save that register on the stack. We make room for four register pushes on line (11), and save the registers on the four following lines (12-15)
- We don't need to initialize the loop count since that value is passed to the subroutine in x11. We do decrement the loop count as part of loop administration on line (25).
- Part of loop administration is incrementing the address of the data. Since we're working with halfwords in this subroutine, we increment the address by two each loop iteration; see line (24).
- Note that this subroutine uses eight labels. Only three of the labels (find_big_uhalf, loop, and Admin) are actually required by the code. The unused labels make the code more readable to humans but does not increase the storage space requirements of the program.

```

(00) #-----
(01) # Subroutine: find_big_uhalf
(02) #
(03) # This subroutine finds the largest of continuous unsigned halfwords
(04) # starting at the data memory address in x10. The number of halfwords
(05) # the subroutine inspects is passed in x11. The largest value is
(06) # passed back to the subroutine in x15.
(07) #
(08) # Tweaked registers: x15
(09) #-----
(10) find_big_uhalf:
(11) store:  addi sp,sp,-16      # room for 4 regs
(12)         sw  x15,0(sp)      # push 4 regs
(13)         sw  x20,4(sp)
(14)         sw  x21,8(sp)
(15)         sw  x11,12(sp)
(16)
(17) init:   mv   x20,x0        # smallest possible
(18)
(19) loop:   lbu  x21,0(x10)     # get data
(20)         blt  x21,x20,Admin  # jump if we don't find new large
(21)
(22)         mv   x20,x21        # store new large
(23)
(24) Admin:  addi x10,x10,2      # incr addr by halfword #bytes
(25)         addi x11,x11,-1     # decr loop count
(26)         bne  x15,x0,loop    # do it again
(27)
(28) xfer:   mv   x10,x20        # x15 is largest
(29)
(30) restore: lw  x15,0(sp)      # pop four pushed regs
(31)         lw  x20,4(sp)
(32)         lw  x21,8(sp)
(33)         lw  x11,12(sp)
(34)         addi sp,sp,16      # readjust stack pointer
(35)
(36) done:   ret                  # bring it home

```

Figure 15.19: Solution to this example problem.

Example 15.15: Memory Data Size Conversion

Write a RISC-V assembly language subroutine that reads unsigned bytes of data starting at the address in x10, and stores that data as equivalent values in words starting at the address in x20. Register x11 holds the number of data pieces to translate. Don't allow the subroutine to permanently change any register.

Solution Notes: Fun stuff embedded in the solution. This is another generic subroutine that translates data at one address in memory to another address in memory. Note that it is specific to what size of the data it translates to and from. This lack of genericity is the inspiration for a later problem.

- The body of the subroutine uses three registers, so we make room to save those three registers by adjusting the stack on line (10), and pushing the registers on lines (11-14).
- We don't need to initialize anything in this program because all the values of interest are passed to the subroutine by the code that calls the subroutine. We do leave in a "init" label on line (16) as good programming practice; the comment says why the label has no associated code.
- We increment the byte data source pointer by one and the word data destination pointer by two as part of loop administration on line (23) and line (24), respectively.
- We use a do-while loop in the code. This is relatively bullet-proof because we know the number of values to translate that is passed to the program is non zero. We'll redo this solution with another approach to show the possibilities.
- We restore the context on lines (28-32) by popping values off the stack and adjusting the stack pointer.

```

(00) #-----
(01) # Subroutine: byte_to_word
(02) #
(03) # This subroutine finds translates contiguous unsigned byte data
(04) # starting at the value in x10 to word data starting at the address in
(05) # x20. Register x11 holds the number if values to translate.
(06) #
(07) # Tweaked registers: none
(08) #-----
(09) byte_to_word:
(10) store:   addi sp,sp,-16      # room for 4 regs
(11)         sw   x11,0(sp)      # push 4 regs
(12)         sw   x20,4(sp)
(13)         sw   x21,8(sp)
(14)         sw   x10,12(sp)
(15)
(16) init:
(17)
(18) check:  beq  x11,x0,restore # quit if loop count is zero
(19)
(20) loop:   lbu  x21,0(x10)     # get hald data at x10
(21)         sw   x21,0(x20)     # store data as word x20
(22)
(23) admin:  addi x10,x10,1      # incr addr by number of bytes
(24)         addi x20,x20,4      # incr addr by word  number of byte
(25)         addi x11,x11,-1     # decr loop count
(26)         bne  x11,x0,loop    # do it again
(27)
(28) restore: lw   x11,0(sp)     # pop 4 pushed regs
(29)         lw   x20,4(sp)
(30)         lw   x21,8(sp)
(31)         lw   x10,12(sp)
(32)         addi sp,sp,16      # readjust stack pointer
(33)
(34) done:   ret

```

Figure 15.20: Solution to this example problem.

Figure 15.21 shows an alternative solution to this example. The solution below is arguably better. The difference in this solution is that we modeled the main part of the algorithm as a while loop rather than a do while loop. This means that the check for zero loop iterations was part of the body of the loop that than actually checking for that condition in the previous solution (recall we used a do-while loop in that solution).

```

(00) #-----
(01) # Subroutine: byte_to_word
(02) #
(03) # This subroutine finds translates contiguous unsigned byte data
(04) # starting at the value in x10 to word data starting at the address in
(05) # x20. Register x11 holds the number if values to translate.
(06) #
(07) # Tweaked registers: none
(08) #-----
(09) byte_to_word:
(10) store:   addi sp,sp,-16      # room for 4 regs
(11)         sw   x11,0(sp)      # push 4 regs
(12)         sw   x20,4(sp)
(13)         sw   x21,8(sp)
(14)         sw   x10,12(sp)
(15)
(16) init:           # nothing to init
(17)
(18) loop:   beq  x11,x0,restore # quit if loop count is zero
(19)         lbu x21,0(x10)     # get hald data at x10
(20)         sw  x21,0(x20)     # store data as word x20
(21)
(22) admin:  addi x10,x10,1     # incr addr by number of bytes
(23)         addi x20,x20,4     # incr addr by word  number of byte
(24)         addi x11,x11,-1    # decr loop count
(25)         j    loop         # do it again
(26)
(27) restore: lw  x11,0(sp)     # pop 4 pushed regs
(28)         lw  x20,4(sp)
(29)         lw  x21,8(sp)
(30)         lw  x10,12(sp)
(31)         addi sp,sp,16     # readjust stack pointer
(32)
(33) done:   ret                # come on up to the house

```

Figure 15.21: An alternative solution to this problem.

Example 15.16: Two-Digit BCD Number Doubler

Write a RISC-V MCU assembly language subroutine that doubles a two digit BCD number contained x20. The result is passed back to the calling routine in x10.

Solution Notes: Yet even more fun stuff in assembly language programming land. This problem can be done in two distinct ways. The most understandable way would be to translate the code from BCD to binary, double the value, then translate the value back to BCD. This would be generally straightforward as there are many such translation routines out there. But since this problem only deals with a 2-digit decimal value, and we don't have the support translation subroutines already coded, we'll take a different approach.

The algorithm we'll use is to double the 1's digit; if the result is greater than 10, it exceeds the decimal digit range, so we then need to subtract 10 and later increment the 10's digit. We roughly do the same thing for the 10's digit, but in that case, we need to increment the 100's digit. We're adding two 2-digit decimal number (max = 99), the result will be between zero and 198 ([0,198]). You'll see this happen in the algorithm. Here are some other cool things to note about the solution.

- We cleverly forgot to say “don't permanently change any registers”, which means we don't need to push the registers at the beginning of the routine and pop them later. We of course should do this in real life, but not doing so here makes the solution shorter.

- We initialize three values starting on line (09). This includes copying the original value on line (10), clearing an accumulator register on line (09), and keeping around a “carry value” on line (11), which we’ll use for the possible carry from the 1’s to 10’s digit.
- The processing of the 1’s data starts at line (13) where we first mask off all but the 1’s nibble. We then double that value on line (14). We need to check to see if that value is greater than 10, and if it is, we need to decrease it by 10 and add carry value to our carry register x15. We use the slti instruction on line (15) because it works well with immediate values (which is not true of branch instructions). We set the x25 register to indicate the value is less than 10, which means we don’t have to do anything. If the value is not less than 10, we need to subtract 10 from the value, which we do on line (19), and then set the carry value in x15. When we’re done processing the 10’s digit, we’ll add the value in x15 without checking to see what it is.
- We added 0x10 to “increment” the 10’s digit. We do this because we don’t want to have to shift the 10’s digit to the 1’s position. This is a common trick when working with BCD values. Expect to see that again when we process the 10’s digit.
- We accumulate the resulting 1’s value on line (21) whether we’ve modified it or not.
- We then process the 10’s digit starting on line (23). We start by retrieving the original value. Recall that we were using a copy when we processed the 1’s digit. The remainder of the algorithm is similar to the 1’s processing so we’ll not bore you to death with more verbose description. The only difference is that we need to add 0x100 when there is a carry out from the 10’s processing, which we do on line (31). Note on line (32), the assembler is smart enough to handle negative hexadecimal numbers.
- We do the final accumulation on line (33).

```

(00) #-----
(01) # Subroutine: bcd_2x
(02) #
(03) # This subroutine multiplies the two digit BCD value in x10 by two and
(04) # stores the result in x10.
(05) #
(06) # Tweaked registers: x10, x21, x15, x25
(07) #-----
(08) bcd_2x:
(09) init:      mv   x10,x0          # clear x10
(10)          mv   x21,x20         # copy x 20
(11)          mv   x15,x0         # clear carry value
(12)
(13) ones:     andi  x21,x21,0xF    # mask low nib
(14)          add  x21,x21,x21     # 2x low nib
(15)          slti x25,x21,0x0A    # check is < 10
(16)          beq  x25,x0,fix_1s   # branch if not
(17)          j    done_1         # jump if < 10
(18)
(19) fix_1s:  addi  x21,x21,-10    # adjust 1's sum
(20)          addi x15,x15,0x10    # store carry
(21) done_1:  add   x10,x10,x21    # accumulate 1's value
(22)
(23) tens:    mv    x21,x20       # get copy
(24)          andi  x21,x21, 0xF0  # mask
(25)          add  x21,x21,x21     # 2x tens
(26)          add  x21,x21,x15     # add carry
(27)          slti x25,x21,0xA0    # see if > 0xA0
(28)          beq  x25,x0,fix_10s  # branch to fix
(29)          j    done_10        # jump to not fix
(30)
(31) fix_10s: addi  x10,x10,0x100  # increment 100's digit
(32)          addi x21,x21,-0x0A0  # decrement 10's value
(33) done_10: add   x10,x10,x21    # accumulate 10's res
(34)
(35) restore: ret                  # take it home

```

Figure 15.22: The solution to this example.

Example 15.17: Memory-Based Absolute Value Conversion

Write a RISC-V MCU assembly language subroutine that replaced signed bytes in contiguous memory with their absolute values. The address of the first value is passed to the subroutine in x10; the number of value to operate on is stored in register x11. The subroutine should not permanently change and register values.

Solution Notes: More fun stuff in solutionland. This is a relatively straightforward solution, though it does use an instruction that we've not used before. Here are some cool things to note about the solution.

- We left in some test code for this solution on lines (09-16). If you use these in the simulator, you'll be able to see the results develop in memory starting at the junk label.
- The subroutine uses three registers, so we need to make room for those registers on the stack by adjusting the stack pointer on line (19) and then copying the three registers used in the algorithm to memory on lines (20-22). We need to adjust the stack pointer in the direction of lower memory, which we do subtracting 12 from the stack pointer value, which represents 4 bytes for each of the three registers that the subroutines changes. We of course don't know which registers we use until we finish coding the algorithm.
- We choose a while loop for this algorithm because it has an initial check for the loop value, which we do on line (25). There are many possible ways to structure this algorithm, this is the way that makes the most sense for us.
- We load a byte value on line (26) using the lb instruction, which loads signed bytes into a registers. We then use an if-else construct to determine if the value is negative or not. If the value is negative, we must negate it, which we do with the neg instruction on line (29). The neg instruction is a pseudoinstruction, but who really cares? If the value is positive, we do not change it.
- Line (32) has the loop administration, which include decrement the loop counter on line (33) and incrementing the address value on line (32). Pretty exciting stuff.
- When we complete all required iterations, we restore the registers the subroutine uses on lines (37-39). We follow that with an adjustment of the stack pointer on line (40).

```

(00) #-----
(01) # Subroutine: abs_mem
(02) #
(03) # This subroutine multiplies takes the absolute value of signed bytes
(04) # in memory starting at the address in x10, and does this for the number
(05) # of values represented by the count in x11.
(06) #
(07) # Tweaked registers: none
(08) #-----
(09) #----- test code -----
(10) #.data
(11) #junk: .byte -3,-5,4,8,-11
(12) #
(13) #.text
(14) #     li    x11,5
(15) #     la    x10,junk
(16) #----- test code -----
(17)
(18) abs_mem:
(19) store:  addi  sp,sp,-12    # room on stack
(20)         lw   x10,0(sp)    # push regs
(21)         lw   x11,4(sp)
(22)         lw   x12,8(sp)
(23) init:
(24)
(25) loop:   beq   x11,x0,done  # check if zero
(26)         lb   x20,0(x10)   # load value
(27)         bge  x20,x0,write # br if > 0
(28)
(29)         neg  x20,x20      # change sign
(30) write:  sb   x20,0(x10)   # store value
(31)
(32) admin:  addi  x10,x10,1    # incr addr
(33)         addi x11,x11,-1   # decr loop count
(34)         j    loop        # do again
(35)
(36) done:
(37) rstore: lw   x10,0(sp)    # pop regs
(38)         lw   x11,4(sp)
(39)         lw   x12,8(sp)
(40)         addi sp,sp,12    # adjust sp
(41)         ret   # bring it home

```

Figure 15.23: The solution to this example.

Example 15.18: Sorting Values

Write a RISC-V MCU assembly language subroutine that sorts ten words in descending order. The ten words are contiguous in memory and start at the address stored in register x10. The subroutine should not permanently change and register values.

Solution Notes: More fun stuff in solutionland. The most straightforward sort is bubble sort. While this solution is straightforward, it not efficient computationally speaking. My vote goes for the straightforwardness of the solution. There are many ways to structure this code; I've opted to do that way that divides the complexity of the code into subroutines. The bubble sort is a classic “loop inside of a loop” algorithm, so we use the notion of the “inside loop” and the “outside loop” throughout the solution description. Here are other lowlights of the solution.

- I left in some test code so you can give it a try in the simulator; this code is on lines (08-11).
- The algorithm uses six registers, so we push them all. The subroutine also calls a subroutine, which means we also must push the return address register, which we do on line (20).

- The init code starts on line (23) and includes initializing both the inside and outside loops to 9. The significance of 9 is that it is one less than 10, which is the number of values we want to sort. We'll be tweaking with the address, so we also make a working copy of the original address in x10 on line (25).
- The inner-loop basically calls the swap subroutine. All the swapping work is done in the swap routine, which we intentionally did to simplify the calling code in the bbl_sort subroutine.
- The calling code passes the address of the first word to consider; the subroutine leverages the fact that the second word is 4 beyond (bytes or one word) the first word. The swap routine does not save registers; we save the registers the swap routine uses in the store and restore sections of the bbl_sort subroutine.
- The swap_q loads the two words to compare into registers. If the two values need to be swapped, the subroutine swaps them by storing them in opposite addresses from which they were loaded. Otherwise, the subroutine simply returns
- The inner loop administration consists of advancing the address and decrementing the loop count, which is done on lines (31-32).
- The outer loop administration consists of resetting the inner loop counter on line (36), decrementing the outer loop counter on line (35), and resetting the address value back to the start of the numbers to be sorted on line (37).
- Lines (41-48) restore context, including the return address, which we needed to save because we called the swap_q subroutine.
- We had to adjust our labels in this program. I like to use labels such as "done", but there are several contexts in which I needed to say "done". The solution I went with is to use "done_1" and "done_2". Not too exciting, but it works.
- This solution included two subroutines, both with nice looking (if I do say so myself) headers with all the pertinent information included.

```

(00) #-----
(01) # Subroutine: bbl_sort
(02) #
(03) # This subroutine sorts (bubble sort) 10 words in memory
(04) # starting at the address passed in x10.
(05) #
(06) # Tweaked registers: none
(07) #-----
(08) #.data      # test code for simulator
(09) #arr:      .word 10,3,5,4,3,8,3,4,7,1
(10) #.text
(11) #          la      x10,arr
(12)
(13) bbl_sort:
(14) store:      addi   sp,sp,-28      # adjust stack pointer
(15)             sw    x10,0(sp)      # save context
(16)             sw    x11,4(sp)
(17)             sw    x12,8(sp)
(18)             sw    x25,12(sp)
(19)             sw    x26,16(sp)
(20)             sw    x30,20(sp)
(21)             sw    ra,24(sp)      # push return address
(22)
(23) init:       li     x25,9         # inside count
(24)             li     x26,9         # outside count
(25)             mv     x30,x10      # array start address
(26)
(27) loop_out:   beq    x26,x0,done_out # outer while loop
(28)
(29) loop_in:    beq    x25,x0,done_in  # inner while loop
(30)             call   swap_q        # do swap
(31)             addi   x25,x25,-1     # decr inner loop count
(32)             addi   x10,x10,4     # advance address
(33)             j      loop_in       # keep doing it
(34)
(35) done_in:    addi   x26,x26,-1     # decr outer loop count
(36)             li     x25,9         # reload inner loop count
(37)             mv     x10,x30      # reload starting address
(38)             j      loop_out     # jump to outer loop
(39)
(40) done_out:
(41) restore:   lw     x10,0(sp)      # restore context
(42)             lw     x11,4(sp)
(43)             lw     x12,8(sp)
(44)             lw     x25,12(sp)
(45)             lw     x26,16(sp)
(46)             lw     x30,20(sp)
(47)             lw     ra,24(sp)     # restore return address
(48)             addi   sp,sp,28     # adjust stack
(49) done_1:    ret
(50) #-----
(51)
(52) #-----
(53) # Subroutine: swap_q
(54) #
(55) # This subroutine sorts two words in memory starting at the
(56) # address passed in x10.
(57) #
(58) # Tweaked registers: x11,x12
(59) #-----
(60) swap_q:
(61)             lw     x11,0(x10)
(62)             lw     x12,4(x10)
(63)             bge   x11,x12,done_2
(64)             sw    x11,4(x10)
(65)             sw    x12,0(x10)
(66) done_2:    ret
(67) #-----

```

Figure 15.24: Yet another meaning-packed solution.

Example 15.19: Increasing Number Determination

Write a RISC-V assembly language subroutine that determines if all the data in contiguous memory locations is non-zero and always increasing. Consider the data to be signed halfwords. X15 contains the address of the first halfword; continue checking until two contiguous pieces of data are equivalent. Return the number of increasing data in x12. Assume the data terminates the algorithm in a reasonable amount of time.

Solution Notes: This is a classic hardware problem done many times in your introductory digital design course. This is another one of those problems that has a special starting initialization that we want to use in order to make the algorithm more generic and thus easier to encode. For this problem, it means starting the algorithm by loading the first piece of data.

- This problem has a special and exciting exit condition from the loop; we iterated many of our past loops a known number of times; this problem iterates a conditional number of times. Note that the ending condition is when we find two pieces of contiguous data that are equivalent.
- For this problem, we are only reading data, so there is no need for a **sb** instruction.
- We left in the test code for the subroutine on lines (10-14).
- The initialization comprised of a few things in order to make the algorithm more generic. Line (17) clears a counter register to hold the number of pieces of data in a row. We then need to get the first piece of data on line (18), which we check to see if it's zero (and quit if it is). We then need to increment our counter and address value. The fact that we increment the counter is arbitrary. This means there is one at least one piece of non-zero data increasing data in the way we did this.
- The body of the algorithm gets another piece of data on line (24). If the data is zero, we quit by branching to the end (line 25). We also check to see if the current data is less than or equal to the old data; we quit if it is (line 26).
- The loop administration for this algorithm includes three items. First, we copy the new data to the old data on line (28). Second, we increment the memory address on line (29). Third, we increment our counter on line (30).
- We opted not to save context on this problem simply to save space. Once you do it a few times, it's the same stuff over and over again. Not too exciting after you do it a few times.

```

(00) #-----
(01) # Subroutine: Cnt_incr
(02) #
(03) # This subroutine goes to a specific address in memory and counts
(04) # how many pieces of data are both non-zero and increasing. The count
(05) # is returned in x12.
(06) #
(07) #
(08) # Tweaked registers: x10,x11,x12,x15
(09) #-----
(10) #.data
(11) #junk: .byte -6, -2, 1, 1, 3, 8, 8
(12) #
(13) #.text
(14) #      la      x15,junk
(15)
(16) Cnt_incr:
(17) init:  mv     x12,x0          # counter reg
(18)        lb     x10,0(x15)     # get first piece of data
(19)        beq   x10,x0,done     # quit if data=0
(20)        addi  x15,x15,1       # increment address
(21)        addi  x12,x12,1       # increment counter
(22)
(23)
(24) loop:  lb     x11,0(x15)     # get more data
(25)        beq   x11,x0,done     # quit if data=0
(26)        ble   x11,x10,done    # quit if not greater than
(27)
(28) admin: mv     x10,x11         # store last data
(29)        addi  x15,x15,1       # incr address
(30)        addi  x12,x12,1       # incr counter
(31)        j     loop           # do it again
(32)
(33) done:  ret                    # take it home

```

Figure 15.25: Solution to this example problem.

Example 15.20: Increasing Number Determination Yet Again

Write a RISC-V assembly language subroutine that determines if all the data in contiguous memory locations is non-zero and always increasing. Any zero value does not count toward the average. The final result should be rounded up and returned to the calling code in register x25.

Solution Notes: This problem is similar to other problems, but with two new items. First, we add (and count that add) only if the value is non-zero. Second, when we do that math to take the average, we round up instead of truncating. Note that when we right-shift a number, we lose those bits as part of the number, so we are by definition truncating the value. This is easy to do with the shift-right instructions, but not always what we want to do. Here is some other fun stuff in the solution. A.

- The init routine clears the accumulator and set the loop count on lines (20-21).
- We model the body of this algorithm with a while loop, which starts on line (23).
- The body of the algorithm is to get data (line (24)) and use that data in the calculation it is not zero. That means we first examine the data to see if it's a candidate for adding on line (25). If we include the data, we then branch to our admin line, though we branch to only part of the admin (the part that advances the counter). When the value is non-zero, we advance the counter but we also decrement the loop count on the line before the admin label (line (28)).

- When we exit the loop, we must process the sum, which means a divide by 6 using a shift-right instruction on line (34).
- In order to round the result up, we are going to isolate the 6th bit from the right and add it to the final value. We first shift the sum right by five bits on line (32), then mask all but the LSB on line (33). We don't know if this value is a 1 or a 0, but we don't care; we simply add this value to the sum that right-shifted by 6 places on line (35). Wow. Too much excitement for one problem.

```

(00) #-----
(01) # Subroutine: nz_avg_64
(02) #
(03) # This subroutine finds the average of 64 non-zero unsigned words in
(04) # memory. The first piece of data must be located at the address in x10;
(05) # all other data is contiguous. Any zero value does not add to the
(06) # overall number of value being averaged. The final answer is rounded up
(07) # as opposed to being truncated.
(08) #
(09) # Tweaked registers: x25, x20, x11, x10
(10) #-----
(11)
(12) #-- test code -----
(13) .data
(14) junk: .word 1,2,4,8,16,0,32,64,128
(15)
(16) .text
(17)     la    x10,junk
(18) #-----
(19) nz_avg_64:
(20) init:   mv    x25,x0      # clear accum
(21)        li    x11,64     # set count (8 for test code)
(22)
(23) loop:   beq   x11,x0,done # see if done
(24)        lw    x20,0(x10)  # get data
(25)        beq   x20,x0,admin # skip if zero
(26) more:   add   x25,x25,x20 # accumulate
(27)
(28)        addi  x11,x11,-1   # decr loop count
(29) admin:  addi  x10,x10,4   # advance addr
(30)        j     loop        # rinse, repeat
(31)
(32) done:   srli  x11,x25,5   # save lsb
(33)        andi  x11,x11,1   # mask lsb
(34)        srli  x25,x25,6   # take avg
(35)        add   x25,x25,x11 # add 2^-1 bit
(36)
(37)        ret                    # take it home

```

Figure 15.26: Solution to this example problem.

Example 15.21: Register-Based Parity Determination

Write a RISC-V assembly language subroutine that determines the parity of the value in register x10. The parity is passed back to the calling program in x10, where x10=1 indicates odd parity and x10=0 is even parity.

Solution Notes: This is a popular operation, but also a great opportunity to use a LUT in a solution. We've probably done this solution previously not using a LUT, but we'll include that solution here as well. Here is some fun stuff to note about the solution in Figure 15.27

- The first part of the solution is to define the LUT, which we do on lines (11-12). We of course need to put the LUT in the data segment, which we declare using the `.data` directive on line (10).
- The LUT defines 16 bytes of data, which represents the number of bits set in the set of 4-bits (nibble). For example, the 0, 1, 1, 2 (the first four values in the LUT) represent the number of bits set in 0000, 0001, 0010, and 0011. We provide a value for each possible combination of four bits.
- We'll look up nibbles in the table, which means we have to perform eight table look-ups. This means we need a loop that iterates eight times, which we initialize on line (15).
- We'll be counting bits, or accumulating them, so we'll need to clear a register to use as a counter, which we do on line (16).
- We then need to store the base address of the LUT, which is the value associated with the "par_val" label. We use the `la` instruction to do this on line (17).
- The body of the loop is a while loop, so it starts with checking if there is more work to do, which we do on line (19).
- The first step in the algorithm is to mask the lower nibble of the data of interest (x10), which we do on line (20). This gives us the offset into the LUT. We add this offset to the base address of the LUT, which we previously stored in x30; this is on line (21). The value in x22 is now the address of the value we're looking for in the table, a value we grab with the load byte unsigned instruction on line (22). The data we load is the number of bit set in the nibble we got by masking on line (21). We accumulate the number on line (23).
- The loop admin first needs to shift the original value right by four places, which we do one line (25). We then need to decrement the loop count, which we do on line (26).
- Once we complete the loop, we need to mask the result (mask the LSB) and then store the result in x10; we do both of these tasks with one instruction on line (30).

```

(00) #-----
(01) # Subroutine: Par_32b
(02) #
(03) # This subroutine finds parity of the value in x10 and returns the
(04) # result in x10 where x10=1 = odd and x10=0 is even parity.
(05) #
(06) # Tweaked registers: x20, x10, x21, x22, x15
(07) #-----
(08)
(09) # num of bits set in each nibble (range: [0,15])
(10) .data
(11) par_val: .byte 0,1,1,2,1,2,2,3 # values 0 -> 7
(12)          .byte 1,2,2,3,2,3,3,4 # values 8 -> 15
(13) .text
(14) Par_32b:
(15) init:    li    x20,8          # loop count
(16)          mv    x15,x0         # bit count
(17)          la    x30,par_val    # get address of LUT
(18)
(19) loop:    beq   x20,x0,done     # done yet?
(20)          Andi  x21,x10,0xF    # calc table offset
(21)          add   x22,x30,x21    # calc index
(22)          lbu   x22,0(x22)     # table look-up
(23)          add   x15,x15,x22    # accumulate
(24)
(25) admin:   srli  x10,x10,4      # shift right one nibble
(26)          addi  x20,x20,-1     # decr loop count
(27)          j     loop          # rinse, repeat
(28)
(29) done:    mv    x10,x15        # load count to x10
(30)          andi  x10,x10,1     # mask LSB
(31)          ret                   # take it on home

```

Figure 15.27: Solution to this example problem.

This above solution requires about 150 clock cycles to execute. We redo this problem (without verbose description) in Figure 15.28. This solution uses a different algorithm that does not use a LUT. Although the code is noticeably shorter, the runtime is significantly greater, as the algorithm in Figure 15.28 requires almost 400 clock cycles to execute. The final word here is that the LUT solution ran faster, but it required more code space and more data space. It's a common tradeoff in computerland.

```

(00) #-----
(01) # Subroutine: Par_32b
(02) #
(03) # This subroutine finds parity of the value in x10 and returns the
(04) # result in x10 where x10=1 = odd and x10=0 is even parity.
(05) #
(06) # Tweaked registers: x15, x20, x11, x10
(07) #-----
(08)
(09) .text
(10) Par_32b:
(11) init:    li    x20,32         # loop count
(12)          mv    x15,x0         # bit count
(13)
(14) loop:    beq   x20,x0,done     # done yet?
(15)          Andi  x21,x10,0x1    # mask bit
(16)          add   x15,x15,x21    # accumulate
(17)          srli  x10,x10,1      # shift
(18) admin:   addi  x20,x20,-1     # decr loop count
(19)          j     loop          # rinse, repeat
(20)
(21) done:    andi  x10,x15,1     # mask LSB
(22)          ret                   # take it on home

```

Figure 15.28: Solution to this example problem.

Example 15.22: Interrupt Paced I/O

Write a RISC-V OTTER interrupt-driven assembly language program that does the following. Each time the MCU receives an interrupt, the program inputs a value from port address 0x11002222 and adds this value to a running total. Once the program receives ten interrupts, the program outputs the sum to address 0x11003333. The program then waits for a button press (LSB of port address 0x11005555) to happen, to start accumulation again from zero.

- Don't worry about button debouncing for this button.
- Don't do any I/O from the ISR

Solution Notes: This is our first interrupt driven program. This follows a standard format of interrupt driven programs in that it's not too exciting (contrived problems) but it does show the correctly architected interrupt driven program. Here's some stuff to note in the solution.

- We first place the I/O addresses called out in the problem into registers, which we do on lines (11-13). Putting the addresses in registers saves instructions later in the program.
- Because this is an interrupt driven program, we need to load the interrupt vector (the address of the first instruction in the ISR) into CSR[mtvec]. We do this on lines (15-16).
- We need to continually re-enable the interrupts after we receive an interrupt, for we place a 1 in x9 on line (18). Once again, this saves instructions later in the program.
- We then do some more initialization stuff including clearing the accumulator and setting the iterative count value on lines (20-21).
- Lastly for the initialization stuff, we turn on the interrupts on line (24). We use x8 as a flag register, so we clear that value on line (23).
- The program then goes into a polling loop waiting for an interrupt. This loop constantly checks the x8 register, which we use as a flag. This register is initially cleared, and is then only set when the program receives an interrupt. The entire ISR is thus to set that x8 value to non-zero and return from the ISR (lines (46-47)).
- When the program receives an interrupt we first input a value and add that value to our running total, which we do on lines (28-29).
- Next the program does admin stuff that first include decrementing the loop count on line (31). If the loop count is non-zero, we're done with admin stuff and we get ready for the next interrupt by jumping back to somewhere near the start of the program. We arranged the init code such that we could do this and thus saved a few instructions. If there are still more values to add, we only need to turn the interrupts back on, which we do by jumping to the redo_2 label.
- If the loop count has run out, we need to first output the accumulated value, which we do on line (34). We then need to go into a polling loop waiting for a button press. The polling loop on lines (36-38) consists of inputting the buttons, masking the LSB, and checking to see if it is set or not. If it is not set, the button of interest is not pressed and we keep looking/waiting. If the bit is set, there was a button press and we branch to our complete initialization routine starting on the line associated with the redo_1 label.

```

(00) #-----
(01) # This program that does the following: Each time the MCU receives
(02) # an interrupt, the program inputs a value from port address 0x11002222
(03) # and adds this value to a running total. Once the program receives
(04) # ten interrupts, the program output the sum to address 0x11003333. The
(05) # program then waits for a button press (LSB of port address 0x11005555)
(06) # to happen, to start accumulation again from zero. Don't worry about
(07) # button debouncing for this button.
(08) #-----
(09)
(10) .text
(11) .init:  li    x10,0x11002222 # input port address
(12)        li    x11,0x11003333 # output port address
(13)        li    x12,0x11005555 # button port address
(14)
(15)        la    x6,ISR          # load address of ISR into x6
(16)        csrrw x0,mtvec,x6    # store address as interrupt vector CSR[mtvec]
(17)
(18)        li    x9,1           # store 1 for interrupt enable
(19)
(20) redo_1: mv    x20,x0         # accumulation value
(21)        li    x25,10         # iteration count
(22)
(23) redo_2: mv    x8,x0          # clear flag value
(24)        csrrw x0,mie,x9      # enable interrupts
(25)
(26) wait:   beq   x8,x0,wait     # wait for interrupt
(27)
(28) body:   lw    x15,0(x10)     # input data
(29)        add   x20,x20,x15     # accumulate
(30)
(31) admin:  addi  x25,x25,-1     # decrement loop count
(32)        bnez  x25,redo_2     # jump to reset stuff
(33)
(34) done:   sw    x20,0(x11)     # output accumulated value
(35)
(36) poll:   lw    x20,0(x12)     # input buttons
(37)        andi  x20,x20,1      # mask LSB
(38)        beq   x20,x0,poll    # keep looking for button press
(39)
(40)        j     redo_1          # rinse, repeat
(41) #-----
(42)
(43) #-----
(44) #- The ISR: sets bit x8 to flag task code
(45) #-----
(46) ISR:    mv    x8,x9
(47)        mret
(48) #-----

```

Figure 15.29: The solution to this example.

Example 15.23: Interrupt-Driven Programming

Write a RISC-V OTTER interrupt-driven assembly language program that does the following. When a button is pressed (LSB of port address 0x11008888), the program waits for interrupts. Each time it receives an interrupt, the program stores the average of the current value it read from port address 0x11009999 with the previous value it read from that port in contiguous memory addresses 0x0000FF00. After 100 values are written to memory (101 interrupts), the program then waits for another button press. Don't write a value until after the MCU receives the second interrupt.

- Don't worry about button debouncing for this button.
- Don't do any I/O from the ISR
- Assume additions never overflow 32 bits.

Solution Notes: This is another interrupt driven program. This has the standard format of an interrupt driven program. Here's some stuff to note in the solution.

- We first place the I/O addressed called out in the problem into registers, which we do on lines (12-13).
- We next store the address of the interrupt service routine in the CSR register (16).
- We need to do many stores of data to memory starting at the given memory location, so we put that value in a register also on line (18).
- We'll need to enable interrupts quite often, so we leave a '1' in x9 on line (19).
- We then do a bunch of administrative work starting at the instruction with the "restart" label. As you'll see later in this solution, we've arranged this solution such that we can reuse the three instructions at this label, lines (21-23), later in the program. This program does the same thing over and over again, so it makes sense to reuse as much code as possible.
- The next this to do is wait for a button press, which is essentially the dreaded poll line lines (25-27). We first load some data, mask it, and check for the right-most button being pressed.
- If program execution falls through the poll, it is then that we enable interrupts on line (29). We generally keep interrupts disabled until we truly need them (or are ready for them). In this program, we don't need to deal with interrupts until the button has been pressed.
- After the interrupts are enable, we go into a second poll that is waiting for interrupts, which is on line (31). Once we receive an interrupt, we drop out of the poll and start doing more meaningful stuff. We first load some real data on line 33. This represents the first piece of data, so we don't do anything with it because we're going to store an average of two pieces of data in memory. We clear the flag on line (34), and enable the interrupts again on line (35).
- We enter a third poll on line (31), waiting for more interrupts. The functionality is similar to the previous poll in that we first load some data on line (39). We next need to add the new data to the previous value on line (40) and average the two pieces of data using a shift right on line (41). This is the value we want to store, which we do one line (42). The last part of the body of this algorithm is to make the more recent data into the older data in preparation for receiving more interrupts.
- The loop admin includes clearing the x8 flag on line (45), advancing the address on line (46), and decrementing the loop count one line (17). At this point, if the loop count is less than zero, we leave the algorithm by jumping to the restart label. The code at the restart label prepares the algorithm to happen again after yet another button press. If there are still more counts left in the loop count, we enable interrupts on line (51) and jump back to the third poll, which is associated with the "wait3" label.

- The ISR is relatively simple; it comprises of signaling the background task by putting a non-zero value in x8.

```

(00) #-----
(01) # This program that does the following: When a button is pressed
(02) # (LSB of port address 0x11008888), the program waits for interrupts.
(03) # Each time it receives an interrupt, the program stores the average of
(04) # the current value it read from port address 0x11009999 with the
(05) # previous value it read from that port in contiguous memory addresses
(06) # starting at 0x0000FF00. After 100 values are written to memory
(07) # (101 interrupts), the program then waits for another button press
(08) # and repeats the same functionality. The buttons are not debounced.
(09) #-----
(10)
(11) .text
(12) .init: li    x10,0x11008888 # button port address
(13)       li    x11,0x11009999 # input port address
(14)
(15)       la    x6,ISR          # load address of ISR into x6
(16)       csrrw x0,mtvec,x6    # store address as interrupt vector CSR[mtvec]
(17)
(18)       li    x29,0x0000FF00 # establish memory address
(19)       li    x9,1           # store 1 for interrupt enable
(20)
(21) restart: mv    x8,x0        # clear flag value
(22)        li    x17,100       # set loop count
(23)        mv    x30,x29       # make working copy of memory address
(24)
(25) wait1:  lw    x20,0(x10)     # get button data
(26)        andi  x20,x20,1     # mask button data
(27)        beq   x20,x0,wait1  # keep waiting
(28)
(29)        csrrw x0,mie,x9     # enable interrupts
(30)
(31) wait2:  beq   x8,x0,wait2   # wait for first interrupt
(32)
(33)        lw    x25,0(x11)    # get first piece of data
(34)        mv    x8,x0        # clear flag value
(35)        csrrw x0,mie,x9     # enable interrupts
(36)
(37) wait3:  beq   x8,x0,wait3   # wait for more interrupts
(38)
(39)        lw    x26,0(x11)    # get first piece of data
(40)        add   x25,x25,x26   # add to previous input
(41)        srli  x25,x25,1     # divide by 2
(42)        sw    x25,0(x30)    # store avg in memory
(43)        mv    x25,x26      # save previous input
(44)
(45) admin:  mv    x8,x0        # clear flag value
(46)        addi  x30,x30,4     # advance address
(47)        addi  x17,x17,-1    # decrement loop count
(48)
(49)        bltz  x17,restart   # start over if done
(50)
(51)        csrrw x0,mie,x9     # enable interrupts
(52)        j     wait3        # wait for next interrupt
(53) #-----
(54)
(55) #-----
(56) #- The ISR: sets bit x8 to flag task code
(57) #-----
(58) ISR:    mv    x8,x9
(59)        mret
(60) #-----

```

Figure 15.30: The solution to this example.

Example 15.24: Digital Averaging Filter

Write a RISC-V assembly language subroutine that implements a digital averaging filter. This subroutine averages four memory locations of unsigned halfwords starting at `mem[x10]`, and replaces the halfword at address `x10` in memory with the average of the four contiguous values. The subroutine replaces the number of values passed to the subroutine in register `x11`. The average rounds up before written.

Solution Notes: This is a classic subroutine that implements a digital filter. This is actually a potentially useful subroutine; you'll probably see other filters in later subroutines because that's all I can think of at this point in time.

- First thing to note is the great header. We once again did not push/pop registers simply to save space on the paper. All good subroutines protect the registers they use.
- We start the algorithm by loading four chunks of data (halfwords) into four registers, which we do on lines (17-20).
- Starting as line (22), we add the four previous values we loaded. This takes three lines.
- After we add the four values, we shift the values right one time (divide by two). We need to divide by four, but we divide by two because we need to round up the average we're calculating. The first divide is on line (25). We mask that result (the LSB) on line (26), we later add that value to the calculation after we divide it another time, which we do on line (27). We add the roundup bit on line (28).
- We have two forms of admin to do in the subroutine; we have both data admin and normal admin. We start the data admin on line (30), where we first store our calculated result. After that we shift the data in our data registers (`x20-x23`) on lines (31-33).
- The normal admin includes advancing the memory address on line (35), loading some new data on line (36), and decrementing the loop count on line (38). After we decrement the loop count, we check the loop count and branch necessary.

```

(00) #-----
(01) # Subroutine: Dig_smoothing_filt_4x
(02) #
(03) # This subroutine implements a digital smoothing filter, AKA a smoothing
(04) # filter, AKA, a low-pass filter. This subroutines replaces the 16-bit
(05) # value as mem[x] with the average of memory locations mem[x], mem[x+1],
(06) # mem[x+2] and mem[x+3]. The data is stored starting at the address
(07) # passed to the subroutine in x10. The number of values to filter is
(08) # passed to the subroutine in register x11. This subroutines assumes
(09) # there is enough data to generate valid averages for all data, which
(10) # means there needs to be more data than the count in x11.
(11) #
(12) # Tweaked registers: x10,x11,x20,x21,x22,x23,x24
(13) #-----
(14)
(15) Dig_smoothing_filt_4x:
(16)
(17) preload:    lhu    x20,0(x10)    # get first piece of data
(18)            lhu    x21,4(x10)    # get 3 more half words
(19)            lhu    x22,8(x10)
(20)            lhu    x23,12(x10)
(21)
(22) loop:      add    x20,x20,x21    # add first two locations
(23)            add    x20,x20,x22    # accumulate third value
(24)            add    x20,x20,x23    # accumulate fourth value
(25)            srli   x20,x20,1     # divide by 2
(26)            andi   x24,x20,1     # mask LSB
(27)            srli   x20,x20,1     # divide by 2 (again)
(28)            add    x20,x20,x24    # round up
(29)
(30) d_admin:   sh     x20,0(x10)    # store result in memory
(31)            mv     x20,x21        # shift data around
(32)            mv     x21,x22
(33)            mv     x22,x23
(34)
(35) admin:     addi   x10,x10,2     # advance data pointer
(36)            lhu    x23,12(x10)   # get new data
(37)
(38)            addi   x11,x11,-1    # decrement loop count
(39)            bgez  x11,loop       # jump if more data to process
(40)
(41)            ret     # take it home jimie

```

Figure 15.31: The solution to this example.

Example 15.25: Digital Median Filter

Write a RISC-V assembly language subroutine that implements a digital median filter. This subroutine examines three contiguous memory locations of unsigned halfwords starting at `mem[x10]`, and transfers the median value to the address passed to the subroutine in register `x8`. The subroutine thus does not change any of the original values in memory; it transfers the median of the original data to another area in memory.

Solution Notes: This is a classic subroutine that implements yet another type of digital filter. This too is actually a potentially useful subroutine, and one of several filters you'll see in this set of problems. Here are the details for this example:

- The subroutine has both a meaningful description in the provided header, and also some test code that has been left in the text but commented out on lines (11-17).

- We generally stopped writing subroutines that protect the data by pushing it onto the stack at the start of the subroutine (to save space), but we do need to do something different in this subroutine. Because this subroutine calls another subroutine, we need to save the return address associated with this subroutine before we call the nested subroutine. We do this by pushing it on the stack on lines (21-22).
- The algorithm then structured such that we're ready to start, which we do by loading three halfwords into registers on lines (24-26). Using registers for the data is horrifically non-generic, which causes us to write a one-off sort algorithm. We call the sort algorithm on line (28).
- The sort algorithm is on lines (43-69). Note the subroutine has a nice descriptive header that includes which registers are tweaked (we once again don't bother saving/restoring registers). The algorithm is a hardcoded bubble sort, which hardcodes the inner loop, but does allow the outer loop to be parameterized. The sort uses the XOR register swapping trick to sort individual registers; fun stuff.
- Once we've sorted the data, the data in the middle of the three register value-wise is the data we choose to store at the new memory location, which we do on line (30). The data "in the middle" is thus the median value, as the filter name implies, and becomes the "new" value.
- The next part of the algorithm is the loop administrative tasks that include advancing the address pointer on line (32), decrementing the loop count on line (34), and checking to see if we need to do more iterations or not on line (35). We modeled this loop as a do-while loop, and we opted to make the subroutine less safe by not verify the loop count passed to the algorithm in x11 was non-zero. Once again, we did this to save space on the page.
- When we run out of iterations, the algorithm is done and we prepare to exit by popping the return address off the stack on lines (37-38). Recall that we had to do this because we used nested subroutines in our approach.
- We return from the subroutine on line (40); we opted to include a "done" label for clarity, even though the code itself never actually uses the done label (it's there to make human readers happy).

```

(00) #-----
(01) # Subroutine: Dig_median_filter_3x
(02) #
(03) # This subroutine implements a digital median filter. The values from
(04) # three contiguous memory locations are sorted; the median value of these
(05) # three values are stored at a different memory location. The data to filter
(06) # is stored as unsigned halfwords starting at the address in x10. This
(07) # subroutine stores the filtered data starting at the address in x8.
(08) # The number of times to # filter these sets of data is stored in x11
(09) #
(10) # Tweaked registers: x8,x10,x20,x21,x22, and ra (x1)
(11) #----- test code -----
(12) # .data
(13) # junk:      .half      23,26,25,28,29,30,32
(14) # .text
(15) #          la      x10,junk
(16) #          li      sp,0x6120
(17) #-----
(18) .text
(19) Dig_median_filter_3x:
(20)
(21) init:      addi    sp,sp,-4      # make space on stack
(22)           sw      ra,0(sp)      # push return address
(23)
(24) load:      lhu     x20,0(x10)    # get first piece of data
(25)           lhu     x21,2(x10)    # get 2 more half words
(26)           lhu     x22,4(x10)
(27)
(28)           call   Median        # sort three input values
(29)
(30) loop:      sh      x21,0(x8)     # store median value
(31)
(32) admin:     addi    x10,x10,2     # advance data pointer
(33)
(34)           addi    x11,x11,-1    # decrement loop count
(35)           bgez   x11,load       # jump if more data to process
(36)
(37)           lw      ra,0(sp)      # restore return address
(38)           addi    sp,sp,4       # pop from stack
(39)
(40) done:      ret                    # take it home jimie
(41) #-----
(42)
(43) #-----
(44) #- Subroutine: Median
(45) #
(46) # This subroutine sorts the values in three register: x20,x21, & x22.
(47) # The sorting order does not matter because we are interested in
(48) # the median value, which will be in x21 at end of subroutine
(49) #
(50) # Tweaked - 418 -ource- 418 -y: x30,x20,x21,x22
(51) #-----
(52) Median:    li      x30,2         # load loop count
(53)
(54) loop_m:    beq     x30,x0,sorted  # check loop count
(55)           bge     x20,x21,nswp_1  # compare regs
(56)           xor     x20,x20,x21     # swap if needed
(57)           xor     x21,x21,x20
(58)           xor     x20,x20,x21
(59)
(60) nswp_1:    bge     x21,x22,lp_admin # compare regs
(61)           xor     x21,x21,x22     # swap if needed
(62)           xor     x22,x22,x21
(63)           xor     x21,x21,x22
(64)
(65) lp_admin:  addi    x30,x30,-1     # decrement loop count
(66)           j      loop_m          # rinse, repeat
(67)
(68) sorted:    ret                    # done
(69) #-----

```

Figure 15.32: The solution to this example.

Example 15.26: Odd-Even Value Check

Write a RISC-V assembly language subroutine that counts the number of odd, even, and zero values in a section of memory. The memory starts at the address in x10, and x11 holds the number of memory locations to analyze. The results of odd, even, and zero counts are stored contiguous words starting at the address passed to the subroutine in register x20. The input data to inspect is word values.

Solution Notes: This is a classic subroutine that implements yet another type of digital filter. This too is actually a potentially useful subroutine, and one of several filters you'll see in this set of problems. Here are the details for this example:

- Once again, meaningful subroutine description including a list of tweaked registers and some test code that you can use to verify the subroutine actually works: lines (0-18).
- Once again, we do not store context with pushes/pops of registers.
- The initialization code for this subroutine consists of clearing three register that the subroutine uses as accumulators for the odd, even, and zero counts. The code starts at the “init” label on lines (22-24).
- We model this algorithm using a while loop, so we first check to see if we have more values to count on line (26).
- The heart of the algorithm starts on line (26), where we grab some data. We then check to see if the data is zero on line (28); if the data is zero, we increment our zero count on line (29). If the data is non-zero, we need to examine the LSB to determine if it is odd or even, which we want to do in such a way as to increment the odd and even counters without looking at the value. We first mask the LSB of the original data on line (31), and then add the result to the odd counter on line (32). We then toggle that LSB on line (33) and add it to the even counter on line (34). Somewhat of a tricky algorithm, but it works with doing extra conditional statements.
- Loop admin consists of advancing data address and decrementing the loop counter on lines (36-37).
- When the loop count is zero, we save the three counts in three contiguous addresses on lines (40-42).
- We return from the subroutine on line (44); note that we include a “leave” label which brings comfort to human viewers of the code.

```

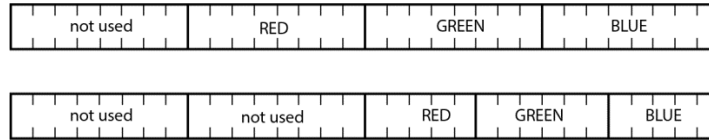
(00) #-----
(01) # Subroutine: Count_vals:
(02) #
(03) # This subroutine counts the number of odd, even, and zero values for words
(04) # starting at address x10 in memory. The subroutine uses x11 to hold the
(05) # count of the number of values to examine. The counts are stored at three
(06) # contiguous words starting at the address in x20. The subroutine considers
(07) # zero to be an even value.
(08) #
(09) # Tweaked register: x10,x11,x25,x26,x27,x30
(10) #-----
(11)
(12) #----- test code -----
(13) # .data
(14) # junk:      .word      23,26,25,0,0,0,34,23
(15) # .text
(16) #          la      x10,junk
(17) #          li      x11,8
(18) #-----
(19) #.text
(20) Count_vals:
(21)
(22) init:      mv      x25,x0      # odd count
(23)           mv      x26,x0      # even count
(24)           mv      x27,x0      # zero count
(25)
(26) loop:      beq     x11,x0,done   # see if we're done
(27)           lw      x30,0(x10)    # get data
(28)           bne     x30,x0,not_z  # check for zero
(29)           addi    x27,x27,1     # increment zero counter
(30)
(31) not_z:     andi    x30,x30,1     # mask LSB
(32)           add     x25,x25,x30    # increment odd count
(33)           xori    x30,x30,1     # toggle
(34)           add     x26,x26,x30    # increment even count
(35)
(36) admin:     addi    x10,x10,4     # advance address counter
(37)           addi    x11,x11,-1    # decrement loop count
(38)           j       loop         # rinse, repeat (if necessary)
(39)
(40) done:      sw      x25,0(x20)    # store odd count
(41)           sw      x26,4(x20)    # store even count
(42)           sw      x27,8(x20)    # store zero count
(43)
(44) leave:     ret
(45) #-----

```

Figure 15.33: The solution to this example.

Example 15.27: RGB Data Compressor

Write a RISC-V assembly language subroutine that converts a register containing three bytes of RGB data (red, green, blue) into a register containing two bytes of RGB data. The subroutine removes the lower bits of data in each color byte according to the diagram below. The calling program places the data in x25; the subroutine returns the data in that same register. Don't allow the subroutine to permanently change any register other than x25.



Solution Notes: This is actually a useful algorithm that I've actually used on the job. This represents an instant compression of an image (compression means reduction in storage size) by 33.3%. You probably would not notice the change in image quality on anything but a high-quality display. Here are the details for this example:

- Yet another meaningful subroutine description including a list of tweaked registers and some test code that you can use to verify.
- We first make copies of the passed data on lines (15-17). There are many approaches to performing the required tasks in this subroutine; we'll take what we feel is the easiest.
- We process data one color at a time starting with the red byte on line (19). We first shift the data right by as many bit locations as we need to clear the two other color bytes and the lower three bits of the red data on line (19). We then shift the data left into the location we need it to be, which is a 11 bit locations to the left.
- We take a similar approach on the green data by first shifting it left by two bytes on line (22). We then shift it right by 21 bit locations to get the left-most bit into its final position on line (23). If you're reading this, be the first person to mention it to me and I'll give you a Starbucks gift certificate. We still have data on the right side that we don't want, so we clear that data with a mask on line (25) after loading the mask value on line (24).
- We next process the blue byte by shifting it right to lose the lower-end bits on line (27), then masking all but the good blue bits with an immediate mask on line (28).
- Our final task is to combine the three colors which we have left in x10 (red), x21 (green), and x22 (blue) using two OR instructions on lines (30-31).

```

(00) #-----
(01) # Subroutine: Pack_rgb_24_16:
(02) #
(03) # This subroutine translates 24-bit color data (RGB) to 16-bits by making the
(04) # the red data (left most byte) to 5-bits, the green data (middle byte) to
(05) # 6 bits, and the blue byte (right-most byte) to 5-bits. These of total to
(06) # 16 bits. This approach uses truncation to reduce color values. Data is
(07) # passed to and returned from this subroutine in x10.
(08) #
(09) # Tweaked register: x10,x20,x21,x22
(10) #-----
(11) Pack_rgb_24_16:
(12)
(13) # test data          li    x10,0x00FFFFFF
(14)
(15) init:               mv     x20,x10          # red: make working copies
(16)                   mv     x21,x10          # green
(17)                   mv     x22,x10          # blue
(18)
(19)                   srli   x20,x20,19      # RED: clear right zeros
(20)                   slli   x10,x20,11      # shift back left
(21)
(22)                   slli   x21,x21,16      # GREEN: clear left zeros
(23)                   srli   x21,x21,21      # shift into place
(24)                   li     x20,0x000007E0
(25)                   and    x21,x21,x20     # clear bottom bits
(26)
(27)                   srli   x22,x22,3       # BLUE: shift off bottom 3 bits
(28)                   li     x20,0x0000001F  # mask bottom bits
(29)
(30)                   or     x10,x20,x10     # combind red & blue
(31)                   or     x10,x10,x21     # include green
(32)
(33)                   ret     # bring it on home

```

Figure 15.34: The solution to this example.

Example 15.28: N Factorial

Write a RISC-V assembly language subroutine that converts a calculates $N!$, where N is passed to the subroutine in $x20$. The result is returned in $x30$. Assume the value in $x20$ never will never be so large that the result exceeds the capacity of $x30$. Use the Mult: subroutine listed below in your solution (there is no header to save space). Don't use recursion in your solution.

```

Mult:   mv     x15,x0

loop2:  beq    x11,x0,done1
        add   x15,x15,x10
        addi  x11,x11,-1
        j     loop2

done1:  ret

```

Solution Notes: What would an assembly language programming course be without doing some version of a factorial program. The classic solution uses recursion, but this solution does not. We'll save the recursive solution for another day.

- The subroutine should be saving context, but we did not in order to save space.

- Since is subroutine makes a nested call to the Mult subroutine, we need to save **ra**, which we do by pushing it on the stack on lines (11-12).
- The subroutine first clears the result register, which we do on line (14). We do this because we want to exit the subroutine if the passed value in x20 is zero, which we check for on line (15).
- If the code makes it to line (17), then the passed value of N must be at least 1. At this point, we're prepare to do the multiplication. If the N value is a 1, then we exit the subroutine on line (19) because that conditional fails because we subtracted 1 from N on line (18). If the original N value was '1', the subroutine returns '1' as it is now in x30. This approach provides checks for the N=1 and N=0 cases, which are special cases. If the subroutine continues, we know N was at least 2, and the generic code that follows actually works.
- We prepare the values to send to the Mult subroutine on line (21) and line (23). We use a decremented N value on line (22) to give us the value to multiply by the result. Keep in mind that the value in x30 is the accumulated multiplication result that we eventually return from the subroutine.
- Context is restored by popping **ra** off the stack on lines (29-30).

```

(00) #-----
(01) # Subroutine: N_fact:
(02) #
(03) # This subroutine calculates N-factorial. The value of N is passed to the
(04) # subroutine in x20; the subroutine passes the result back in x30.
(05) #
(06) # Passed values: x20
(07) #
(08) # Tweaked register: x20,x30,x31,x10
(09) #-----
(10) N_fact:
(11) init:  addi  sp,sp,-1          # store return address on stack
(12)        sw   ra,0(sp)
(13)
(14)        mv   x30,x0           # clear register for final result
(15)        beq  x30,x20, done    # check to see if passed val = 0
(16)
(17)        mv   x30,x20         # move current N value to x30
(18) loop:  addi  x31,x20,-1      # move N-1 to x31
(19)        beq  x31,x0,done     # quit if N-1 is zero
(20)
(21)        mv   x10,x30         # prepare to call subroutine
(22)        addi x20,x20,-1      # decrement other subroutine operand
(23)        mv   x11,x20        # put in proper register
(24)        call Mult           # so the multiple
(25)
(26)        mv   x30,x15        # transfer result to accumulator
(27)        j    loop           # go back, check condition
(28)
(29) done:  lw    ra,0(sp)       # restore context
(30)        addi sp,sp,4
(31)
(32)        ret                    # bring it on home
(33) #-----
(34)
(35) #----- header not included to save space -----
(36) Mult:  mv   x15,x0
(37)
(38) loop2: beq   x11,x0,done1
(39)        add  x15,x15,x10
(40)        addi x11,x11,-1
(41)        j    loop2
(42)
(43) done1: ret

```

Figure 15.35: The solution to this example.

And to show that it can be done, and that it's a cool exercise to do so, we can also solve this problem using a recursive algorithm. Figure 15.36 show a recursive solution to this example with the following highlights. This program runs; you should step it through the simulator to see the stack pointer decrement as the recursion becomes deeper and increment as the algorithm exits the recursion. :

- There are three subroutines listed; we only provided a decent header for one of them in an effort to save space.
- The program include checks for $N=0$ and $N=1$, where the answer is 0 and 1, respectively. If the sent value is neither of these numbers, then the algorithm does the recursion thing.
- We must save the return address at all levels, which we do on lines (16-17), and then again on lines (28-20). The associated restorations are done on lines (22-23) and lines (38-39).
- `Nf_rec` is the recursive subroutine. After saving context, the subroutine sets up for the call to the `Multiply` subroutine on lines (31-32), which it does by sending the current result and one less than the current N value to the subroutine. The algorithm exits if the decremented N value is zero on line (33).
- After the `Multiply` subroutine call, the algorithm makes a recursive call on line (36). The key to making recursion work the fact that the N value is decremented at each level of recursion. At some point, N becomes zero and allows the algorithm to break out of the recursion.
- The `Multiply` subroutine is dangerous because it does no checks before it operated; we did this to save space.


```

(00) #-----
(01) # Subroutine: N_fact:
(02) #
(03) # This subroutine calculates N-factorial recursively. The value of N is passed
(04) # to the subroutine in x10.
(05) #
(06) # Passed values: x10
(07) # Returned values: x20
(08) # Tweaked register: x21,x30,x10
(09) #-----
(10) N_fact_recursive:
(11)     mv    x20,x0          # clear result register
(12)     beq  x10,x0,done     # quit if N=0
(13)     li   x20,1
(14)     beq  x10,x20,done   # quit if N=1
(15)
(16)     addi sp,sp,-4        # push return address
(17)     sw   x1,0(sp)
(18)     mv   x20,x10        # move N to x20
(19)
(20)     call Nf_rec         # call N! recursive
(21)
(22)     lw   x1,0(sp)        # restore ra
(23)     addi sp,sp,4
(24) done:  ret              # take it on home
(25) #-----
(26)
(27) #-----
(28) Nf_rec:  addi sp,sp,-4    # push return address
(29)         sw   x1,0(sp)
(30)
(31)         addi x10,x10,-1  # decrement N
(32)         mv   x21,x10    # put new N in x21
(33)         beq  x21,x0,exit # quit if new N=0
(34)
(35)         call Multiply    # do multiply
(36)         call Nf_rec      # recursive subroutine call
(37)
(38)         lw   x1,0(sp)    # restore return address (pop)
(39)         addi sp,sp,4
(40)
(41) exit:   ret              # take it to the home
(42) #-----
(43)
(44) #-----
(45) Multiply: mv    x30,x0    # dangerous multiply routine
(46) loop:   add   x30,x30,x20 # accumulate
(47)         addi x21,x21,-1  # loop admin
(48)         bne  x21,x0,loop
(49)         mv   x20,x30    # move result to x20
(50)         ret              # go back
(51) #-----

```

Figure 15.36: The solution to this example.

Example 15.29: Finding Largest Value in Memory

Write a RISC-V assembly language subroutines that finds the largest value in a given span of memory. The values in the memory start at the address in x10; the number of values to check is given in x20. Store the largest unary value in x25 in binary format. The values in memory are words in stoneage unary format.

Solution Notes: This problem mixes two types of quite popular assembly language programs. First, the program must access memory in a generic manner. Second, the program does some type of number conversion, which in this case is the conversion of stoneage unary to binary. Exciting stuff indeed.

- This problem is ideally suited to a subroutine call for the conversion part of the program. For both subroutines, we provide information-packed headers that make it easy for programmers to safely use the code. We also provide some test code so you can run the test yourself in case you are so inclined.
- The program is structured to have the main subroutine use a while-loop to handle the memory access. The subroutine then uses a nested subroutine **Calc_unary** to convert a stoneage unary value in a register to a binary value. Note that because we use a nested subroutine call, we must save the return address (ra) on the stack before the nested call (lines (22-23)), and then pop it off the stack once the main subroutine is done calling the nested subroutine (lines (34-35)).
- Saving the return address is part of the initialization, the other part is to set up for the algorithm. We want to keep the algorithm generic, so we start the code with the smallest possible value in the x25, which we do on line (21). There are many ways to do this problem; this is probably the most straight-forward, which sounds good to me.
- We then get the data from memory on line (26), send it to the subroutine on line (27), and then conditionally branch based on the result on line (28). If the newly converted value is less or equal to the current largest value, then we continue the loop. Otherwise, we make the currently input value as the new largest value, which we do on line (29).
- The loop administration is for all iterations regardless of whether it was a new large value or not; this includes advancing the address pointer by four because we're using words on line (30), and decrementing the loop count on line (31).
- The **Calc_unary** counts the number of set bits in a register by masking the LSB and accumulating it, an algorithm we've used way too many times up to this point.
- Note that we used labels such as "done1" and "done2" rather than just "done" because we can't reuse the same label in an assembly language program.

```

(00) #-----
(01) # Subroutine: Big_unary:
(02) #
(03) # This subroutine finds the largest storage unary value in a given span of
(04) # memory. The memory starts at the value passed to the subroutine in x10 and
(05) # checks the number of values (words) store in x20. The result is passed back
(06) # to the calling routine x25.
(07) #
(08) # Passed values: x10,x20
(09) #
(10) # Tweaked register: x25, x30, x10
(11) #-----
(12) #---- test code -----
(13) .data                                # data segment
(14) junk:          .word  0x3, 0x7      # dummy data
(15) .text
(16)             la    x10,junk          # load address of junk
(17)             li    x20,2            # load count of data
(18) #---- test code -----
(19)
(20) Big_unary:
(21) init1:      mv    x25,x0            # designated large value
(22)             addi  sp,sp,-4          # make space for ra
(23)             sw    ra,0(sp)         # store return address
(24)
(25) loop1:     beq   x20,x0,done1      # quit if count is zero
(26)             lw    x30,0(x10)       # get value
(27)             call  Calc_unary       # find unary equivalent
(28)             ble   x31,x25,admin    # jump if less than
(29)             mv    x25,x31         # set new greater value
(30) admin:     addi  x10,x10,4         # advance address
(31)             addi  x20,x20,-1       # decrement count
(32)             j     loop1           # repeat
(33)
(34) done1:     lw    ra,0(sp)          # pop return address
(35)             addi  sp,sp,4          # adjust sp
(36)             ret                    # going home, all the time
(37)
(38) #-----
(39) # Subroutine: Calc_unary:
(40) #
(41) # This subroutine converts the unary value in x30 and returns result in x31.
(42) #
(43) # Passed values: x30
(44) #
(45) # Tweaked register: x25, x31, x29
(46) #-----
(47) Calc_unary:
(48) init2:     mv    x31,x0            # init count
(49) loop2:     beq   x30,x0,done2      # see if no more ones
(50)             andi  x29,x30,1        # mask LSB
(51)             add  x31,x31,x29       # accumulate count
(52)             srli x30,x30,1        # shift value 1 to right
(52)             j     loop2           # do it again
(54) done2:     ret                    # bring it home

```

Figure 15.37: The solution to this example.

15.4 C Code-Based RISC-V Programming Problems

This section contains problems that relate to basic C coding principles and constructs. While this is not an exhaustive list, it does contain some of the more basic and important C constructs and subsequently shows their relation to the underlying RISC-V assembly language.

Example 15.30: for Loop

Write a RISC-V assembly language code that implements the following C programming construct. Assume that x10 holds the “A” value, and x13 holds the “B” value.

```
#define VAL 48

for (i = 0; i < VAL; i++) {
    A += B;
}
```

Solution Notes: The code in the example is not a complete program, so the solution is not a complete program either. Both sets of code are examples of code fragments of C code (for the original problem) and assembly code (for the solution). Here are a few items of interest:

- We use a `.equ` assembler directive in an attempt to match the `#define` preprocessor directive in the problem description.
- We modeled the solution as a do-while loop because we knew based on the constant iteration count that we always need to execute the loop at least one time.
- There are many ways we could model this C code, this is one of them. Please be receptive to other solutions.

```
(00) ;-----
(01) ;-- Assembler Directives (somewhere in the program)
(02) ;-----
(03) .equ    VAL,0x30                # constant definition
(04) ;-----
(05) #~~~~~ program fragment ~~~~~
(06) init:   li    x31,VAL            # initialize iterative count
(07)
(08) loop:   add   x10,x10,x13        # do addition: A = A + B
(09)
(10) admin:  addi  x31,x31,-1         # decrement loop count
(11)
(12)        bne   x31,x0,loop        # branch if loop count !=0
(13)
(14)        j     loop              # jump to attempt new iteration
(15)
done:                                           # code breaks out of loop
#~~~~~ program fragment ~~~~~
```

Figure 15.38: The solution to this example.

Example 15.31: for Loop again

Write some RISC-V assembly language code that implements the following C programming construct. Assume x8 holds the “c_cnt” value, x10 holds the “A” value, and x13 holds the “B” value.

```
#define VAL 48

for (i = c_cnt; i < VAL; i+=2) {
    A += B;
}
```

Solution: The code is similar to a previous example; both sets of code are examples of code fragments of C code (for the original problem) and assembly code (for the solution). This problem is eerily similar to the previous problem, so it is important you realize the differences, as they are rather special and somewhat tricky.

- The previous example was a simple iterative loop, where we needed to do something a constant number of times. The loop in this example is not as simple. First, we're not starting the loop count at zero; in this problem, we start it at what we could consider a variable value. Because we do not know what this value could be, we must model this loop as a while loop to ensure that it does not execute the body of the loop not even one time when the conditions are correct. Line (09) in the solution check the loop conditions before it can enter the body of the loop.
- We also need to initialize the loop count in this problem, which we did not do in the previous problem based on the constant and known loop count. We initialize the loop count according to the program description on line (07). The previous problem knew at assemble time how many times the loop would iterate; the code in this problem does not know the iteration count until runtime.
- This program adds two to the loop count each time through the loop, which is also different from the previous problem. We account for that on line (13) in the solution by advancing the count by two.

```

(00) ;-----
(01) ;- Assembler Directives (somewhere in the program)
(02) ;-----
(02) .equ      VAL,0x30                # constant definition
(03) ;-----
(04) #~~~~~ program fragment ~~~~~
(05) init:     li      x31,VAL          # initialize iterative count
(06)         mv      x29,x8           # copy loop start count
(07)
(08) loop:     bge     x29,x31,done     # jump when loop is completed
(09)
(10)         add     x10,x10,x13       # addition: A = A + B (body of loop)
(11)
(12)
(13) admin:   addi    x29,x29,2        # advance loop count count
(14)         j      loop              # jump to attempt new iteration
(15)
(16) done:
(17) #~~~~~ program fragment ~~~~~

```

Figure 15.39: The solution to this example.

Example 15.32: if/else Statement

Write RISC-V assembly language code that implements the following C programming construct. Assume x18 holds “a_val” and x25 holds “sensor_01”.

```
#define C_DUB      192
#define RESET_VAL  65
#define INIT_VAL   80
#define INC_VAL    3

if (a_val == C_DUB) {
    sensor_01 = RESET_VAL;
}
else {
    sensor_01 = INIT_VAL + INC_VAL;
}
```

Solution: As you can see from the problem statement, this is a classic if/else construct. Recall that we have many ways to write if/else constructs, we always do so such that they contain one conditional branch and one unconditional branch. The code in Figure 15.40 shows that as well as some other interesting stuff:

- We once again use assembler directives to encode the values we need to use in the code. The problem used these values as constants, we opt to do the same in our code.
- We use labels to help identify the actual if and else lines in the code. The way we structured the code requires us to use the “else” label, but the “if” label is primarily a comment.
- The if clause assigns a value to a register while the else clause assignment to the same register is a result of an addition instruction.
- The conditional associated with the if clause on line (13) jumps over the if clause to the else when the condition is not true. If the code takes the if, it then unconditionally jumps over the else on line (15).

```
(00) ;-----
(01) ; - Assembler Directives (somewhere in the program)
(02) ;-----
(02) .equ      C_DUB,192                # constant definitions
(03) .equ      RESET_VAL,65
(04) .equ      INIT_VAL,80
(05) .equ      INC_VAL,3
(06) ;-----
(07) #~~~~~ program fragment ~~~~~
(08) init:     li      x10,C_DUB        # put constants into registers
(09)          li      x12,INIT_VAL     #
(10)          li      x13,INC_VAL      #
(11)
(12)
(13)          bne     x18,x10,else     # jump to else if not equal
(14) if:       li      x25,RESET_VAL   # make assignment
(15)          j       done             # jump over else
(16)
(17) else:     add     x25,x12,x13     # jump when loop is completed
(18)
(19) done:
(20)          # code breaks out of loop
#~~~~~ program fragment ~~~~~
```

Figure 15.40: The solution to this example.

Example 15.33: case Statement

Write a fragment of RISC-V assembly language code that implements the following C programming construct. Consider all variables to be declared as unsigned chars. Assume x10 holds “val”, x11 holds “a_val”, x12 holds “b_val”, and x13 hold “c_val”.

```
switch (val)
{
    case 0x01:
        a_val++;
        break;

    case 0x08:
        b_val++;
        break;

    case 0x02:
        c_val++;
        break;

    default:
        a_val = 0;
}
```

Solution: Once again, the code in the example is not a complete program. There are many ways to do this problem; the code below shows one possible and probably solution, with a few fun things to note:

- The problem stated that all variable types were unsigned characters, which is a fact that does not matter for this program. All of the compare operations in the program fragment use registers, which are 32-bit value representations of the C unsigned characters, which are 8-bit values.
- A case statement is simply a special compact form of a string of if/else statements, which is what the code below reflects. The code is of course sequential and it performs one compare at a time. We coded the compares in the order they were given in the program, but the order does not matter. If programmers know the value was most likely to be one of the values, then they would place that compare first in the code; this problem provided no such information.
- This case statement contained a **break** statement for each compare, which is typically the way C uses case statements, but not always. A common C programming error is to not include a break where you actually meant to, which would alter the functionality of the code. This case statement also contained a **default** clause, which is also optional.
- The case statement has three “cases”; there are thus three if/else clauses in the assembly code. We provided slightly helpful labels on lines (01,06,11) with the “cx” terminology in the assembly code.

```

#~~~~~ program fragment ~~~~~
(00) c0:    li    x20,1      # load compare value
(01)      bne   x10,x20,c1  # branch if not equal
(02)      addi  x11,x11,1   # increment x11 (a_val)
(03)      j     done      # jump out of construct
(04)
(05) c1:    li    x20,8      # load compare value
(06)      bne   x10,x20,c2  # branch if not equal
(07)      addi  x12,x12,1   # increment x12 (b_val)
(08)      j     done      # jump out of construct
(09)
(10)
(11) c2:    li    x20,2      # load compare value
(12)      bne   x10,x20,def  # branch if not equal
(13)      addi  x13,x13,1   # increment x13 (c_val)
(14)      j     done      # jump out of construct
(15)
(16)
(17) def:   mv    x11,x0     # clear x11
(18)
(19) done:  # continue with program
#~~~~~ program fragment ~~~~~

```

Figure 15.41: The solution to this example.

Example 15.34: Complex if/else Construct

Write RISC-V assembly language code that implements the following C programming construct. Assume x17 holds x_val, x23 holds sensor_23, x24 holds sensor_24, and x11 holds f_val. Consider all values to be unsigned.

```

#define C_INC      93
#define RESET_VAL  44
#define CLAMP_VAL  156

if (x_val <= (C_INC + f_val) ) {
    sensor_23 = CLAMP_VAL;
}
else {
    sensor_24 = RESET_VAL;
}

```

Solution: This is another if/else statement, but now the condition associated with the if statement is not as simple as the other versions. Programmers can implement if/else statements in many different ways, but there always an approach that minimizes instructions. The designers of the RISC-V instruction set provide six base conditional instructions, and another ten conditional pseudoinstructions based on those six base instructions. This instructional support provides the means for programmers to generate efficient code. Here are the exciting highlights for this problem:

- We use assembler directives to encode the values provided as preprocessor directives in the original code. Both C compilers and RISC-V assemblers have such an option.
- We must formulate the conditions for the conditional statement before we actually use the conditional statement, which essentially means we need to do the addition one of the conditional argument. We do this on line (07) by adding f_val to the provided constant and saving it in another register. We opted to save the value in another register, which was arbitrary; we could have also saved the result of the addition in the x11 register.

- The problem stated that everything was unsigned values, so we opted for a `bgtu` pseudoinstruction on line (09). There are many ways to do the compare; this is the one that felt most clear for us. Encoding statements such as these can become very confusing when you're not used to working with them. Be sure to check over your final approach when you've completed the problem and be sure to check out your approach in a simulator.
- We implemented the if/else clause with one conditional branch statement and one unconditional branch statement. This is the more efficient approach and is one you should always implement when you're writing if/else clauses in your code.
- We opted to use `li` pseudoinstructions in our code instead of `addi` instructions. Keep in mind that the assembler translates the `li` pseudoinstruction into an `addi` instruction, but seeing an `addi` instruction in the code can make human readers think there is an addition operation happening, which is not the case in this code. In this case, we would be using `x0` as one of the operands to the `addi` instruction, which is not really addition.

```

(00) ;-----
(01) ; - Assembler Directives (somewhere in the program)
(02) ;-----
(02) .equ      C_INC,192           # constant definitions
(03) .equ      RESET_VAL,65
(04) .equ      CLAMP_VAL,80
(05) ;-----
(06) ;
(07) init:      addi    x15,x11,C_INC # preliminary math
(08)
(09) compare:  bgtu    x15,x17,if    # start compare value build
(10)
(11) else:      li     x24,RESET_VAL # add f_val
(12)           j      done          # jump over if
(13)
(14) if:        li     x23,CLAMP_VAL # do initial comparison
(15)
(16) done:
(17)           # somewhat meaningful label
;-----

```

Figure 15.42: The solution to this example.

Example 15.35: while Loop

Write RISC-V assembly language code that implements the following C programming construct. Assume `x10` holds `acc_val`, `x11` holds `add_val`, and `x15` holds `count`. Consider the variables to be

```

#define VAL_X 0x77

count = 0;
acc_val = 0;

while (acc_val < VAL_X) {
    acc_val += add_val
    count++;
}

```

Solution: This problem is the classic while loop, well known to check the condition before executing the body of the loop. There are many approaches to solving this problem; the code below shows a good approach in that the loop contains one conditional branch and one unconditional branch.

- The code uses an assembler directive to encode the preprocessor directive in the original problem description.
- The code has an “init” section, where it set a value in a register (06), the sets the value of two registers to zero lines (07-08).
- The while loop first check the contition on line (10); it the condition is not true, the body of the loop does nto execute.

```

(00) ;-----
(01) ; - Assembler Directives
(02) ;-----
(03) .equ   VAL_X,0x77
(04) ;-----
(05)
(06) init:  li    x20,VAL_X      # put value in register
(07)        mv    x10,x0        # clear acc_val
(08)        mv    x15,x0        # clear count
(09)
(10) loop:  bltu   x10,x20,done  # check condition
(11)        add   x10,x10,x11    # body of loop, do add
(12)        addi  x15,x15,1      # increment count
(13)        j     loop          # jump to check condition
(14)
(15) done:                # onto other good things

```

Figure 15.43: A possible solution for this example.

Any time we write assembly code, we should always wonder whether there is a more efficient way to code things. The following solution represents the output of such thoughts. In this solution, we attempt to use a sltiu instruction in an effort to make the code more efficient. As you can see the code has the same number of instrutions, so the second approach is not more space efficient. The while loop now uses to instructions to examine the condition on lines (09-10). Because the extra instruction is part of the loop, the second solution requires more instructions to execute, and is thus less runtime efficient. Nice try, though.

```

(00) ;-----
(01) ; - Assembler Directives
(02) ;-----
(03) .equ   VAL_X,0x77
(04) ;-----
(05)
(06) init:  mv    x10,x0        # clear acc_val
(07)        mv    x15,x0        # clear count
(08)
(09) loop:  sltiu  x20,x10,VAL_X # check condition
(10)        beq   x20,x0,done    # branch if condition fails
(11)        add   x10,x10,x11    # body of loop, do add
(12)        addi  x15,x15,1      # increment count
(13)        j     loop          # jump to check condition
(14)
(15) done:                # onto other good things

```

Figure 15.44: Another possible solution for this example.

Example 15.36: The Classic do-while Loop

Write RISC-V assembly language code that implements the following C programming construct. Assume x10 holds acc_val, x11 holds add_val, and x15 holds count.

```
#define VAL_X 0x77

count = 0;
acc_val = 0;

do {
    acc_val += add_val;
    count++;
}
while (acc_val < VAL_X);
```

Solution: This problem is purposely similar to the previous problem. The previous problem was a while loop but this problem is a do-while loop. We provided them for their pure comparison value.

- As you can see, the code in the following solution is rather interesting because the do-while loop with similar statements executes using one less instruction. Keep in mind that although the problems appear the same, they operate inherently differently in the code because one is a while loop and the other is a do-while loop. So the moral of this story is that if you know your loop always executes at least once, model it as a do-while loop and save an instruction, which means your loop executes in a shorter amount of time.

```
(00) ;-----
(01) ; - Assembler Directives
(02) ;-----
(03) .equ    VAL_X,0x77
(04) ;-----
(05)
(06) init:  li    x20,VAL_X      # put value in register
(07)        mv    x10,x0        # clear acc_val
(08)        mv    x15,x0        # clear count
(09)
(10) loop:  add    x10,x10,x11   # body of loop, the add
(11)        addi   x15,x15,1    # body of loop, increment count
(12)        bltu  x10,x20,loop  # check condition, loop if necessary
(13)
(14) done:  # onto other good things
```

Figure 15.45: A possible solution for this example.

Example 15.37: C-Type memcpy Function

Write an assembly language subroutine that implements a C `memcpy()` function. See the C definition for a `memcpy()` below. For this subroutine, assume that `s1` is provided in `x11` and `s2` is provided in `x12`, respectively; the value of `n` is provided in `x10`. Your function should copy `n`-bytes of data starting at the RAM location specified in `x11` to the RAM locations specified in `x12`. For this problem, you can assume the `n`-bytes value is small enough not to cause any problems. Make your code as efficient as possible.

```
memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

The `memcpy()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`.

Solution: This is a standard C function that copies memory from one area in memory to another. We solve this two different ways. The first way is straightforward but sort of mechanical in that we did not think it out before writing the code. We took what we learned writing the code and rewrote the code for the second solution. Here are the highlights of the first solution:

- The approach we take is to separately find the count of words, halves, and bytes. The second half of the subroutines then uses those counts to read from one area in memory to another using words, halves, then bytes. This is a real generic and certainly non-clever approach.
- We use two while loops for finding the word and half count; the value that remains in `x10` is then the byte count. We essentially repeated the first while loop for the second while loop and changed a few key values.
- The second part of the subroutine uses the counts in the first part of the program to read data from one location and copy it to the other. It uses three while loops in the same way as the first part of the subroutine used two while loops.

```

(00) #-----
(01) # Subroutine: memcpy:
(02) #
(03) # This subroutine stores a chunk of data where the size of the chunk is
(04) # passed to the subroutine in x10. This subroutine does it in the most
(05) # efficient way (fewest writes) possible.
(06)
(07) # Passed values x10 (size of a data chunk)
(08) # Passed values x11 (address of data to copy)
(09) # Passed value: x12 (address to copy data to)
(10) # Return values: none
(11) # Tweaked registers: x20,x21,x30,x10,x25,x10,x11,x12
(12) #-----
(13) memcpy:
(14) li      x10,12
(15)
(16) init:   mv      x20,x0      # clear word counter
(17)        mv      x21,x0      # clear halfword counter
(18)
(19)        li      x30,4       # load size of word
(20)
(21) word:   bltu   x10,x30,wdone # branch if no more words
(22)        addi   x20,x20,1     # increment word count
(23)        addi   x10,x10,-4    # reduce by word size
(24)        j      word         # do again
(25)
(26) wdone:  srli   x30,x30,1    # divide word size by 2
(27)
(28) half:   bltu   x10,x30,st_words # branch if no more words
(29)        addi   x21,x21,1     # increment halfword count
(30)        addi   x10,x10,-2    # reduce by word size
(31)        j      half         # do again
(32)
(33) st_words: beq    x20,x0,st_halfs # branch to half store is zero
(34)        lw     x25,0(x11)      # get a word
(35)        sw     x25,0(x12)      # store at new address
(36) admin1:  addi   x20,x20,-1    # decrement word count
(37)        addi   x11,x11,4      # advance mem address by word
(38)        addi   x12,x12,4      # advance mem address by word
(39)        j      st_words      # repeat
(40)
(41) st_halfs: beq    x21,x0,st_bytes # branch to half store is zero
(42)        lh     x25,0(x11)      # get a half
(43)        sh     x25,0(x12)      # store at new address
(44) admin2:  addi   x20,x20,-1    # decrement word count
(45)        addi   x11,x11,2      # advance mem address by word
(46)        addi   x12,x12,2      # advance mem address by word
(47)        j      st_halfs      # repeat
(48)
(49) st_bytes: beq    x10,x0,done   # branch to half store is zero
(50)        lb     x25,0(x11)      # get a word
(51)        sb     x25,0(x12)      # store at new address
(52) admin3:  addi   x10,x10,-1    # decrement word count
(52)        addi   x11,x11,1      # advance mem address by word
(54)        addi   x12,x12,1      # advance mem address by word
(55)        j      st_bytes      # repeat
(56)
(57) done:    ret

```

Figure 15.46: The unthoughtout solution to this example.

The second version of the solution uses a much more intelligent and thus more efficient solution. Here are some of the highlights:

- We don't break the program into two parts; we instead copy as we need to. We don't find the count of how many words and halves to copy in advance, we mostly copy them on the fly. Additionally, we note that the after the algorithm deals with the words, there is only a possibility

of one or zero halves to copy, and one or zero bytes to copy. Knowing this allows us to not use while loops for copy halves and bytes, we use if/else statements instead.

- The overall runtime increases for the second algorithm. There is less code as well, and the code uses less registers. Normally we would save and restore context in these examples, but that would make the code even longer. The fact that the second algorithm uses less registers means that if we chose to save/restore context, we could do it much faster than the first algorithm because that one used more registers.

```

(00) #-----
(01) # Subroutine: memcpy:
(02) #
(03) # This subroutine stores a chunk of data where the size of the chunk is
(04) # passed to the subroutine in x10. This subroutine does it in the most
(05) # efficient way (fewest writes) possible.
(06)
(07) # Passed values x10 (size of a data chunk)
(08) # Passed values x11 (address of data to copy)
(09) # Passed value: x12 (address to copy data to)
(10) # Return values: none
(11) # Tweaked registers: x30,x25,x31,x10,x11,x12
(12) #-----
(13) memcpy:
(14) li      x10,15
(15)
(16)         li      x30,4          # load size of word
(17)
(18) word:   bltu   x10,x30,wdone   # branch if no more words
(19)         lw      x25,0(x11)     # get a word
(20)         sw      x25,0(x12)     # store at new address
(21) admin1: addi   x10,x10,-4      # decrease word count
(22)         addi   x11,x11,4       # advance mem address by word
(23)         addi   x12,x12,4       # advance mem address by word
(24)         j      word           # repeat
(25)
(26) wdone:  srli   x30,x30,1       # divide word size by 2
(27)
(28) half1:  bltu   x10,x30,bytel   # branch if no more halves
(29)         lh      x25,0(x11)     # get a half
(30)         sh      x25,0(x12)     # store at new address
(31)         addi   x10,x10,-2      # decrease size counter
(32)
(33) bytel:  beq    x10,x0,done      # branch if no bytes
(34)         lb      x25,0(x11)     # get a byte
(35)         sb      x25,0(x12)     # store at new address
(36)
(37) done:   ret

```

Figure 15.47: The well-thoughtout solution to this example.

15.5 Chapter Summary

- This chapter contained many example programs that show many common techniques to assembly language programming. The chapter started with easy problems that became; the chapter problems became more complicated as the chapter progressed.
 - The programming areas in this chapter include introductory problems, more complicated problems, and C programming-based problems. Yes, lots of happy stuff embedded in those many solutions.
 - The RISC-V uses memory-mapped I/O, which results in input/output using the load-type/store-type memory access instructions.
-

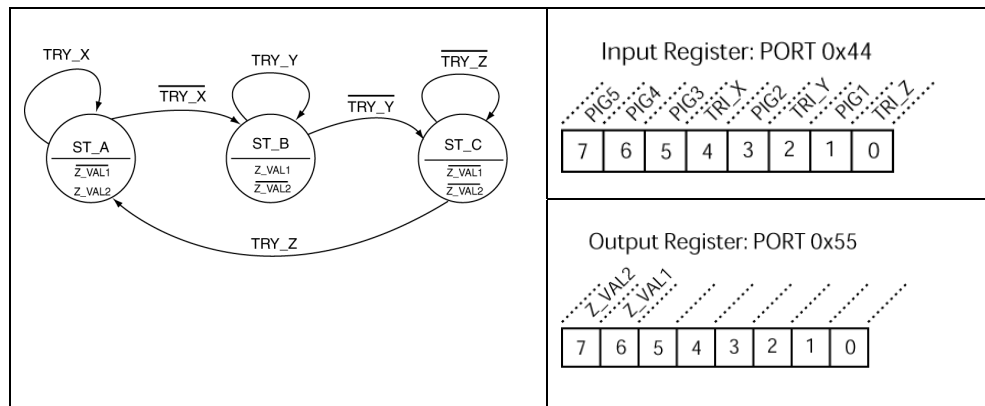
15.6 Chapter Exercises

- 1) Briefly describe why we always write RISC-V assembly language programs using endless loops.
 - 2) Briefly describe what would happen if our RISC-V program could not be characterized by an endless loop.
 - 3) Briefly describe which type of divisions/multiplications can be done very efficiently using the RISC-V instruction set.
 - 4) Briefly describe if right-shifting by two results in a truncated or rounded-up number.
 - 5) What is the largest digital number that a RISC-V register can represent using a BCD format?
 - 6) Briefly describe how parity is typically determined in hardware.
 - 7) The *bubble sort* algorithm is famous for having nested loops; briefly describe what this means in the context of assembly language programming.
 - 8) Briefly describe why you should always attempt to use do-while loops rather than while loops for iterative constructs.
 - 9) Compilers have many ways to translate higher-level language code into assembly code. Briefly describe how you would know if the compiler is performing a correct and/or efficient job.
 - 10) Briefly describe the use of a *flag register*.
-

15.7 Chapter Programming Problems

For the following problems:

- Minimize the amount of code in your solutions
 - Make your code look exquisite in terms of readability
 - Fully comment your code, including proper headers for subroutines
- 1) Write a RISC-V MCU assembly language subroutine that counts that examines and possibly modifies a value in memory. The memory location in question is stored in `x20`. If the value at that location is even, then the number is multiplied by four and stored back at the same address; otherwise the value is divided by two and stored at the same address. Don't worry about overflow and underflow for this problem
 - 2) Write a RISC-V MCU assembly language program that does the following (assume the associated hardware includes 16 switches at port address `0x11008000` and 16 LEDs at port address `0x1100C000`): the program toggles the right-most LED each time the state of the left-most switch changes. When the program detects that change in switch value, it copies 100 bytes of data from the memory address starting at `0x0000D000` to the addresses starting at `0x0000E000`. If the switch value is currently on, the data is copied directly; otherwise, a two's complement of the data is copied.
 - 3) Write a RISC-V assembly language subroutine that clamps a span of 8-bit unsigned binary number in memory into the range `[33,233]`. This means if the number is in the given range, it is not altered. If the number is less than the lower bound, the number is clamped to the lower bound. If the number is greater than the upper bound, the number is clamped to the upper bound. The binary value is provided in `x20`; the beginning of the range is given by the address in `x25`, and the number of values to clamp is given in `x30`.
 - 4) Write a RISC-V fragment of assembly code that performs a firmware-based debounce of a button. The button is the right-most bit of the data from port address `0x11008004`. Have the fragment call a subroutine `Delay_bounce`, but don't bother defining that subroutine. The bounce should be associated with a `0→1` transition.
 - 5) Write a RISC-V fragment of assembly code that performs a firmware-based debounce of a button. The button is the right-most bit of the data from port address `0x11008004`. Have the fragment call a subroutine `Delay_bounce`, but don't bother defining that subroutine. The bounce should be associated with a `1→0` transition.
 - 6) Write a RISC-V assembly language program the implements the following FSM.



PART FIVE: RISC-V OTTER MCU Hardware Matters



16 RISC-V Architecture Details

16.1 Introduction

All of the previous chapters that dealt with the RISC-V MCU did so at primarily a programming level. We purposely limited our mention of hardware details in an effort to not frighten programmers who have no knowledge of the hardware implements an actual computer. This chapter delves into those details by describing the underlying hardware details of the RISC-V MCU's submodules at both a low and high-level context. The notion here is that that act of executing an instruction makes certain things happen in the underlying RISC-V MCU hardware. In other words, there are certain actions the RISC-V MCU's hardware must take to correctly implement any given instruction. This chapter describes the RISC-V various submodules and their relation to the execution of instructions in the RISC-V MCU's instruction set.

Main Chapter Topics

- **DESCRIPTION OF RISC-V MCU'S SUBMODULES:** This chapter describes the various submodules in the RISC-V MCU architecture. These submodules include the control units, the program counter, the main memory, the branch address generator, the immediate generator, the branch condition generator, and the ALU.
- **HARDWARE DETAILS OF INSTRUCTION EXECUTION:** This chapter provides pertinent hardware details regarding the execution of instruction.
- **OVERVIEW OF THE RISC-V MCU WRAPPER:** This chapter describes the "wrapper" which we use to interface the RISC-V MCU with external hardware such as a development board or other modules.

Why This Chapter is Important

This chapter is important because it describes the low-level architecture details of the RISC-V MCU and its interfacing to the outside world with particular attention to instruction execution.

16.2 The Big RISC-V MCU Overview

The RISC-V MCU is simply a relative large and relatively complex digital circuit that has the ability to run programs. It can run roughly any program written using RISC-V assembly language, which means it's quite versatile. Because it has the ability to run programs, we refer to this circuit as a computer, or probably better stated, as a microcontroller (MCU).

The RISC-V OTTER MCU has a level of complexity that makes it tough to understand as one large circuit. The only way we (or at least me) can understand this circuit is to subdivide it into various modules. This act of subdividing large circuit is one of the primary characteristics of modern digital design in that modeling the circuit in a hierarchy facilitates the understanding of how the circuit operates. Because we are in the hardware portion of this text, we need to understand absolutely everything about this circuit, and thus why it is we are able to refer to it as a computer. Be sure to note that pure programmers don't require the same level of understanding as hardware designer; programmers are only responsible for writing programs. There are a world full people who can program computers, but a whole lot less people who understand the hardware the programs execute on. I'm glad I'm a hardware person who knows how to write efficient programs. The relation here is not obvious, but so I'll state it plainly, Mealy's First and Only Law of Computer Programming (sorry pure programmers... it's true):

Mealy's First and Only Law of Computer Programming: If you understand the hardware of the computer your program will run on, then you can write better programs.

In terms of the various operations computers perform, things don't come for free. You use a MCU to solve problems by writing programs; the computer executes the instructions in your program in order to solve the problem. Executing instructions takes time, eats power, and generally speaking, you must solve every problem of interest using some sort of algorithm. If you understand the instruction set from a low level, you can write programs that are more efficient because you understand how to use the instructions in the given instruction set in an efficient way and also know how to not use instructions in inefficient manners. Pure programmers are not privy to the details.

Figure 16.1 show a high-level view of the RISC-V MCU, which includes a listing of its main submodules. This chapter individually describes most of these submodules in the sections that follow. We do, however, save the description of the CSR module for the chapter describing the RISC-V MCU interrupt architecture.

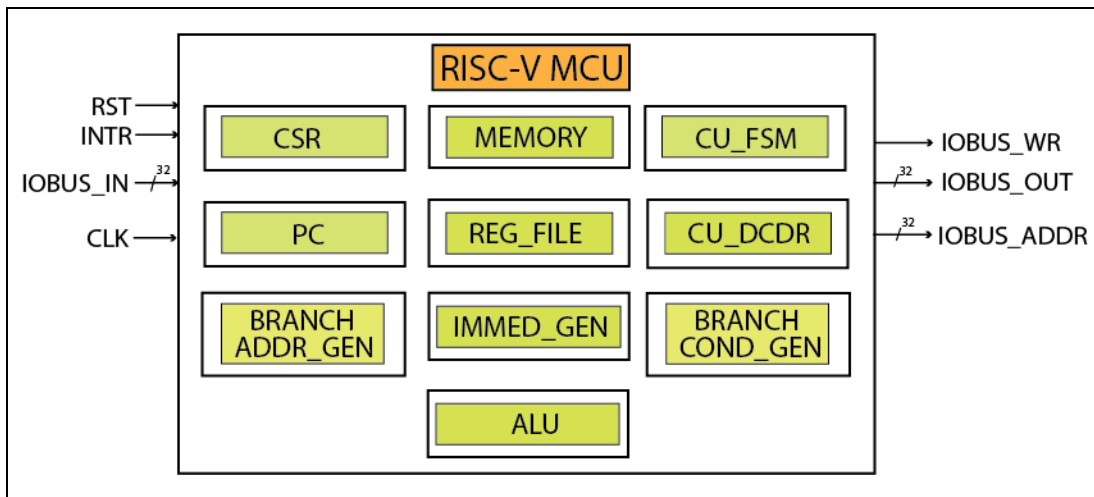


Figure 16.1: A high-level view of the RISC-V MCU and its submodules.

16.3 The Control Units

The current implementation of the RISC-V MCU uses two control-type modules what refer to as the control unit FSM (CU_FSM) and the control unit decoder (CU_DCDR). The two modules control the low-level operations of the RISC-V MCU. As their names' imply, the CU_FSM is truly an FSM, which means it's a sequential circuit, while the CU_DCDR is a decoder, meaning it is a combinatorial circuit. Keep these distinctions in mind in the following discussion.

It is an arbitrary design decision to separate the control unit into two modules. There is no reason preventing us hardware designers from implementing both units in the same module. The thought is that we can comfortably classify the two outputs from the control unit modules into two categories, so we opt to do so to help simplify the understanding of the overall RISC-V operation. Additionally, the description in this section lists but does not describe in any meaningful detail the signals or functionality associated with interrupts. We'll add the required signals as supporting hardware in Chapter 18.

16.3.1 The Control Unit FSM (CU_FSM)

The basic operation of the RISC-V MCU is to sequentially execute instructions stored in program memory. Because the execution of an instruction does not occur in "one step"¹, we need an FSM to provide the control necessary to implement instructions in a specific sequence. If we could execute instruction in one step, we could probably get away with only have a decoder control the operation of the computer. In other words, the execution

¹ Here "one step" means one clock cycle; this is a topic we discuss in an upcoming section.

of an instruction is a multi-step process; we synchronize each of the steps in the process with an active clock edge. Recall that the underlying RISC-V hardware comprises of a significant number of sequential circuit elements, which generally means the operation of these elements depends on and are synchronized to an active clock edge in the circuit.

Most instructions in the RISC-V ISA require two clock cycles for execution, though the load instructions require three clock cycles. In essence, the execution of a program involves the repeated processing of these clock cycles. Other literature on computer architecture refers to these cycles as “T cycles”. These basic clock cycles are important so we give them names, which makes it easier to discuss them. The three clock cycles that we use in the RISC-V OTTER are 1) the fetch cycle, 2) the execute cycle, and for the load instructions only, 3) the writeback cycle. Roughly speaking, the fetch cycle involves “fetching” an instruction from program memory, the execute cycle involve executing the instruction, and the writeback cycle involve writing data from an external source to the register file. More details on these later.

The main responsibility of the CU_FSM is to sequence though the various cycles to implement the instructions. Another way to view the CU_FSM’s responsibility is to control the flow of data through the underlying hardware, a task that it does by sending out the required control signals during each cycle. As you would imagine, control functionality such as this is ideally suited for a finite state machine (FSM).

Figure 16.2 shows the black box diagram for CU_FSM. The signals on the left side of the module are effectively status signals (not including the clock signal), while the signals on the right side are control signals. The FSM basically reacts to the status inputs and sends out the appropriate control signals. Table 16.1 provides a brief description of the signals in the CU_FSM interface.

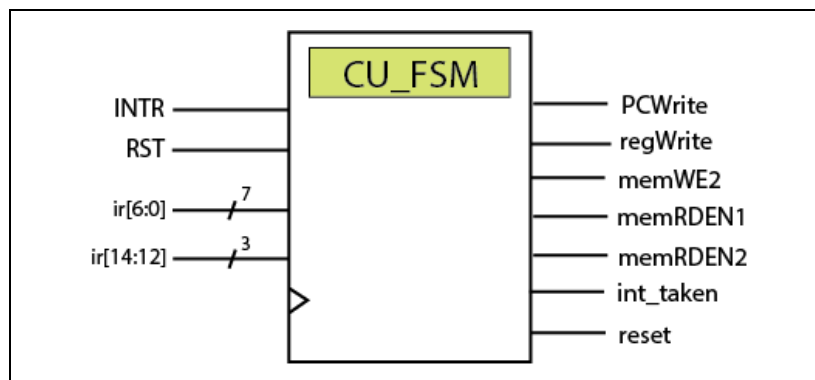


Figure 16.2: The Control Unit FSM black box diagram.

Signal	Type	Comment
INTR	in	An external asynchronous active high interrupt signal that the FSM uses to determine if an active interrupt is pending or not.
ir[6:0]	in	These are the lower seven bits of the instruction word, which serves as the opcode field shared by all instructions.
ir[14:12]	in	These are three bits that serve as the funct3 opcode share by some instructions.
RST	in	An external asynchronous reset signal that the FSM synchronizes and uses to send out reset via the reset signal.
clk	in	The system clock (not shown in Figure 16.2), a rising-edge-triggered signal.
PCWrite	out	Controls the loading of data into the program counter (PC).
regWrite	out	Controls the loading of data into the register file.
memWE2	out	Controls loading of data (writing) into main memory.
memRDEN1	out	Controls the reading of instruction data from main memory (output read enable).
memRDEN2	out	Controls the reading of generic data from main memory (output read enable).
int_taken	out	Controls other modules (CU_DCDR & CSR) handling of interrupts
reset	out	Controls synchronous resetting of the program counter (PC)

Table 16.1: Description of CU_FSM inputs and outputs.

Figure 16.3 shows the state diagram modeling the CU_FSM. This diagram shows three main states, but shows only one external input signal: RST. The state diagram shows a LOAD signal, but this is effectively an internal “condition” and not an external status signal. The first thing to notice about the state diagram is that it is missing all of the control signals (outputs) listed in Table 16.1. It is not conceivable to make the state diagram complete based on the basic functional requirements of the FSM acting as a controller for the RISC-V MCU. We’ll discuss those details soon. The LOAD label in Figure 16.3 signifies that the RISC-V hardware executes all instructions in two clock cycles (fetch & execute), except for the load-type instructions, which require the writeback cycle to complete execution.

The state diagram in Figure 16.3 is actually not complete because we omitted all description of interrupt-based operations. Part of the mechanism includes the **int_taken** signal, which we list in Table 16.1. We’re saving the details of the interrupts architecture until Chapter 18, where we happily fill in all the gory details.

The high-level description of the CU_FSM responsibilities is relatively simple. The fetch cycle retrieves an instruction from program memory. The instruction itself is comprised of field codes and opcodes. The right-most opcode connects to the CU_FSM; the CU_FSM uses this opcode to determine which instruction requires execution. We officially say that the CU_FSM “decodes” the opcode to determine which instruction is being executed, and then sends out the appropriate control signals to “make that instruction happen” in the underlying hardware.

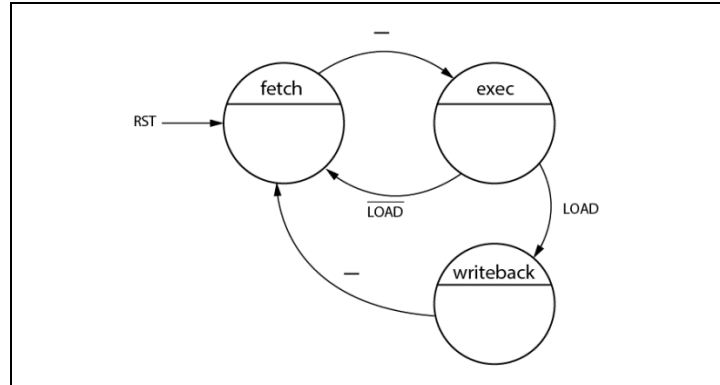


Figure 16.3: The state diagram modeling the Control Unit FSM (no interrupts).

Figure 16.4 shows how the system clock delineates the fetch and execute cycles. Figure 16.4 assumes that RISC-V circuit elements are rising-edge triggered, and that the instructions are non-load-type. Figure 16.4 shows two and one half instructions cycles.

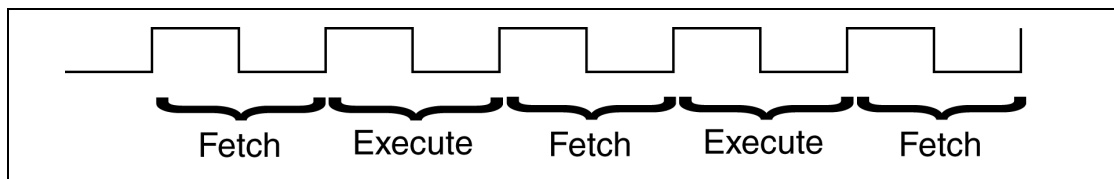


Figure 16.4: An example cycle sequence for executing non-load-type instructions.

16.3.1.1 Individual FSM States

Each of the three states in the FSM has distinctive responsibilities. We'll describe those responsibilities in general terms in this section in order to provide an intuitive notion of what each state does and the part they play in implementing instructions.

The Fetch Cycle: The FSM's fetch state implements the fetch cycle. This state's single responsibility is to retrieve, or *fetch*, an instruction from program memory. Program memory is a part of main memory; all main memory reads are synchronous. Thus, the one responsibility of the fetch cycle is to assert the read signal associated with program memory.

The Execute Cycle: The FSM's execute cycle has several responsibilities, which is why we sometimes refer to this cycle as the “decode/execute” cycle. The decode/execute moniker is a better name but the name “execute” is far easier to write. As the state diagram indicates, the FSM unconditionally enters the execute cycle after the fetch cycle, which is a fact that never changes in the RISC-V MCU architecture. The fetch cycle provides (output of main memory) the machine code for the instruction the next instruction to execute. The various opcode fields in individual instructions indicate which instruction requires execution. The CU_FSM and the CU_DCDR examine the instruction opcodes (in other words, “decode” the opcodes) and then send out the correct control signals to implement the instruction on the underlying hardware. By “implement”, we mean make the underlying hardware perform the operation requested by the instruction's opcodes in conjunction with the field codes in the instruction. The execute cycle is relatively complex in that it needs to “decode” about 40 instructions and then send out the appropriate control signals. Another way to look at instruction execution is that the control units are directing the flow of data through the hardware so as to implement the operation specified by the instruction.

The Writeback Cycle: The FSM's writeback cycle is associated with only the load-type instructions. If the instruction the MCU is implementing is a load-type instruction, the FSM transitions from the execute state to the writeback state; for all other instructions, the FSM

transitions from the execute state back to the fetch state. We can describe the load-type instruction operations execute state as needing to generate a memory address and the writeback state as using that address to read data from main memory. The hardware simultaneously writes the data read from main memory into the register file and transitions back to the fetch state.

Example 16.1

How many clock cycles does it require for the following RISC-V assembly language code fragment to execute from the starting at the **start** label and going through the **done** label?

```

start:   add    x10,x0,x0
         addi   x10,x10,4
         sub    x13,x11,x12

loop:    beq    x10,x0,done
         lw     x20,0(x21)
         lw     x21,8(x22)
         sw     x21,4(x23)
         addi   x10,x10,-1
         j     loop

done:    nop

```

Solution: This is a classic problem that requires you to understand both iterative constructs and how the RISC-V hardware implements instructions. The first thing you need to do in these problems is to examine the code to look for iterative constructs. This code had a loop in it so this problem is not a matter of counting instructions, you must also consider how many times the code runs through the loop and what instructions on in the body of the loop.

There are two things to be aware of in problems such as this. First, we need to look for load-type instructions, which are important because they require three clock cycles to execute. Second, you need to look for iterative constructs, which there almost always is simply to make these problems more exciting (and less boring).

The value in x10 controls how many times the loop iterates, which we can see from the first two instructions. Register x10 is first cleared, then advanced by four, so the number of loop iterations is four. The loop itself has six instructions, which the loop executes each of the four times through the loop. There are two load-type instructions in the loop, so they require three clock cycles. All the instructions in the body of the loop execute four times, but the instruction that tests the loop count (the **beq** instruction) executes one more time than the number of times the loop body executes. Here is the painful gathering of information for the solution, we gather the final solution from adding the values in the third column in Table 16.2: A painfully detailed description of the solution., which is 66 clock cycles.

Instructions	Number of Clock Cycles	Total	Comments
The first three	6	6	3 @ 2 clock cycles
The loop	4 * (2 + 3 + 3 + 2 + 2 + 2)	56	4 times @ 14 clock cycles
The beq	2	2	Last time the loop
The last one	2	2	The final instruction

Table 16.2: A painfully detailed description of the solution.

Example 16.2

How many clock cycles does it require for the following RISC-V assembly language code fragment to execute from the starting at the **start** label and going through the **done** label?

```

start:    add    x10,x0,x0
          addi   x10,x10,8

loop:     slli   x10,x23,2
          lw     x20,0(x21)
          sw     x22,12(x28)
          sw     x21,4(x23)
          addi   x10,x10,-1
          bne   x10,x0,loop

done:     sub    x23,x24,x25

```

Solution: At first glance, this problem looks similar to the previous problem, but looks can be deceiving, particularly when you’re really tired. The previous problem contained a while loop, whereas this problem contains a do-while loop. What this means is that we need to be careful about which instructions are part of the loop administration as this example is different from the previous example.

Similar to the previous problem the first two instructions in this example establish the iteration count. The body of the loop has six instructions, one of which is a load-type instruction; this means the body of the loop requires 13 clock cycles to execute. The final instruction adds two more clock cycles.

The cool thing to note in this problem is that the do-while loop only has one loop administrative instruction per iteration. The while-loop in the previous example had two such instructions. Also, the do-while loop does not include that “extra” instruction, which the previous problem required to test the loop condition.

Instructions	Number of Clock Cycles	Total	Comments
The first two	6	4	2 @ 2 clock cycles
The loop	8 * (2 + 3 + 2 + 2 + 2 + 2)	104	8 times @ 13 clock cycles
The last one	2	2	The final instruction

Table 16.3: Another relatively painful description of the solution.

16.3.2 The Control Unit Decoder

The RISC-V implements the control unit using two modules. The CU_DCDR works conjunction with the CU_FSM to implement instructions in the underlying hardware. As the name implies, the CU_DCDR is a type of decoder, which means it’s a combinatorial circuit. Each of the outputs from the CU_DCDR connects to MUX select inputs in other parts of the RISC-V MCU hardware. Figure 16.5 shows the high-level interface of the CU_DCDR module; Table 16.4 provides a brief description of the module’s inputs and outputs.

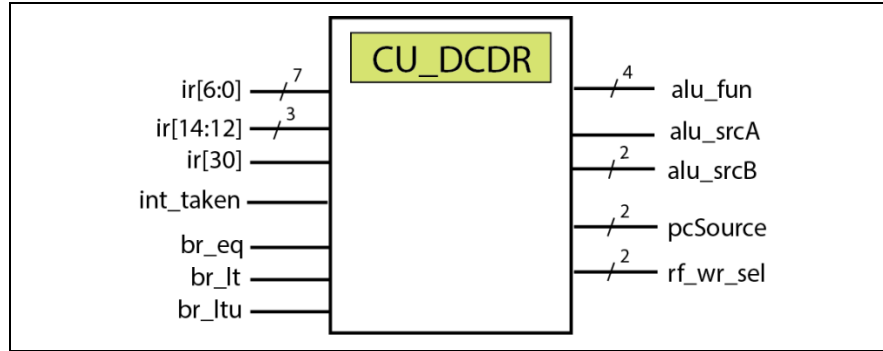


Figure 16.5: Black box diagram of the CU_DCDR module.

Every input to the CU_DCDR is part of one of the three opcode fields in the various instruction formats. Decoding all instructions is a relatively simple and structured process based on the instruction opcodes. There can be up to three levels of decoding for each instruction, where the three levels correspond to the three possible opcode fields. The first step is to examine the **opcode** (ir[6:0]), which roughly determines the type of instruction. The CU_DCDR can decode some instructions using only the **opcode** field, but most instructions require at least the **funct3** opcode field (ir[14:12]) as well. The second step, when necessary, requires the CU_DCDR to use the **funct3** opcode field to further decode the instruction. A few instructions require a third step in the decoding process, which entails the use of the **funct7** opcode field. The CU_DCDR only requires one bit of the **funct7** field (ir[30]) for instruction decoding, which is why we don't route the other six bits of that field to the module. Note that not all instructions require all three decoding steps.

Being that the CU_DCDR is a decoder, it is always outputting data. The only time this data is meaningful is after the valid opcodes become available after entering the execute cycle. Exiting the fetch cycle includes a synchronous read of the program memory data, which includes the instruction bits. Entering the execute cycle makes the opcode bits available to both the CU_FSM and the CU_DCDR. Once the CU_DCDR decodes the valid opcode during the execute cycle, the control outputs of the CU_DCDR become valid.

The CU_DCDR effectively has two modes of operation: 1) decoding instructions, and 2) decoding interrupts. As for the decoding of instructions, all non-interrupt decoding is similar, including the decoding of the three-state load-type instructions. Once the valid instruction bits become available the execute cycle, they remain on the CU_DCDR's output until the execute cycle of the next instruction. The CU_DCDR must also recognize when the CU_FSM is acting on an interrupt, which it does by examining the **int_taken** signal.

Signal	Type	Comment
ir[6:0]	in	7 bits of the instruction register, forming the opcode field in all instructions
ir[14:12]	in	3 bits of the instruction register forming the funct3 opcode field in instructions
ir[30]	in	1 bit in instruction register, part of the funct7 opcode bits
int_taken	in	1-bit signal indicating MCU entered an interrupt cycle
br_eq br_lt br_ltu	in	Three 1-bit signals used by conditional branch instructions to determine appropriate program flow control actions.
alu_fun	out	Controls the selection of ALU operations (4 bits)
alu_srcA	out	Controls the selection of the source A (rs1) ALU operand (1 bit)
alu_srcB	out	Controls the selection of the source B (rs2) ALU operand (2 bits)
pcSource	out	Controls the selection of the address data for loading into the PC (2-bits)
rf_wr_sel	out	Controls the selection of data for loading into register file (2-bits)

Table 16.4: Description of CU_DCDR inputs and outputs.

16.4 The Program Counter (PC) (no interrupt support)

The program counter, or “PC”, is probably the most common sub-module in computer architecture. The PC’s basic responsibility in a computer architecture is providing a pointer (address) to an instruction in program memory. The official definition of the PC is that it holds the address of the instruction in program memory that the MCU is currently executing. As you’ll soon see, the correctness of this definition depends underlying timing considerations, which we’ll discuss in more detail later. More specifically, the PC points either to the currently executing instruction or to the instruction following the instruction that is currently executing.

Figure 16.6 shows a high-level block diagram of the PC in the RISC-V MCU. The first thing you may notice about this diagram is that we don’t implement the PC as a counter; we instead implement it as a register. While a counter is a type of register, we choose to use a register for the PC because we are only loading values into the PC; we are never actually doing an increment operation as we would do in a normal counter. Even though it may cause initial confusion, we’ll keep referring to this module as the “PC”.

Being the PC is only a simple register, it only has typical register inputs such as clear, load enable, data, and a clock input. We opt to include some external circuitry as part of what we consider the PC, which includes both a MUX and an adder. As we indicate in Figure 16.6, we refer to the entire module as the PC_MOD. Here are the interesting things to note about Figure 16.6.

- The MUX in the PC_MOD includes some select functionality beyond a simple register. The CU_DCDR controls the select inputs to the MUX.
- The adder is the box in Figure 16.6 that contains the “+4” label. While this notation is really handy, you must realize that using this notation essentially means that you’ll need some type of adder to implement the box.
- The register heart of the PC is 32-bits wide; the MUX data inputs are also 32-bits wide.
- The PC’s responsibility is to provide an address to program memory of an instruction that requires execution. We thus arbitrarily reduce the actual address lines from 32 to 14 using the reduction box (the box with the “-“ in it). We reduce the number of bits by removing the two most significant bytes and the two least significant bits. We can remove the top two bytes because the current RISC-V MCU OTTER memory is limited to two bytes worth of memory space. We can remove the two least significant bits because program memory is byte-oriented and every RISC-V instruction is four byte long.

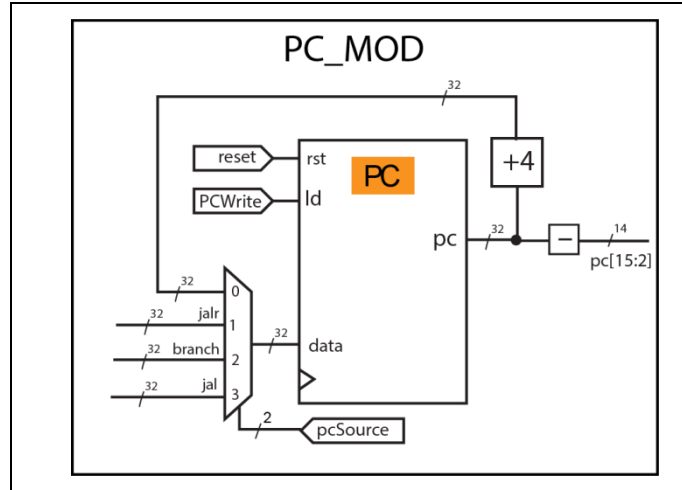


Figure 16.6: The Program Counter block diagram.

16.4.1 PC Inputs and Outputs

There are relatively few connections to the PC; Table 16.5 lists the PC interface as including a description of the pertinent signals. There is truly nothing special about the PC: it's just a relatively simple register. We'll describe the functionality associated with the MUX in a future section in this chapter.

Signal	Type	Comment
pc	output	A 32-bit signal that reflects the current value stored in the PC; the PC uses this value as an address to access instructions in the main memory.
ld	input	Controls the synchronous parallel loading of data to the PC.
data	input	The data that is parallel loaded into the PC when the ld signal is asserted and synchronized to an active clock edge.
rst	input	Synchronously resets the PC when asserted.
clk	input	System clock

Table 16.5: PC input/output signal description.

16.4.2 PC Functionality

The PC is a simple register but it has a few responsibilities that are key to the operation of the RISC-V MCU. This section describes the functionality of the PC as it relates to the diagram in Figure 16.6. Keep in mind as you read the following bullets that many of the implementation details that are arbitrary. There are many ways to do this; the current RISC-V OTTER MCU architecture chooses one of the simpler approaches.

- We use the output of the PC to access instructions in memory, which requires two types of operations relatively to the instructions in the program. All RISC-V MCU instructions are one of two types: *program flow control* or otherwise. Instructions that are not program flow control related always load the address of the next instruction into the PC. As you can see from Figure 16.6, the address of the next instruction is four greater than the address of the current instruction. The “+4” box in Figure 16.6 modifies the current address by adding four to it. This works because RISC-V instructions are 32-bits wide and the RISC-V MCU's main memory is byte addressable. The fact that main memory is byte addressable means that we have to advance the PC output

forward by four (or our bytes) in able to access the next instruction in program memory. Note that the “+4” modification of the PC is an input to the MUX, which means the hardware has the ability to load it into the PC under control of the `pcSource` signal.

- The PC is also responsible for implementing program flow control instructions. Recall that program flow control instructions are ones that cause program flow to transfer to an instruction other than the next instruction in program memory. The instructions are thus associated with conditional and unconditional branches. The `pcSource` signal is a control signal, and is an output of the control unit; the control unit is responsible for sending out the correct `pcSource` signal based on the instruction that the MCU is currently executing. If the MCU is currently executing a branch instruction, the hardware loads one of the three lower signals connected to the MUX in Figure 16.6 into the PC. Another module in the RISC-V MCU architecture determines the actual value of the data that loads. The three lower inputs to the MUX are `jalr`, `branch`, and `jal`. The `jalr` and `jal` inputs correspond to the `jalr` and `jal` instructions, respectively, which are both unconditional branch instructions. Be sure to recall that the `jalr` and `jal` instructions implement subroutines calls and returns. The `branch` input corresponds to all of the branch instruction; the `BRANCH_COND_GEN` module determines if the branch is taken or not, and then directs the `CU_DCDR` to output the correct `pcSource` signal. If the program does not take the branch, the hardware executes the next instruction in program memory, which is does by advancing the PC by four using the non-branch MUX input.
- The output of the PC drives the program memory address input of main memory. As you’ll see later, main memory is quite specialized in that it serves as both program and data memory. The size of main memory is currently limited on the development board, so we do not use all the output bits of the PC to access instruction. There are 16k 32-bit locations (or 64k 8-bit locations) in main memory, which is why we opt to only connect 14 of the PC’s outputs to the address input of main memory.

16.4.3 `jal` & `jalr` Instruction Details

The RISC-V MCU has two unconditional branch instructions: `jal` and `jalr`. Recall that although the RISC-V ISA includes a `call` and `ret` instructions, they are pseudoinstructions that the assembler translates to the `jal` and `jalr` instructions (and sometimes includes others). This section examines the implementation details of these instructions.

The `jal` and `jalr` instruction mnemonics stand for “jump and link” and “jump and link register”. These are very versatile instructions, though their versatility initially makes them rather challenging to understand. Table 16.6 shows the pertinent information regarding the two instructions; here’s the full skinny:

- The link part of the instruction is the same for both instructions; both instructions create a “link” by storing the address of the instruction following the current instruction (`jal` or `jalr`) in the stated register. This hardware can then use this value as a return address when returning from a subroutine. The RTL shows that the RISC-V MCU hardware stores the return address in the register listed as a destination operand. If the programmer provides no register, the assembler instructs the hardware to store the return address in `x1`, which we also refer to as “`ra`”. Note in the RTL that the value being stored is four greater than the current PC value, which is simply the address of the instruction following the `jal` or `jalr` instruction. Because the link part of the instructions is the same, programmers can use either instruction to call subroutines.
- The jump part of the instruction (the part that modifies the PC) is not the same for the `jal` and `jalr` instructions; this is where the “register” in jump and link register comes in. Both versions modify the PC, but they do so in different ways, as indicated by the RTL. The RTL for the instructions show that the new PC is a function of a label, used as a relative offset value, and a second value. The second value for the `jal` instruction is a PC while the second value for the `jalr` instruction is a register. The notion here is that the register to use with the `jalr` instruction is the link register used by a `jal` or `jalr` instruction, which effectively allows the `jalr` instruction to

act as a return from subroutine instruction. Thus, while we can use either instruction to call subroutines, we can only use the `jalr` instruction to return from subroutines. The `jalr` instruction does not use the label value in the address calculation when returning from a subroutine; the hardware expects the value in the register used as the return address register to be an absolute address value (after all, it is a 32-bit register).

Instr Type	Instruction Form	Instruction RTL	Example Usage
J-Type	<code>jal rd,lab</code>	$X[rd] \leftarrow PC + 4$ $PC \leftarrow PC + lab$	<code>jal x8,lab</code>
	<code>jal lab</code>	$x1 \leftarrow PC + 4$ $PC \leftarrow PC + lab$	<code>jal lab</code>
I-Type	<code>jalr rd,rs1,lab</code>	$X[rd] \leftarrow PC + 4$ $PC \leftarrow rs1 + lab$	<code>jalr x5,x6,lab</code>
	<code>jalr rs,lab</code>	$x1 \leftarrow rs1 + lab$ $PC \leftarrow rs1 + lab$	<code>jalr x7,lab</code>

Table 16.6: Two forms of the two unconditional branch instructions.

These instructions are definitely tough to understand at first, but all is not lost. We actually rarely if ever have a need to use these instructions because the RISC-V includes four pseudoinstructions that are much more intuitive. Table 16.7 lists these four pseudoinstructions with usage information. This table underscores the fact that we don't need to understand the exactly how the pseudoinstruction translate to base instructions because the assembler takes care of most of the details. Our mission becomes one of understanding the how the individual base instructions work on the hardware level so we can correctly implement them. Once we've correctly implemented the `jal` and `jalr` instructions, the assembler does the correct math and formats the correct fields in the machine code associated with the instructions such that when the hardware executes them, they simply work.

Instruction Form	Equivalent Base Instruction(s)	Example Usage	Comment
<code>j label</code>	<code>jal x0,label</code>	<code>j label</code>	Jump to instruction associated with label
<code>jr rs1</code>	<code>jalr x0,0(rs1)</code>	<code>jr x8</code>	Jump to instruction at address in rs1
<code>call rd,label</code>	<code>auipc rd,hi(label)</code> <code>jalr rd,lo(rd)</code>	<code>call x5,subrot</code>	Jump to instruction associated with label; Store current address in rd
<code>call label</code>	<code>auipc x1,hi(label)</code> <code>jalr x1,lo(x1)</code>	<code>call subrot</code>	Jump to instruction associated with label; Store current address in x1
<code>ret</code>	<code>jalr x0,0(x1)</code>	<code>ret</code>	Jump to instruction at address in x1

Table 16.7: The program flow control pseudoinstructions and their base instruction translations.

Here's the final summary of these two instructions. Keep in mind that RISC-V designers created these instructions to be versatile, but that design goal makes it hard for humans to easily understand how these instructions operate. Here is the final summary:

- Both `jal` and `jalr` are jump instructions that transfer program control to somewhere other than the next instruction the program. The hardware implements these jumps by loading absolute address values into the PC.
- The assembler is responsible for encoding the correct immediate values into the underlying machine code for both instructions while the hardware is responsible for calculating the absolute address for both instructions.
- The encoded immediate value for both instructions represents signed values, which allow program control to jump forward or backward in the program. The instruction uses both immediate in the absolute address calculations.
- These two instructions jump; the primary difference between these two instructions is how they calculate the absolute address, which is the address in instruction memory to jump to (the new value loaded into the PC). The `jal` instruction uses the immediate value as a signed offset that will modify the current PC value; the `jalr` instruction also has an offset, but the instruction uses that offset to modify an address in a register to form the absolute address. Thus, the new absolute address in the `jal` instruction is a function of the current PC value, while the absolute address in the `jalr` instruction is not.
- The fact that the `jal` instruction's absolute address calculation is based on a signed offset value added to the PC, the instruction is limited to how far in program memory it can jump. It can jump in either direction, with the offset value effectively limited to a 21-bit value, which means 20 bits in either direction. The `jalr` instruction does not have limits on the jump distance because the register value in the `jalr` instruction's address calculation can be an absolute address.

Table 16.8 show the instruction formats for the `jal` and `jalr` instructions. Both of these formats use `rd` to specify the destination register and `rs1` to specify the source register. The `jalr` instruction has a field for a source register because it uses a value in a register to calculate the address to jump to.

Table 16.9 shows the underlying machine code formats for both the `jal` and `jalr` instructions. The `jal` instruction uses a 20-bit immediate field to encode the signed offset value. There are two things to notice regarding the immediate field in the `jal` instruction. First, the ordering is somewhat wacky. The RISC-V decided upon this approach in an effort to save hardware by aligning some of the bits with similar bits in other instruction formats. Second, the LSB of the immediate value is not included in the machine code. Since these are jump in instruction memory, and instructions are 4-bytes wide, there is no need to encode the two LSBs. We only encode the LSB to allow RISC-V to support 16-bit wide instructions. This approach effectively doubles the jump range without having to encode an extra bit.

The immediate field for the `jalr` instruction is typically set to zero, which supports the use of `jalr` in returning from subroutines. Recall that both jump instruction “link” the return address (store the value of the next instruction following the current instruction being executed) in `ra`.

Instr Type	Instruction Format	
I-type		
J-type		

Table 16.8: I-type and J-type instruction formats.

Instr Instr type	Instruction Format
jalr I-type	
jal J-type	

Table 16.9: Machine code format for `jal` and `jalr` instructions.

16.4.4 Conditional Branch Instruction Details

One of the other values that the PC can load are the branch address, which are the addresses the program flow jumps to when the conditions associated with the branch instruct the hardware to “take” the branch (as opposed to not taking that branch and instead continuing onto execute the next instruction in memory). All conditional branch instructions share the same B-type format. Table 16.10 shows the B-type instruction format; we use shading to indicate opcode fields, and no shading to indicate field codes. Table 16.11 shows the machine code formats for each of the branch instructions. Here is some other fun stuff to note about the B-type instruction format:

- The B-type format includes two 5-bit source register fields; the underlying hardware uses the values in the registers designated by these fields as sources for the conditions that the branch instructions uses to determine whether to take the branch or not.
- The B-type format includes a 12-bit immediate field, which it stores in some wacky order and divided into two chunks (done this way to save hardware resources). The immediate value serves as a *signed offset* that is added to the current PC value and loaded into the PC to effectively implement the branch (when the conditions determine that the branch needs to be taken). The instruction does not encode the LSB of the branch address because the value is always zero based on the width of the RISC-V instructions. Not encoding the LSB allows the branch range to double without requiring needing to store the extra bit. It is the assembler’s responsibility to form and encode the correct immediate value; it is the hardware’s responsibility to reconstruct the absolute branch address from the current PC and the encoded immediate value.

Instr Type	Instruction Format
B-type	

Table 16.10: B-type instruction format.

Instr Instr type	Instruction Format
beq	
bge	
bgeu	
blt	
bltu	
bne	

Table 16.11: Machine code formats for the base branch instructions.

16.5 Main memory

The main memory in the RISC-V OTTER MCU serves three primary functions: 1) stores the program, 2) stores generic data, 3) acts as an interface for I/O operations. Additionally, some portion of data memory serves as the stack. This memory module is thus the most complex module in the RISC-V MCU architecture. In this section, we describe the memory's functionality in terms of its three primary functions.

Figure 16.7 shows the black box diagram for the memory module. The BBD in Figure 16.7 has all inputs to the module on the left side and all outputs from the device on the right side. We modified the ordering of the inputs and outputs to the module compared to the RISC-V MCU schematic to better describe the functionality of the device.

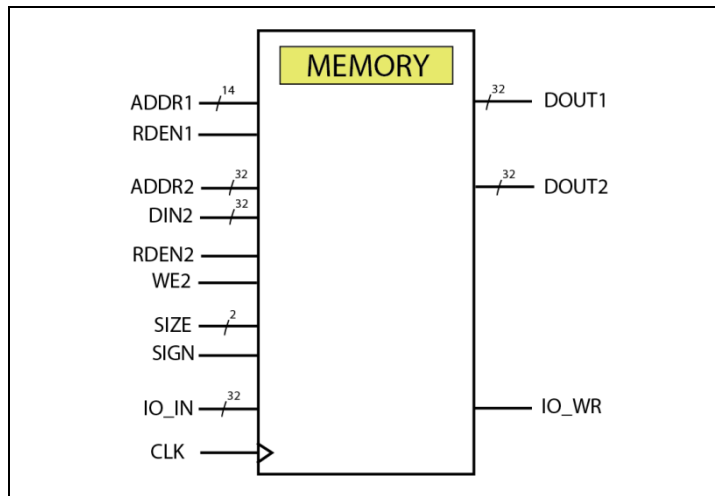


Figure 16.7: Black box diagram for the main memory module.

Table 16.12 lists and briefly describes the input and output signals associated with main memory. The main memory serves two primary functions: program memory and data memory. Although Table 16.12 appears daunting with the sheer number of entries, this one fact helps you grasp it better. We delineate the signals associated with the different memory functions (read and write enables, address, data input, data output) using

the number “1” and “2”, where signals associated with program memory have a “1” in the signal name and signals associated with data memory have a “2” in the signal name. Get used to it; we use it quite a bit.

Signal	Type/ width	Comment
ADDR1	in/14	The address lines providing access to program memory. This input is the output from the PC; the upper 16 bits and lower 2 bits are not connected.
RDEN1	in/1	Enables the instruction addressed by ADDR1 to output to DOUT1; the instruction data output is synchronized to the memory’s rising clock edge.
DOUT1	out/32	The instruction at the address specified by ADDR1. This output is often referred to as the “ ir ”, short for instruction register.
ADDR2	in/32	The address lines providing access to data memory and I/O. The memory module uses the value of these address lines to differentiate between I/O and data access.
RDEN2	in/32	Enables the instruction addressed by ADDR2 to output to DOUT2; the data stored in memory is output is synchronized to the memory’s rising clock edge.
DIN2	in/32	The data written into memory at the address specified by ADDR2 input. Write operations require an asserted WE2 and are synchronized to the rising clock edge.
DOUT2	out/32	The memory read operations, this is the data specified ADDR2. For input operations, this is a copy of the input data on the IO_IN input.
WE2	in/1	The write enable signal for data memory. This signal must be asserted for data to be written to memory on the rising clock edge.
SIZE	in/2	Use for memory reads to determine placement of bytes and half words in destination word; for memory writes it determines which byte or halfword in source register are written to memory.
SIGN	in/1	Used in memory reads to sign extend byte and halfword read: SIGN=1 are for zero extension of read value and SIGN=0 are for sign-extending read values.
IO_IN	in/32	The data input to the RISC-V MCU from the outside world. This data is passed directly to the DOUT2 output for input operations.
IO_WR	out/1	This signal asserts during the execute cycle of the load instructions used as data output instructions. This signal is an output from the RISC-V MCU.
CLK	in/1	The clock input; all memory reads and memory writes are synchronous.

Table 16.12: Description of main memory inputs and outputs.

The physical main memory in the RISC-V MCU has a capacity of 16k x 32, or 64k x 8. We list the capacity in two ways to underscore the fact that it is byte addressable in terms of data transfers. This essentially means that we can read and write individual data at any address 64k physical memory space. The main memory stores both the program and generic data, however. Where exactly it stores the program and data is generally arbitrary, but in projects such as the RISC-V MCU and its probable implementation on an FPGA-based development board, we use the memory segmentation provided in Figure 16.8. If you know how to work with the assembler and know how the assembler interfaces with the development tools, you can construct your memory map any way you want. Otherwise, you should comply with the memory map in Figure 16.8.

Due to resource limitations on the development board, the “physical portion” of main memory is limited to 16k locations of 32-bit data (or 64k of 8-bit data). The memory map in Figure 16.8 shows this by delineating the data storage portion of memory (code, data, and stack segments) in the address space spanning from 0x00000000 to 0x0000FFFF. The memory module treats addresses above the physical memory address in a different way, which primarily is I/O.

There is no magic associated with the address delineations in the physical address space of main memory; the segment boundaries are once again arbitrary. The main mission for programmers is to prevent data in one segment from overwriting important data from another segment. This is a mission that's easier to accomplish if the programmer understands the underlying limitations of physical memory.

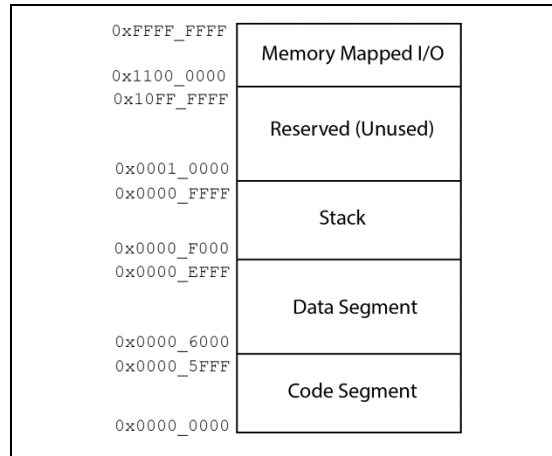


Figure 16.8: The RISC-V MCU memory map.

16.5.1 Physical Memory

Many MCUs, including the RISC-V, refer to the notion of address space. For example, we consider the RISC-V MCU as having a 32-bit address space. The notion of address space does not necessarily correspond to actual physical memory. For example, despite the RISC-V MCU having a 32-bit address space, 64k of that address space refers to physical memory, with each address location referring to a single byte. The hardware designer can “map” that 64k of physical memory anywhere in the 32-bit address space; thus, the memory map in Figure 16.8 is arbitrary.

We divide our description of main memory in this section into physical memory and “other” memory space (I’m trying not to say “virtual memory”). Once again, the physical memory holds instruction and data while we associate the non-physical memory with input/output operations.

16.5.1.1 Program Memory

We refer to the space in memory that stores the program as the program memory. The memory map in Figure 16.8 arbitrarily places the program memory starting at the lower addresses in memory.

All instructions in the RISC-V MCU are 32-bit wide, which can sometimes be confusing because the data in main memory is byte addressable. The main issue for program memory is that every instruction spans four-byte addresses in memory. The reason we don’t use a counter for the program counter is that counter typically increment (add 1). In the RISC-V MCU, it’s simply easier to include the hardware that adds four to go to the next instruction in program memory. Because of this, we can best implement the RISC-V OTTER MCU program counter as a register, but always referred to as a “counter”, and in particular, the “program counter”.

There are versions of the RISC-V that use 16-bit instruction words, but that’s simply not what we’re using in our RISC-V MCU implementation. The RISC-V ISA designers included instruction level support to process 16-bit instructions, which you can see by the fact that the LSB is not stored in the relative addresses associated with the `jal`, `jalr`, and branch instructions.

One of the issues involved in reading 4-byte chunks of data from a byte addressable memory is the issue of alignment. Our RISC-V instructions must grab the correct 4-byte chunk of memory, or the hardware can’t correctly decode and execute the instruction. For example, the program memory needs to output four bytes of the same instruction, not two bytes from one instruction and two bytes from another instruction. This is a common issue in many MCUs, as the width of the instruction word is not typically the same width as the data the instructions need to work with.

As it turns out, the main memory module we use in our RISC-V implementation works with the external hardware to ensure that all program memory accesses are properly aligned. The PC output, which is the address input to program memory portion of main memory, does not connect the two lowest LSBs to the ADDR1 input to the main memory. Related to that is the fact that physical memory does not extend past 0x0000FFFF, so the hardware does not route the two most significant bytes of the PC to the program memory address input (ADDR1). In the end, it then becomes the responsibility of the main memory module to create a 32-bit address from the 14 bits sent to the program memory from main memory.

In theory, program memory is not officially writable. The notion here is that some outside entity using some unspecified mechanism put the machine code into program memory. That being the case, there is no need to write new data to program memory², and thus the program memory portion of data memory does not have a data input or a write enable. You'll see something different with "data" memory. On that note, all memory reads, which includes both program and data memory reads, are synchronous, which means the data at the given address only appears after the memory modules sees a rising clock edge with the appropriate read enable signal asserted. The RDEN1 signal is a positive logic signal that serves as the read enable for the instruction memory portion of main memory.

The final comment for this section is the official starting point of the program. In order for the hardware to start any program, the hardware must know the location of the first instruction in the program. This fact requires some type of agreement between the hardware and the assembler. The instructions are a set of machine code that the programmer can place anywhere in program memory; the programmer must be able to somehow communicate with the hardware what the address of the first instruction is so that the hardware can load that value into PC before it does anything else. This escapes me right now, but there is a special name for this, such as entry point. There currently is no such mechanism in the RISC-V OTTER MCU hardware. Because the current RISC-V hardware will most likely be implemented on an FPGA, the hardware clears the PC when powered on, which means the PC output is 0x00000000. This means that we then must ensure that we place our program at address 0x00000000 in main memory, which is clearly in the code segment according to Figure 16.8.

16.5.1.2 Data Memory

When we refer to data memory, we're "probably" referring to the portion of main memory that is not the program memory but still part of physical memory. But then again, we can consider the entire physical main memory to be data memory because the underlying hardware does not know what you're storing in it: it just stores the bits the instruction tells it to store. Unless we specify otherwise, we'll use the term data memory to refer to part of physical memory not intending and/or not currently storing instruction data.

Data memory of course stores data, but we can further classify it by the nature of the data it stores. In the RISC-V architecture, we use data memory to store data in the "data segment", or in the "stack segment". Both areas store data, but the access is conceptually different. Conceptually speaking, we can access the data in the data segment using absolute addressing, which we access data in the stack segment according to the definition of an abstract data type called, wait for it, the *stack*.

Though we've slapped a label on different chunks of memory, we use the same set of instructions to access all of physical main memory. In theory, we're not supposed to write to program memory, but we can actually do so using memory write (store-type) instructions. Similarly, we use the same memory access instructions to work with both the data and stack segments: it's just memory.

The main memory uses a "2" postfix on the memory's data input, output, control and address to signify data memory. Both data memory reads and writes are synchronous, where **WE2** controls the data writes and **RDEN2** controls the data reads. The **ADDR2** input is an absolute address used to index into that data segment. **ADDR2** is a 32-bit value that effectively addresses 16-bits of physical memory, the meaning the hardware treats addresses above the maximum 16-bit value (0xFFFF) as I/O (see section 16.5.2). The **DIN2** input is 32-bit data used to write to the RAM, the **DOU2** output is the 32-bit data read from RAM.

The load and store instructions allow for the loading (reading) and storing (writing) of data to main memory. There are three flavors of load and store instructions, which differ by the size of data being loaded or stored. The RISC-V instruction set provides the ability to store words (four bytes), halfwords (two bytes), or bytes (one byte,

² There of course is a notion of self-modifying code, but we don't want to go there.

duh!). If all the memory had to do were load and store words, things would be relatively simple; things become slightly more complicated when we need to access something other than words.

Store instructions can write one of three different sizes of data to physical memory: words, halfwords, or bytes. The three store instructions, **sw**, **sh**, and **sb** support the writing of these three sizes of data. The combination of the assembler and the RISC-V MCU hardware provides an absolute address to write the data to in memory; the hardware is ultimately responsible for providing this address from the summation of the base address provided by a register and a sign-extended offset value encoded in the immediate field in the instruction. The absolute address represents the lowest possible memory location address that the hardware can write the data to. This means that the **sw** instruction write a four-byte value starting at that address and includes the next three address values; the **sh** instruction writes a two-byte value starting at that address and includes the next address; the **sb** instruction writes a value at that address.

The main memory handles the address portion of the store instructions in a special way. The memory module only includes the 14 MSBs of the lower two bytes from the 32-bit **DIN2** address input for the address calculation. The **SIZE** signal provides the other two bits to the input to the memory. The size input from memory is the lower two bits of the funct3 field code (ir[13:12]) in the store instructions. The three store instructions are S-type instructions. Table 16.13 shows the instruction formats for the three store-type instructions. Note that bits [13,12] reflect the size of data to store with bytes="00", halfword="01", and words="10". The memory module formulates the absolute memory address by replacing the two lower bits of the two lower bytes of the **ADDR2** input with these values.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
sb	sb $rs2, imm(rs1)$	$M[rs1 + sext(imm)] \leftarrow rs2[7:0]$	sb $x11, 0(x31)$	store byte in memory
S-Type				
sh	sh $rs2, imm(rs1)$	$M[rs1 + sext(imm)] \leftarrow rs2[15:0]$	sh $x11, 0(x31)$	store halfword in memory
S-Type				
sw	sw $rs2, imm(rs1)$	$M[rs1 + sext(imm)] \leftarrow rs2$	sw $x11, 0(x31)$	store word in memory
S-Type				

Table 16.13: The store-type instructions.

Reading from memory is inherently different from writing to memory in terms of how the memory modules respond. Writing to memory takes one of the three data sizes and places it starting at the specified location in memory. On the other hand, reading from memory always places the read value into a register, while the registers are all 32-bits wide. This works nicely for reading words, but it brings up the question of what to do with the non-used register bit locations when reading halfwords and bytes. As it turns out, the memory module fills the unused bits with either zeros (zero extension) or the sign-bit (right-most bit) of the data being read from memory (sign extension).

The RISC-V MCU's load-type instructions perform reads from memory and place the data into specified registers. There are five load type instructions, two each for loading bytes and halfwords, and one for loading words. Loading bytes and halfwords into registers can be specified as either signed or unsigned, which designates how the unused register bits are assigned (signed uses sign extension while unsigned used zero extension). Note that because there is no notion of unfilled bits in the register for reading words from memory, there is no need for two types of load words instructions as there are for loading bytes and halfwords.

Table 16.14 shows the instruction formats and other useful information associated with the five load-type instructions. Note that the instruction formats dedicate a single bit to indicate whether read instructions (load-type) are of the signed or unsigned type. More specifically, the MSB of the **funct3** field (**ir[14]**) indicates which of the load-type instructions (not including **lw**) require zero extension. This bit is input to the memory modules as the **SIGN** input. Table 16.14 uses “sext” and “zext” for sign and zero extending, respectively.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
lb	lb rd, imm(rs1)	$rd \leftarrow \text{sext}(M[\text{rs1}+\text{sext}(\text{imm})][7:0])$	lb x11, 0(x20)	load byte from memory signed
I-Type				
lbu	lbu rd, imm(rs1)	$rd \leftarrow \text{zext}(M[\text{rs1}+\text{sext}(\text{imm})][7:0])$	lbu x12, 0(x30)	load byte from memory unsigned
I-Type				
lh	lh rd, imm(rs1)	$rd \leftarrow \text{sext}(M[\text{rs1}+\text{sext}(\text{imm})][15:0])$	lh x15, 0(x23)	load halfword from memory signed
I-Type				
lhu	lhu rd, imm(rs1)	$rd \leftarrow \text{zext}(M[\text{rs1}+\text{sext}(\text{imm})][15:0])$	lhu x21, 0(x21)	load halfword from memory unsigned
I-Type				
lw	lw rd, imm(rs1)	$rd \leftarrow M[\text{rs1}+\text{sext}(\text{imm})][31:0]$	lw x23, 0(x22)	load word from memory
I-Type				

Table 16.14: The load-type instructions.

16.5.2 Input/Output Memory Space

The memory space in the RISC-V MCU involves input and output because the RISC-V uses memory-mapped I/O (MMIO). The memory map in Figure 16.8 provides a specific segment dedicated to I/O. This memory space is not associated with physical memory, however. In typical digital systems, part of the design includes configuring the system such a memory read or write (load or store) that happens at a particular address will be an I/O operation rather than a memory access operation. Recall that part of the memory mapped I/O mechanism is that the assembler does not know the different between load/store instructions used for physical memory access and the same instructions used for I/O. The same is true in the RISC-V MCU.

The responsibility of interpreting load and store instructions as either memory reads or writes, or I/O instructions lies in the RISC-V MCU hardware. More specifically, we model the main memory model in such a way as to take full responsibility for this determination: no other RISC-V module is involved. The memory module makes this determination based solely on the value on the **ADDR2** input: the module interprets load and store instructions specifying an address above 0x0000FFFF as I/O; the module interprets all other load and store as physical memory accesses.

Figure 16.9 show a diagram that models how the memory module handles load-type instruction. In this diagram, we use two MUXes to indicate how the **ADDR2** input controls the **DOUT2** and **IO_WR** outputs. Both operations are associated with one of the RISC-V’s five load instructions. Here are the details listed by output signal name:

DOUT2: The memory module transfers data from the **IO_IN** input to the **DOUT2** output when the **ADDR2** input is greater than 0xFFFF. This is an input operation in that **IO_IN** is 32-bit input signal that the RISC-V MCU uses to input data from devices external to the MCU. When the **ADDR2** input is less than 0x10000, the memory module places 32-bits of data from a physical memory address on the **DOUT2** output.

IO_WR: This signal is an output signal on the RISC-V MCU. The MCU uses this signal to indicate to external devices that the MCU is executing an output operation. The circuitry that interfaces external hardware to the RISC-V MCU typically uses this signal to control the latching of data output from the MCU into external registers. When the **ADDR2** input is greater than 0xFFFF, the memory module transfers the **WE2** signal to the **IO_WR**; the hardware only asserts **WE2** during the writeback cycle of any load-type instruction. The hardware clears the **IO_WR** signal during all other instructions including load-type instructions used for physical memory reads.

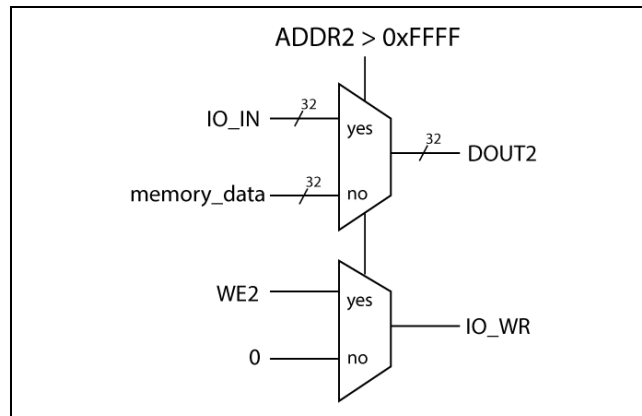


Figure 16.9: Model of memory data associated signals.

16.5.3 Memory Timing Issues

The specific nature of the underlying RISC-V hardware makes the instruction execution timing somewhat unique. The program counter (PC) provides the address of the “current” instruction the RISC-V MCU is currently executing. We put the word “current” in quotations because this is the official definition of the PC, but as you’ll see, it’s not always 100% accurate depending on the exact time you examine it.

Figure 16.10 shows a partial RISC-V schematic highlighting the interface between the PC and the memory module. Figure 16.10 shows the signals found in the timing diagram of Figure 16.11; we don’t bother including less important signals.

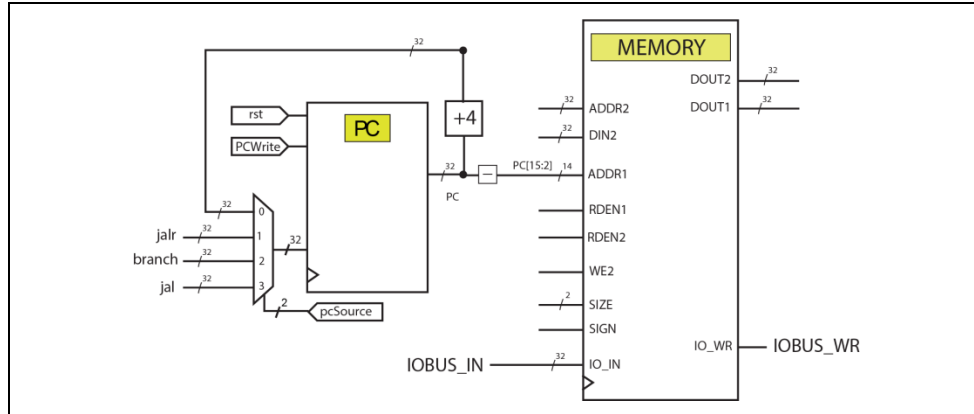


Figure 16.10: Schematic diagram supporting Figure 16.11.

Figure 16.11 shows the important timing features associated with reading instruction from the memory. In this context, reading instruction from memory is the responsibility of the “fetch” cycle part of instruction execution. Here are some of the more important features from Figure 16.11:

- The timing diagram shows the execution of two full instructions and another fetch cycle. The instructions being executed don’t matter here except for the fact they are not load-type instructions or program flow-type instructions³.
- The PC[15:0] output represents the lower two bytes of the PC output. The upper 14 bits of this output becomes the address input to the memory. The address of the first instruction in the timing diagram is arbitrarily at 0x20, which only lists the lower byte of the PC[15:0] signal.
- The **PCWrite** signal is an output from the CU_FSM that controls the latching of data to the PC. This signal asserts upon entering the fetch cycle. Loading data to the PC is synchronous, so the **PCWrite** signal does not do anything until the next clock edge, which also causes a transition to the fetch state of the following instruction.
- The (1) note shows that the change in data is caused by the clock edge (and the **PCWrite** signal). All the RISC-V MCU instructions are 32-bits wide, which causes the PC to advance by four each instruction as the PC[15:0] lines show.
- The **DOUT1** signal is the output of program memory, which should show the underlying bits associated with each instruction. The diagram does not show machine code, but does show the address where the machine code lives.
- Reading from memory is synchronous, which is why the instructions for a given address do not appear immediately after the address changes at (1). The instruction bits for the current instruction appear only after the next active clock edge, which is at the end of the fetch cycle and marked by the (2) note.
- The **PCWrite** does not advance every clock edge, which is why we see the on/off pattern. It makes sense that the PC should advance only once per instruction cycle, which is what the diagram shows.
- Most important to note is the fact that the instruction bit output appears to be delayed by one clock cycle from the PC output. We refer to this as “important” because it’s somewhat non-intuitive. The notion that the read operations on the memory module are synchronous causes this behavior.

³ To be precise, they could be branch instructions, but the branch is not taken.

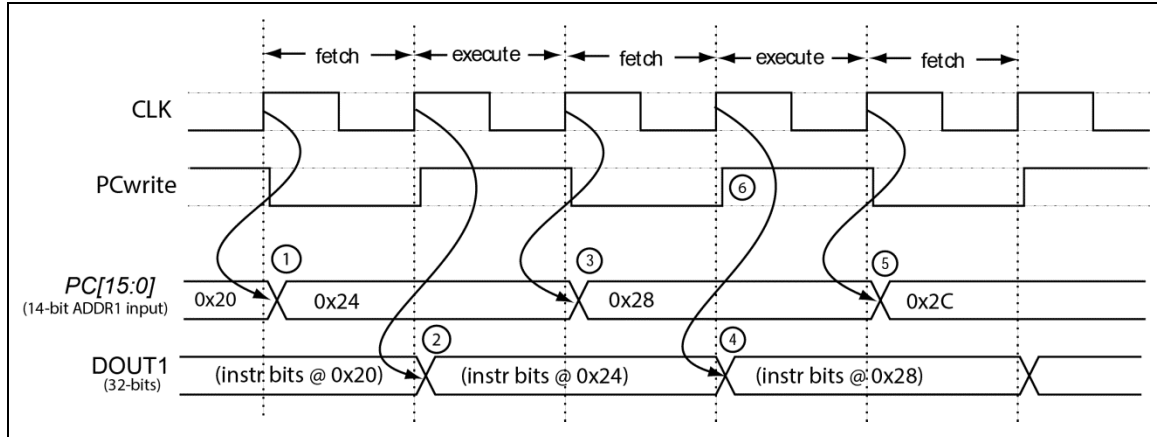


Figure 16.11: Example timing diagram for PC and instruction memory output.

16.5.3.1 Branch Instruction Timing

Branch instructions are one of the flow control-type instructions, which by definition change the value of the program counter to something other than to point at the next instruction in memory. Recall that the definition of the branch is that it is an instruction that jumps conditionally based on a comparison between two registers made as part of the instruction. There are six conditional branch instructions as in the RISC-V base instructions, and ten other pseudo-branch instructions that the assembler translates to base branch instructions.

When the MCU executes a branch instruction, the instruction actually causes a true branch when the condition associated with the instruction test as true. If the conditions are not true, program controls continues onto the next instruction in memory, exactly like a non-program flow control-related instruction. In this case, the only interesting thing to examine here is when the instruction causes a branch. Figure 16.12 show an example timing diagram associated with a branch-type instruction where the program conditions cause the program to *take the branch*. Here are some other fun facts to know about this example.

- The **PCWrite** signal always asserts after entering the execute state. The (1) note indicates that the PC loads a new value at the clock edge between the fetch and execute states. The note at (2) shows the **PCWrite** signal asserted, which cases the PC to load the new value (PC+4) into the PC.
- The instructions executed in the example are either non-program flow control (and non-load-type) or the **beq** instruction listed on the top of the diagram.
- The bits associated with **beq** instruction become available after entering the execute cycle, the label (3) indicates. Once the instruction bits become available, the register addresses to the register file become valid, which outputs valid data from the associated registers. These register's outputs are inputs to the combinatorial **BRANCH_COND_GEN** module, which then outputs relational data about the contents of the two registers to the **CU_DCDR**. The **CU_DCDR** uses the relational data in conjunction with the opcode data (that allowed the **CU_DCDR** to discern which instruction was being executed) to send out the appropriate value to the **pcSource** signal. When the MCU takes the branch, the **pcSource** signal is "10", as the execute state associated with the branch instruction shows.
- Because the **pcSource** signal is "10", the PC MUX directs the value associated with the branch label to the PC. The branch label in this case is done; the note in the diagram indicates the numerical value of done is 0x80, which is then loaded into the PC at the end of the execute cycle as indicated with (4) label.
- The instruction bits associated with the instruction at address 0x80 then appear on the memory output (**DOUT1**) at the beginning of the next execute cycle as noted by (5).

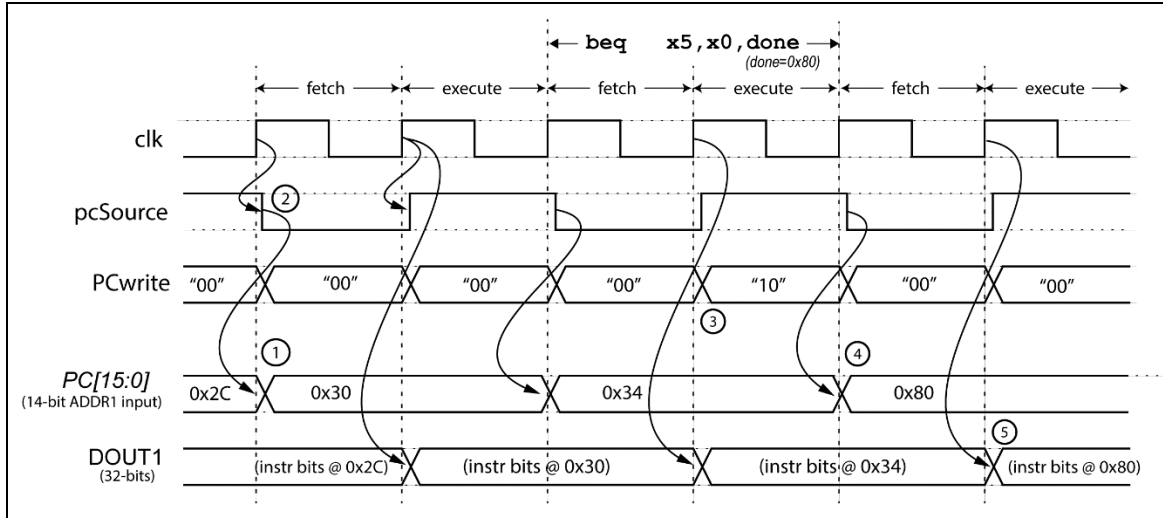


Figure 16.12: Example timing diagram showing a branch instruction with the branch taken.

16.5.3.2 Memory Access: Load-Type Instruction

When we speak of load-type instruction, we are referring to both memory read instructions and input instructions. Because the RISC-V uses memory mapped I/O, the load-type instructions thus serve as both memory reads and external data inputs. Recall that the assembler does not know the difference between memory read and I/O; the underlying RISC-V hardware, namely the main memory module, takes care of the difference.

Load-type instructions in the RISC-V ISA all operate in the same manner: they take data from somewhere (from memory or from some external device) and place that data into a register. The only difference between these instructions is where get the data from. In either case, it is the memory module's responsibility to place the correct data on the memory's **DOUT2** output. When the address associated with the load-type instruction is greater than 0x0000FFFF, data the memory modules transfer the data on the **IOBUS_IN** lines to the **DOUT2** output. When the address is 0x0000FFFF or less, the memory places the data at that memory address on the **DOUT2** output.

Figure 16.13 and Figure 16.14 show timing diagrams for load-type instruction for memory access and data input, respectively. The first thing to notice about these two figures is that they are essentially the same; the only difference is on the **DOUT2** output signal, which the figures do not show. The following is a detailed description of Figure 16.13, which shows a load instruction performing memory access.

- The only thing we know about the instruction before the load instruction is that it is not a load instruction, as it only has a fetch and execute cycle.
- We know this instruction is performing a memory load based on the effective address in the instruction's second operand. The effective address is the sum of the offset and the value in `x8`, which the diagram lists as 0x0000CD00. Because this address is less than 0x00010000, the instruction performs a memory read.
- Load-type instructions require three t-cycles (states), which the RISC-V vernacular list as fetch, execute, and writeback. The instruction in Figure 16.13 is an `lw`; other load-type instructions also require three clock cycles.
- The **PCWrite** signal asserted in the final state of any instruction execution. For two cycle instructions, that means **PCWrite** asserts during the execute state; for load-type instructions, **PCWrite** asserts during the writeback state. Following every **PCWrite** assertion, is the assertion of **RDEN1** during the execute state of all instructions, which allows the instruction bits to appear on the output of program memory (**DOUT1**). The (5) note highlights this.

- The **RDEN2** signal is a read enable signal for the data portion of the memory. This signal asserts at note (4) to allow the address in the memory's **ADDR2** input to have valid data. If for this memory access instruction, the memory outputs the data at the address on the **ADDR2** input to the **DOUT2** output. If this instruction were performing input, it would pass the **IOBUS_IN** input to the **DOUT2** output.
- Note (7) shows that the assertion of the **regWrite** signal allows the loading of the final data value (**DOUT2**), whether it be from memory (read) or an external device (input) into a register file. Note (3) shows that the data from the **DOUT2** memory output is selected as the data that writes to the register; the value "10" selects the **DOUT2** data output to be the written to the register file.
- Note (1) and (2) show that no data needs writing to memory, so **WE2** remains unasserted, and that nothing is being output, so **IO_WR** remains unasserted as well.
- All but the load-type instructions follow a given two-cycle format. The three-cycle load-type instructions effectively stretch that cycle to three states with the inclusion of the writeback state. You can see this as a "one cycle delay" in the PC and **DOUT1** lines associated with the load instruction.

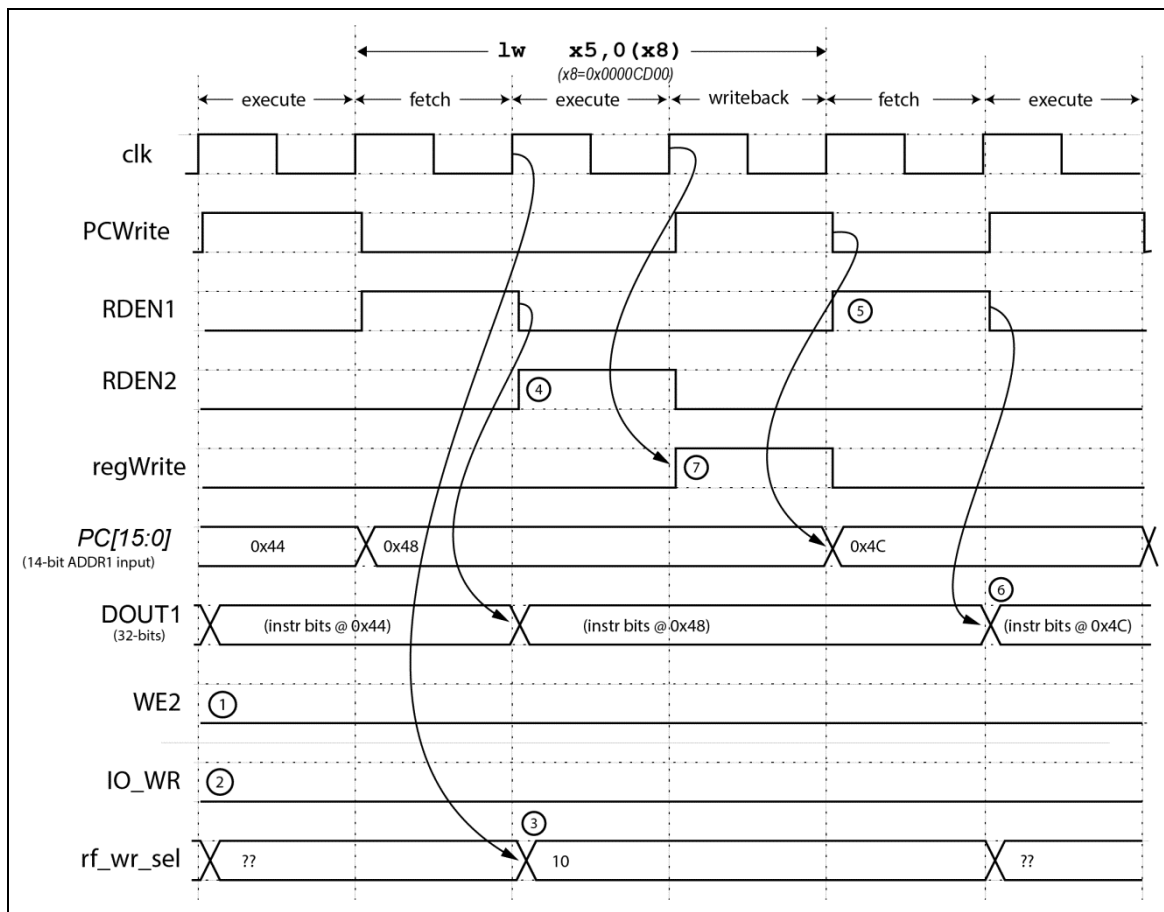


Figure 16.13: Example timing diagram showing a load instruction performing memory access.

16.5.3.3 Inputting Data: Load-Type Instruction

The RISC-V MCU also uses load-type instructions to input data. Figure 16.14 shows the timing associated with a load-type instruction that performs an input operation. This load-type instruction performs an input because the effective address is greater than `0x0000FFFF`, as the note under the instruction in Figure 16.14 indicates. This difference causes the only other difference in the instructions: when the memory modules see that the address is

greater than 0x0000FFFF, it transfers the **IOBUS_IN** value to the output. When its effective address was less than 0x00010000, the memory module transferred the value in memory to the **DOUT2** output (see Figure 16.13). The main point here is the similarity of Figure 16.13 and Figure 16.14; this being the case, we'll not bore you with another detailed description.

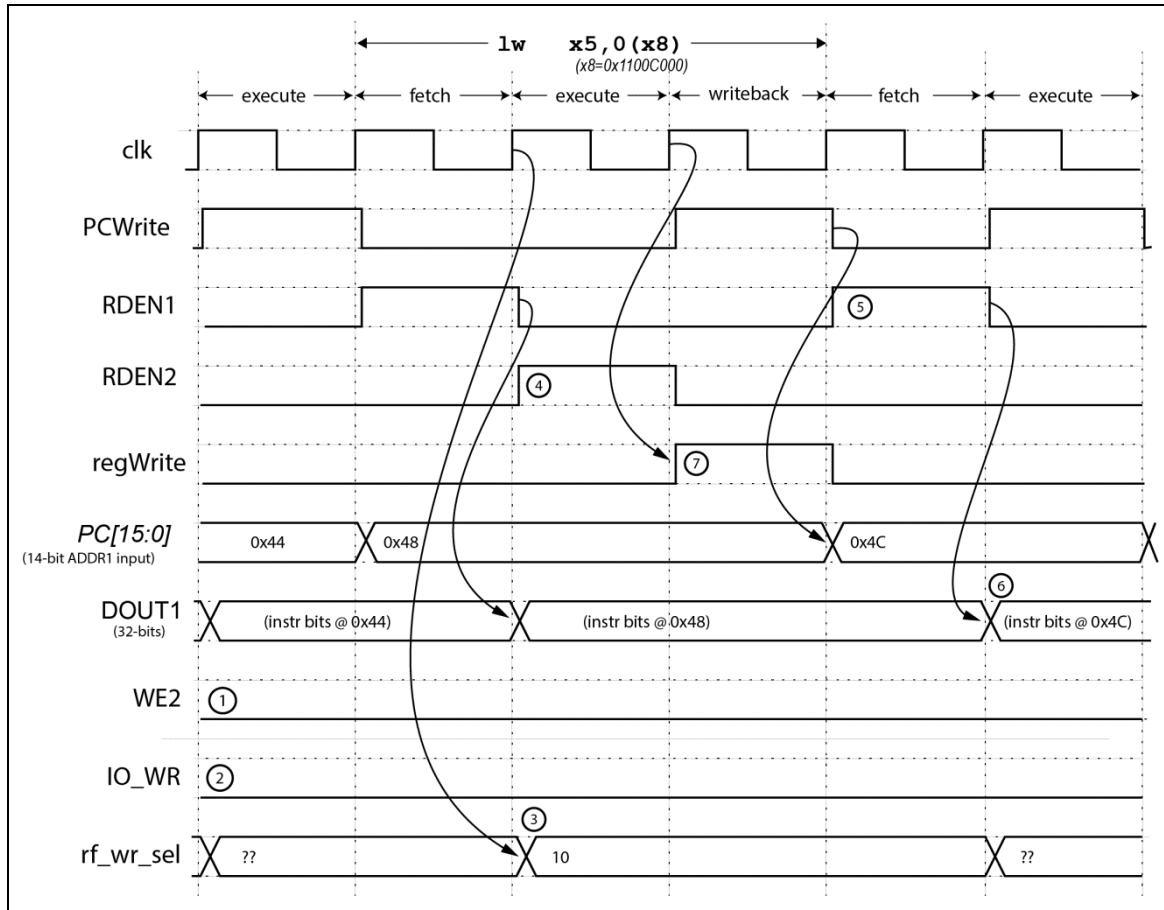


Figure 16.14: Example timing diagram showing a load instruction performing input.

16.5.3.4 Memory Access: Store-Type Instructions

When we speak of store-type instructions, we are referring to both memory write instructions (memory access) and input instructions (I/O). The RISC-V uses memory mapped I/O (MMIO), which means that the store-type instructions thus serve as both memory writes and internal data outputs to devices external to the MCU. One interesting artifact of MMIO is that fact that the assembler does not know the difference between memory read and outputting data; the differences are at the RISC-V MCU hardware level.

Store-type instructions in the RISC-V ISA operate in the roughly the same manner: they take data stored in a register (internal data) and make that data available to other entities. When programmers use store instructions for writing memory, the instruction instructs the underlying hardware to copy data from the register to a location in memory. When programmers use the store instruction as I/O, the data in the register is “made available” to devices external to the RISC-V MCU. We use the notion of being “made available” to mean that the MCU has no notion of what devices external to the MCU do with register data that the MCU made available. The only difference between these memory access and output instructions is where the register data goes.

Recall that the memory module makes the determination of whether the store instruction performs a memory access or a data output. When the address associated with the store-type instruction is greater than 0x0000FFFF, the memory module asserts the `IO_WR` signal on the memory module. When the address is 0x0000FFFF or less, the memory module does not assert `IO_WR`. In both cases, the control unit asserts the memory module’s

write enable **WE2**. It is the responsibility of the RISC-V MCU control units to ensure the data goes to the correct sources.

Figure 16.15 shows an example timing diagram for one store-type instruction. The following is a description of the pertinent details, worthy of attention from mere humans:

- The timing diagram shows one known instruction; all we now about the other instructions are that they are not load-type instructions because they don't include writeback states.
- The **sw** instruction performs a memory access based because the effective address is less than 0x00010000. The effective address is the value in x9 because the instruction includes a zero offset.
- Note (1) indicates that no instructions in the example include reading data from memory.
- Note (2) indicates the instruction is a memory access and thus does not assert the **IO_WR** signal.
- Note (3) indicates that no instructions are writing data to the register file. Recall that the instruction is "doing something" with data in the register file.
- Notes (4) & (5) show the write pulse on the **WE2** signal. The memory module uses this signal as an actual write enable because the instruction is performing a memory write. This signal asserts as part of the execute cycle for all store instructions and remains unasserted otherwise.
- Note (6) shows that the **PCWrite** signal cause the PC to advance by four, indicating that none of the instructions in this example are program flow control instructions. Note (7) reminds us of the fact that the program memory read enable (**RDEN1**) allows the instruction at the current PC address to output from program memory (**DOUT1**).
- Worth noting here is that the instruction is writing data to the memory, which is a synchronous operation. Unlike load instructions, we can perform this operation in two clock cycles because we don't have to read data from memory (memory reads are synchronous operations).

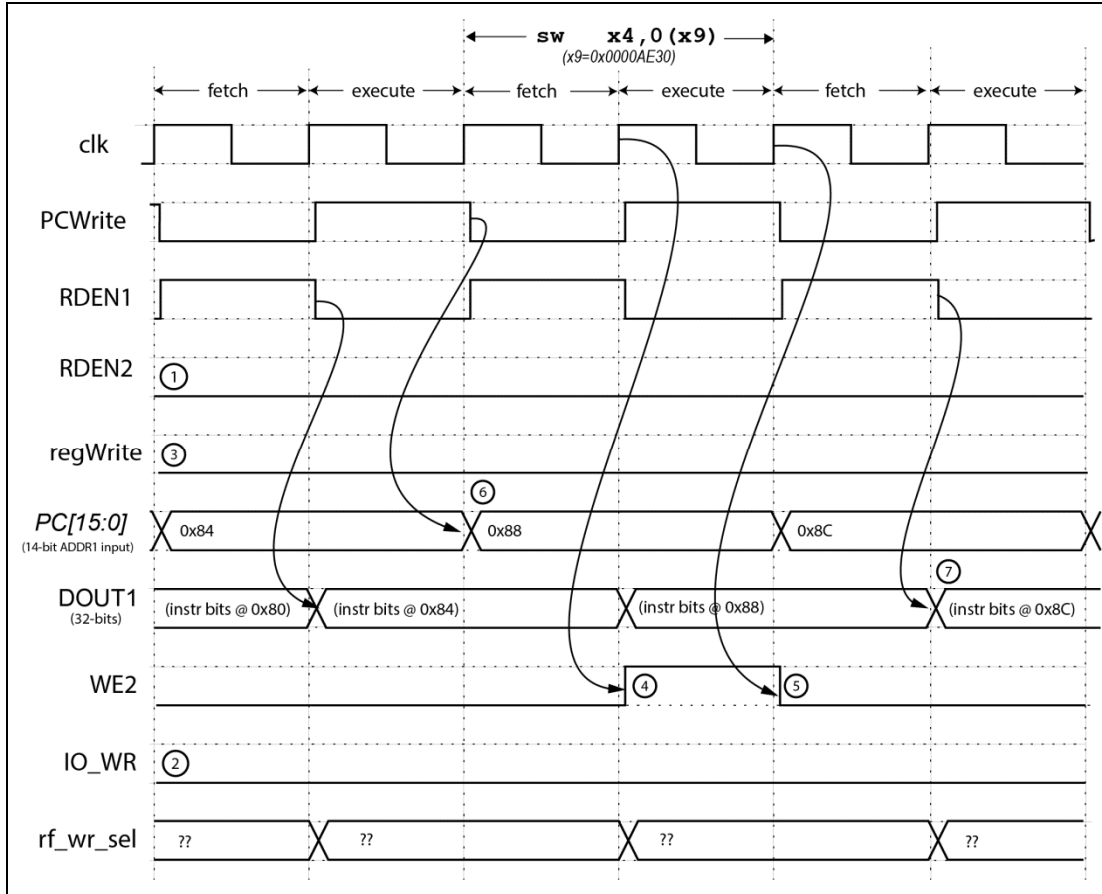


Figure 16.15: Example timing diagram showing a store instruction used for memory access.

16.5.3.5 Outputting Data: Store-Type Instruction

Because it uses a memory-mapped I/O architecture, the RISC-V MCU uses store-type instructions to output data. As with load-type instructions, the memory module is the only RISC-V hardware that knows the difference between a store instruction intended for memory access and a store instruction intended for output. Recall that the difference lies in the effective address value associated with the instruction. This implies that not even the control units in the RISC-V architecture know the difference either, which is a fact that we'll run across in the timing diagram example that follows.

Figure 16.16 shows the timing associated with a store-type instruction that performs an output operation. The store instruction performs an output because the effective memory address in the instruction is greater than `0x0000FFFF`, as noted under the instruction in Figure 16.16. When the memory modules see this difference, it generates a slightly different output than the store instruction intended for memory access. Note that Figure 16.16 doesn't show the register file's output data, which is the data the instruction provides to the outside world.

The timing diagram in Figure 16.16 is similar to the timing diagram in Figure 16.15, so we won't be describing it in the same painful level of detail as we did with Figure 16.15. The most important thing to note about these two timing diagrams is their only difference: **IO_WR** asserts for output operations and does not assert for memory write operations. Once again, this difference is a responsibility of the memory module, as no other hardware modules know the difference between store-type instructions used as I/O or memory access.

- Notes (1), & (3) show unasserted signals, indicating that the instructions do not read memory or write to the register file. Additionally, **rf_wr_sel** signal is always an unknown, which indicates that none of the instructions are writing the register file (**regWrite**=0)

- Notes (2), (4), and (5) show the difference special operation associated with output. The **WE2** signal asserts for all store instructions, but the **IO_WR** only asserts for when the store instruction performs a data output operation. Once again, the asserting of the **WE2** signal is a function of the **CU_FSM**, but the assertion of the **IO_WR** signal is a function of only the memory module.

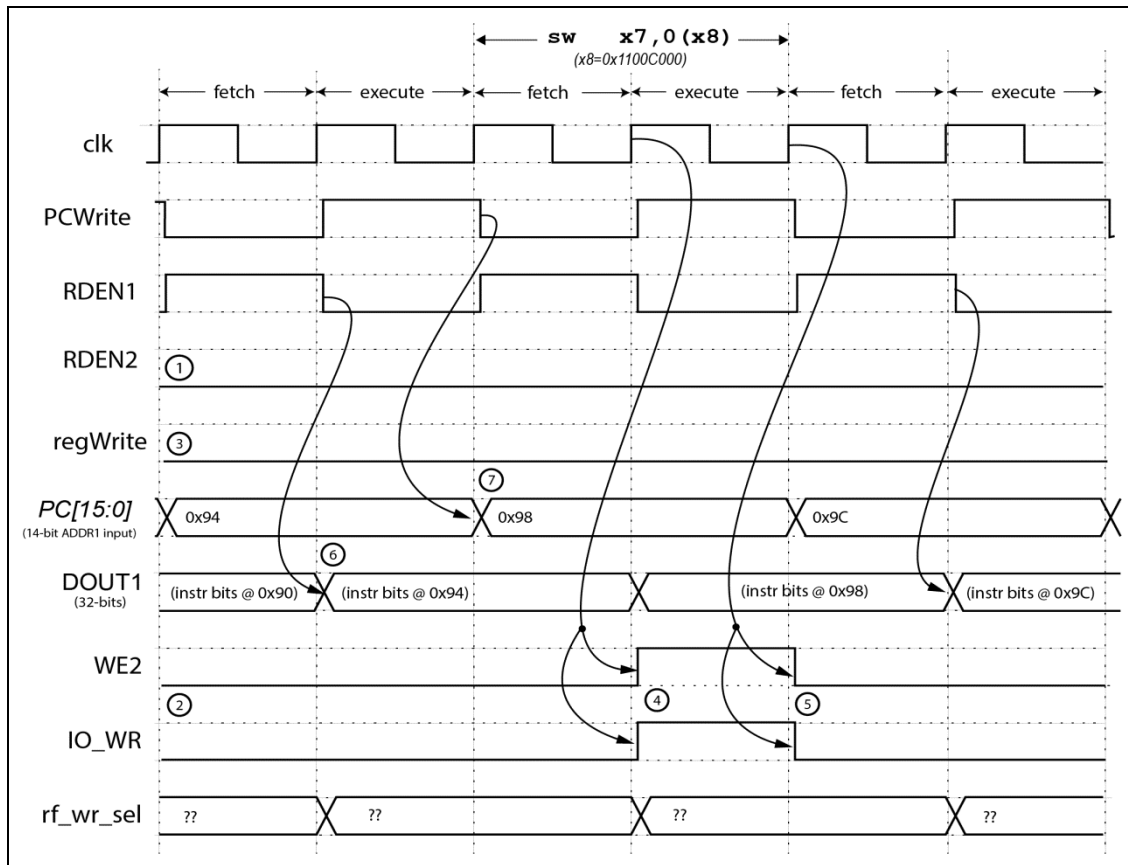


Figure 16.16: Example timing diagram showing a store instruction used outputting data.

16.6 The Immediate Value Generator (IMMED_GEN)

The IMMED_GEN module's function is to convert immediate values stored in the instruction bits (machine code) into 32-bit values. Five of the six RISC-V MCU instruction formats contain immediate fields of varying lengths and a bizarre mixture of formats. The immediate fields reside in the all but the right-most LSBs in the five instruction types that include immediate values. Figure 16.17 shows the block diagram for the IMMED_GEN module. The input to the IMMED_GEN module comprises of the left-most 25 bits of the instruction register; the outputs of the module include the 32-bit versions of the five instruction-type specific immediate values. R-type instructions don't contain immediate fields so are not a part of the IMMED_GEN module. The IMMED_GEN module is a combinatorial circuit.

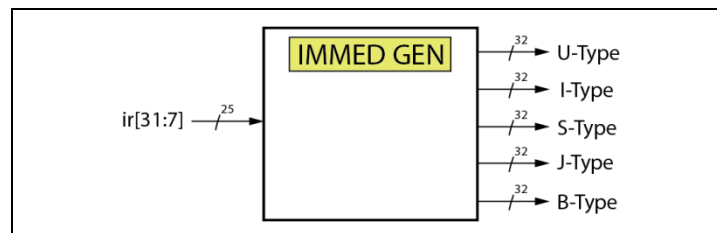


Figure 16.17: Block diagram of IMMED_GEN module.

The RISC-V OTTER MCU includes the IMMED_GEN module primarily to simplify the understanding and implementation of the architecture. People who are implementing the RISC-V MCU in hardware can actually do one of two things. First, they can omit this module, which would require that other modules using the 32-bit immediate values do the reformatting themselves. Second, this module does not need to be an actual module; a better approach would be to provide the functionality this module provides as part of the higher-level hardware module.

Table 16.15 lists the five instruction formats that contain immediate fields. Table 16.16 shows how the RISC-V hardware converts the five instruction immediate field in the instructions into 32-bit values. This IMMED_GEN module is thus responsible for decoding the immediate values from the assembler's encoding and then converting the values to 32-bit numbers. Note in Table 16.16 that the conversions associated with I-type, S-type, and B-type include sign extensions. Also good to note that the J-type and B-type outputs represent relative address values. Be sure to note that immediate values are variables, while opcodes (the shaded values in Table 16.15) are constant relative to individual instructions. Lastly, to note in Table 16.15 is that the strange numbering and omission of some values is one way the RISC-V supports efficient hardware implementations.

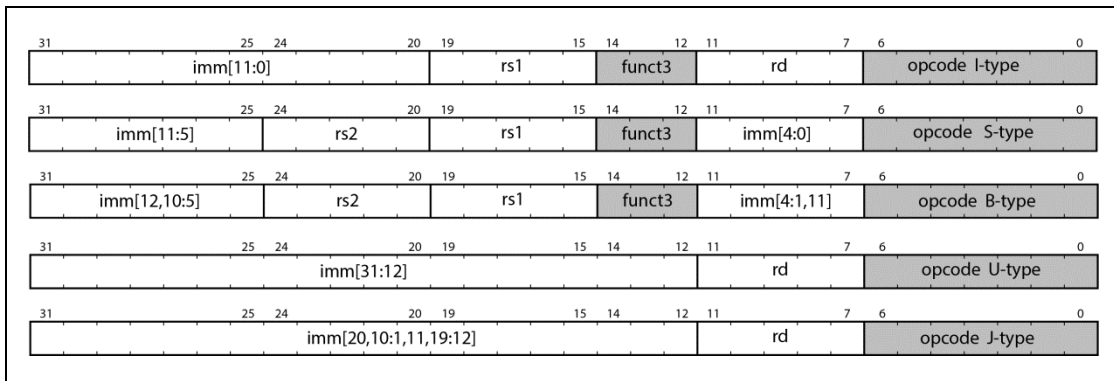


Table 16.15: The 32-bit immediate value transformation map.

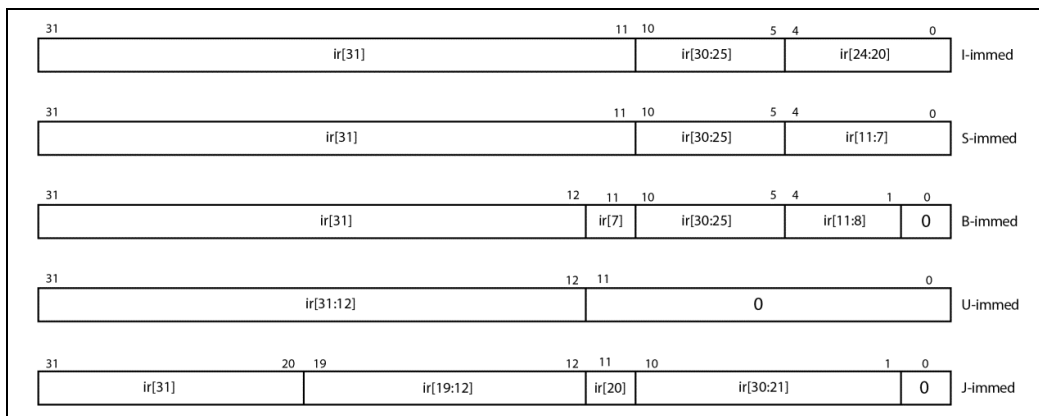


Table 16.16: The 32-bit immediate value transformation map.

16.7 The Branch Address Generator (BRANCH_ADDR_GEN)

The BRANCH_GEN module is responsible for generating absolute instruction memory address from various input data. As the name implies, this module generates 32-bit branch address values to be loaded into the PC, which supports the RISC-V MCU's program flow control instructions including branches and jumps. Another way to view this module is that it converts relative addresses from the immediate values associated with instructions into absolute addresses. This module's inputs are the I-type, J-type, and B-type 32-bit immediate value, and also the 32-bit PC and rs1 values. Note that rs1 is one of the operands output from the register file. The BRANCH_GEN module is a combinatorial circuit.

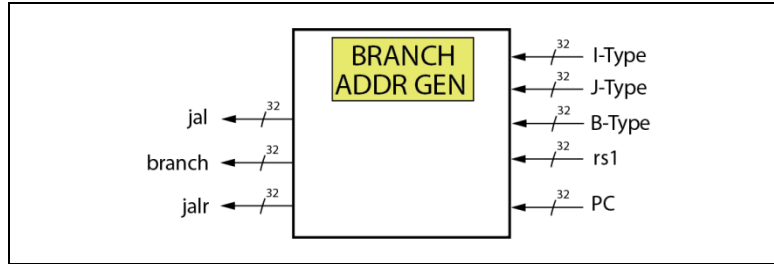


Figure 16.18: Block diagram of the BRANCH_ADDR_GEN module.

Table 16.17 shows the calculations required to convert the BRANCH_ADDR_GEN inputs to 32-bit values using three other forms of information including 1) relative addresses from the IMMED_GEN module (I-type, J-type, and B-type inputs), the program counter (PC), and, 3) one of outputs from the register file. From this table you can see that this modules main responsibility is to generate 32-bit absolute address from the relative addresses of the immediate values and the rs1 and PC registers. A few of the highly interesting things to note from Table 16.17 are:

- The branch-type and **jal** instruction include the value in the calculation. The hardware uses the **jal** instruction as a “branch to subroutine”, which means the instruction modifies the PC value to alter normal program flow.
- The **jalr** instruction does not include the PC value in the calculation because the RISC-V MCU uses the **jalr** instruction to return from subroutines. The hardware stores the return address of the subroutine in a register, which is why the **rs1** value appears in the absolute address calculation for the **jalr** instruction.

BRANCH_ADDR_GEN output	Comment	Calculation
jal	jump and link instruction	PC + J-Type
branch	address for all branch-type instructions	PC + B-Type
jalr	jump and link register instruction	rs1 + I-Type

Table 16.17: Calculation of BRANCH_ADDR_GEN outputs.

Similar the IMMED_GEN module, we provide the BRANCH_ADDR_GEN module for clarity. This section describes a set of functionalities that roughly share a similar purpose; it was convenient to call it a module and give that module a name. The important part of this module is thus the address calculations it forms, which people who are implementing the RISC-V MCU hardware can do without making a separate module.

16.8 The Register File (REG_FILE)

The register file provides storage for the operands associated with various “bit-crunching” operations in the RISC-V MCU. The register file is actually a RAM-type device despite have the word “register” in the module name. Conceptually speaking, the register file is a 32 x 32 RAM that performs the reads and writes typical of any RAM device. The RISC-V register file, however, has capabilities beyond a simple RAM, which is sometimes why we refer to it as a “dual port RAM”, or more aptly, a “multiport RAM”, or even “some freaking amazing RAM thang”.

The extra names for the register file RAM come from the fact that the register file must be able to simultaneously read two different values from RAM and write a third one. The RISC-V bases these “worst case” requirements on the R-type instructions, which read to register and write one register in the same instruction.

This means that the register file requires three 5-bit address inputs: two for reading and one for writing. The register file read operations are asynchronous, which means the read output data changes whenever the input addresses change. The register file write operations are synchronous, meaning the actual data written to the register file is synchronized to the rising edge of the register file's clock input.

The three 5-bit field codes associated with instruction formats provide the read and write addresses to the register file. The data written to the register file is provided by one of four sources including the ALU output (result of bit crunching operations), the memory output (result of store and input operations), or the PC (storage associated with program flow control). Figure 16.19 shows a black box diagram of the RISC-V MCU's register file; Table 16.18 provides a description of the register file's external interface (input and output signals).

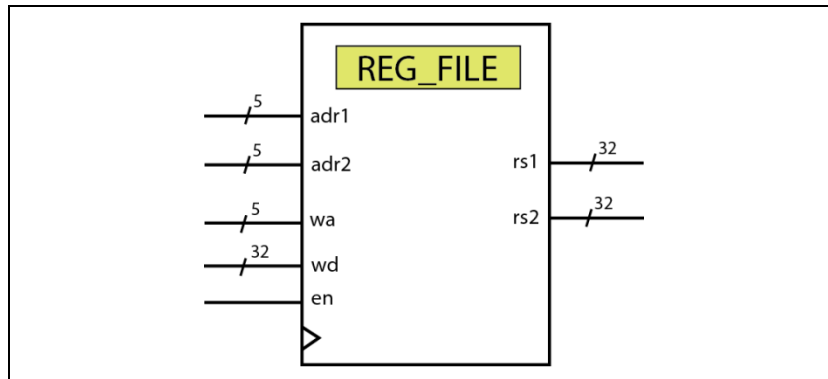


Figure 16.19: Block diagram of the RISC-V register file.

Signal	Type	Comment
adr1 adr2	in	The address lines associated with the rs1 & rs2 source outputs. These addresses are five bits wide to provide read access to the register file's 32 registers. The instruction register (output of program memory) provides data for these signals, which are five-bit field codes in the associated instruction formats. Register file reads are asynchronous.
wa	in	The address lines for destination address, which is the address of data written to the REG_FILE. These addresses are five bits wide to provide write access to the register file's 31 registers (excluding x0). The instruction register (output of program memory) provides data for this signal, which is a five-bit field code in the associated instruction formats.
wd	in	The data written to the register file at the address provided by wa . The data is 32-bit wide and is provided by one of several external sources.
en	in	The "en" (write enable) signal controls the writing of the wd data to the register file. Write register file write are synchronous.
clk	in	The system clock; register files reads are asynchronous while register file writes are synchronous, which means they happen the active edge of the clk (rising edge)
rs1 rs2	out	The data outputs associated with the adr1 & adr2 inputs, respectively. Data outputs are 32-bits wide, which match the width of the data inputs.

Table 16.18: Description of register file inputs and outputs.

16.9 The Arithmetic Logic Unit (ALU)

Arithmetic Logic Units (ALUs) are similar to the PC in that all computers have one in some form or another. ALUs are one of those names in computerland that have misleading meanings. While it's true the ALU performs arithmetic and logic, it most often performs other functions as well. Generally speaking, the ALU does all the

MCU's required bit crunching no matter what specific types of crunching the MCU requires. Thus, the main responsibility of the ALU is to implement the bit crunching operations required by the RISC-V instruction set. The ALU also performs various simple bit transfers as needed by the instruction set. The RISC-V OTTER MCU's ALU is a combinatorial circuit.

Figure 16.20 shows the schematic diagram of the ALU module and a few supporting modules. The ALU portion of the circuit is the sideways "A-shaped" thing, which is common approach to model ALUs in circuit schematics. The ALU has a 4-bit signal, **alu_fun**, that determines the operation the ALU performs. The **alu_fun** signal is an output from the CU_DCDR module, the combinatorial portion of the control unit. Based on the width of the **alu_fun** signal, the ALU can perform up to 16 different operations. The ALU performs operations on up to two 32-bit input operands and generates a 32-bit result on the **result** output.

The ALU supports the operations required by various instructions by tweaking the select signals on the Source A and Source B MUXes. The **alu_srcA** and **alu_srcB** select signals are control signals output from the CU_DCDR control unit module. Table 16.19 shows the list of ALU operations based on ordered values of the **alu_fun** signal select signal.

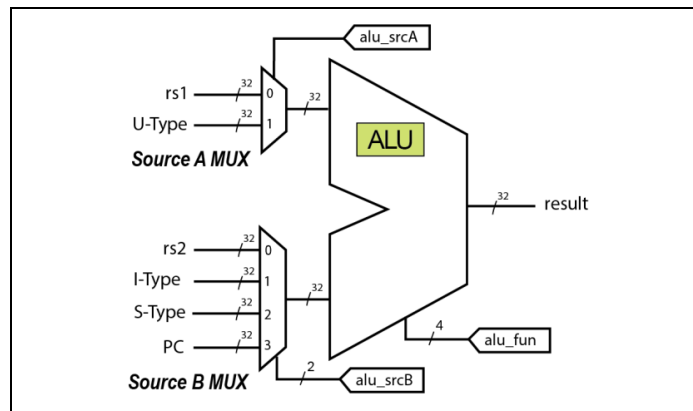


Figure 16.20: Block diagram of the ALU module and supporting modules.

Alu_fun	mnem	Description	RTL	Comment
0000	add	addition	$\text{result} \leftarrow \text{srcA} + \text{srcB}$	carry discarded
0001	sll	Shift left logical	$\text{result} \leftarrow \text{srcA} \ll \text{srcB}$	zero fills on right
0010	slt	Set if less than (signed)	$\text{result} \leftarrow (\text{srcA} <_s \text{srcB}) ? 1 : 0$	C notation
0011	sltu	Set if less than (unsigned)	$\text{result} \leftarrow (\text{srcA} <_u \text{srcB}) ? 1 : 0$	C notation
0100	xor	Logical bit-wise exclusive OR	$\text{result} \leftarrow \text{srcA} \wedge \text{srcB}$	-
0101	srl	Shift right logical	$\text{result} \leftarrow \text{srcA} \gg \text{srcB}$	zero fills on left
0110	or	Logical bit-wise inclusive OR	$\text{result} \leftarrow \text{srcA} \vee \text{srcB}$	-
0111	and	Logical bit-wise AND	$\text{result} \leftarrow \text{srcA} \cdot \text{srcB}$	-
1000	sub	Subtraction	$\text{result} \leftarrow \text{srcA} - \text{srcB}$	borrow discarded
1001	lui	Load upper immediate	$\text{result} \leftarrow \text{srcA}$	Copy
1011	-	<i>not currently used</i>	-	-
1100	-	<i>not currently used</i>	-	-
1101	sra	Shift right arithmetic	$\text{result} \leftarrow \text{srcA} \gg_s \text{srcB}$	sign fills on left
1110	-	<i>not currently used</i>	-	-
1111	-	<i>not currently used</i>	-	-

Table 16.19: List of ALU operations indexed with the `alu_fun` select signal.

Important to note here is that the ALU module does not provide any type of status signal regarding the result of any given ALU operation. Because of this, programmers are required to use the flexibility of the instruction set in order to determine items such as when an ALU operation overflows the 32-bit register width.

16.9.1 Addition and Subtraction

The RISC-V supports three addition and subtraction instructions including two-register argument forms of **add** and **sub**, and a register-immediate form of the **addi** instruction. These instructions perform addition and subtraction as you know and love them; Table 16.20 shows an overview of these instructions including the underlying bit formats. The associated assembly language manual includes a more complete description of these instructions. Here are a few other facts to note about these instructions:

- The hardware performs all operations on 32-bit registers.
- The assembler encodes the immediate value for the **addi** instruction as a 12 bit signed value, which provides a specifiable range of [-2048,2047]. The RISC-V hardware is responsible for performing the sign extension, which it does in the IMMED_GEN module. Table 16.21 shows the mapping the hardware uses to translate the 12-bit immediate field in an I-type instruction to a 32-bit value.
- The underlying hardware discards any overflow (carry-outs and borrows) from each of these operations. It is possible to do 64-bit math by examining the operands after the operation.

Instr format	Instruction Form	Instruction RTL	Example Usage	Comment
add	add rd, rs1, rs2	$rd \leftarrow rs1 + rs2$	add x11, x21, x31	addition
R-Type				
addi	addi rd, rs1, imm	$rd \leftarrow rs1 + sext(imm)$	addi x7, x8, 0x0F	subtraction
I-Type				
sub	sub rd, rs1, rs2	$rd \leftarrow rs1 - rs2$	sub x15, x14, x17	
R-Type				

Table 16.20: The addition and subtraction instructions.

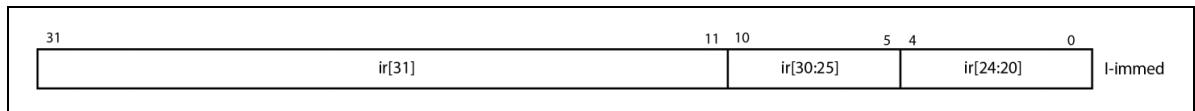


Table 16.21: 32-bit immediate format for I-type instructions.

16.9.2 Shifting Instructions

The RISC-V supports both logical and arithmetic shifts⁴. Instructions supporting logical shifts are available for shifting in both directions; the RISC-V ISA only support arithmetic shifts for right shifts. Both logical and arithmetic shifts are available in both register and immediate forms. We typically use shifting instructions for integer math operations, which supports fast multiples (shift lefts) and divides (shift rights) by powers of two. We typically use logical shifts for unsigned arithmetic and arithmetic shifts for signed arithmetic.

All of the six shift-type instructions are three operand instructions. Each of the six instructions performs shifts on register values and stores the results in register. The third operand specifies the number of bit positions to shift, which effectively makes these instructions fully capable of barrel shifts. The logical-type shift instructions fill vacated bit positions with zero while the arithmetic-type shift instructions consider the register being shifted as a signed value and fill the vacated bit positions with the sign bit of the register.

The RISC-V hardware limits shifting to 32 bits, which the underlying instruction format can represent with a 5-bit value. The instructions thus use only the five LSBs from right-most operand, which programmers can specify by either a register or immediate value depending on the instruction type. Bits that the hardware shifts out of registers by the set of shift instructions are gone forever; there is no available hardware to store any of the shifted out bits. Table 16.22 and Table 16.23 provide an overview of the pertinent information associated with the logical and arithmetic shift instruction, respectively.

⁴ The RISC-V ISA only supports arithmetic right shifts.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
sll	sll rd,rs1,rs2	$rd \leftarrow rs1 \ll rs2[4:0]$	sll x11,x21,x31	logical shift left zero filled
R-Type				
slli	slli rd,rs1,imm	$rd \leftarrow rs1 \ll imm[4:0]$	slli x7,x8,0x0F	logical shift left zero filled
I-Type				
srl	srl rd,rs1,rs2	$rd \leftarrow rs1 \gg rs2[4:0]$	srl x11,x21,x31	logical shift right zero filled
R-Type				
srlrli	srlrli rd,rs1,imm	$rd \leftarrow rs1 \gg imm[4:0]$	srlrli x7,x8,0x0F	logical shift right zero filled
I-Type				

Table 16.22: The logical shift left & right instructions.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
sra	sra rd,rs1,rs2	$rd \leftarrow rs1 \gg rs2[4:0]$	sra x11,x21,x31	Arithmetic shift right Sign filled
R-Type				
srairai	srairai rd,rs1,imm	$rd \leftarrow rs1 \gg imm[4:0]$	srairai x7,x8,0x0F	Arithmetic shift right Sign filled
I-Type				

Table 16.23: The logic shift right instructions.

16.9.3 Logic Instructions

The RISC-V MCU contains has standard Boolean logic instructions including AND, OR, and XOR (exclusive OR). There are both register and immediate versions of the three logic-type instructions. These instructions perform bitwise Boolean logic operations on the instruction operands. Table 16.24 shows an overview of these instructions including the underlying bit formats. The associated assembly language manual includes a more complete description of these instructions. Here are a few other fun facts to chew on:

- The hardware performs all operations on 32-bit registers.
- The assembler encodes the immediate values for the immediate versions of these instructions as 12-bit values, which are “sign extended” to create a 32-bit operand. In these cases, the RISC-V hardware interprets the left-most bit as the sign bit and then performs the sign extension using the

IMMED_GEN module. Table 16.21 shows the mapping the hardware uses to translate the 12-bit immediate field in an I-type instruction to a 32-bit value.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
and	and rd,rs1,rs2	$rd \leftarrow rs1 \cdot rs2$	and x11,x21,x31	bitwise AND
R-Type				
andi	andi rd,rs1,imm	$rd \leftarrow rs1 \cdot sext(imm)$	andi x7,x8,0x0F	bitwise AND sign extend imm
I-Type				
or	or rd,rs1,rs2	$rd \leftarrow rs1 + rs2$	or x11,x21,x31	bitwise OR
R-Type				
ori	ori rd,rs1,imm	$rd \leftarrow rs1 + sext(imm)$	ori x7,x8,0x0F	bitwise OR sign extend imm
I-Type				
xor	xor rd,rs1,rs2	$rd \leftarrow rs1 \wedge rs2$	xor x11,x21,x31	bitwise XOR
R-Type				
xori	xori rd,rs1,imm	$rd \leftarrow rs1 \wedge sext(imm)$	xori x7,x8,0x0F	bitwise XOR sign extend imm
I-Type				

Table 16.24: The Boolean logic-based instructions.

16.9.4 Set-If-Less-Than Instructions

The RISC-V ISA has two types of instructions that perform comparisons. The branch-type instructions perform comparisons based on two register and are program flow control instructions. We consider the branch-type instructions program flow control because they have the ability to branch based on the conditions in the registers and the given branch instruction. The RISC-V ISA can also perform comparisons using the slt-type instructions. Table 16.25 shows all the details of the two flavors of slt-type instructions. The slt-type instructions have two significant differences from the branch-type instructions:

- 1) slt-type instructions are not program flow control instructions, meaning the MCU always executes the instruction following slt-type instruction. This is of course a fancy way of saying that the slt-type instruction never branch under any conditions. The result of this difference is that the hardware uses a different module to perform the comparison. The RISC-V OTTER uses the ALU to perform comparisons for the slt-type instructions (and also the setting or clearing of the destination register). The ALU output, or result, for the slt-type instruction is either a ‘1’ or ‘0’ based on whether the condition was true or not, respectively.
- 2) There are two flavors of slt-type instructions: one type is similar to branch-type instructions where the given comparison is between two register values. The other flavor of slt-type

instruction allows a comparison between a register and immediate value. The good news is that programmers don't have to stick the both compare values into registers; the bad news is that the compare value in the immediate flavor of the slt-type instruction only contains a 12-bit field for the immediate value. This means if you need to compare larger values, you must use the two-register version of the instruction.

Instr type	Instruction Form	Instruction RTL	Example Usage	Comment
slt	slt rd,rs1,rs2	$rd \leftarrow (rs1 <_s rs2) ? 1 : 0$	slt x10,x12,x13	Set if less than (signed)
R-Type				
slti	slti rd,rs1,imm	$rd \leftarrow (rs1 <_s sext(imm)) ? 1 : 0$	slti x10,x12,23	Set if less than imm (signed)
I-Type				
sltu	sltu rd,rs1,rs2	$rd \leftarrow (rs1 <_u rs2) ? 1 : 0$	sltu x10,x8,x9	Set if less than (unsigned)
R-Type				
sltiu	sltiu rd,rs1,imm	$rd \leftarrow (rs1 <_u sext(imm)) ? 1 : 0$	sltiu x10,x8,78	Set if less than imm (unsigned)
I-Type				

Table 16.25: The slt-type instructions.

16.10 The Branch Condition Generator (BRANCH_COND_GEN)

The BRANCH_COND_GEN is a module that provides branch information to the CU_DCDR. The BRANCH_COND_GEN thus supports the six base instructions in the RISC-V ISA. All six branch instructions are B-type instructions. Figure 16.21 shows the schematic diagram for the BRANCH_COND_GEN. The two 32-bit inputs to the module are two source register outputs from the register file; the three outputs represent status signals describing the numerical relations between the two inputs; these become inputs to the CU_DCDR. Table 16.26 provides a more detail description of the modules inputs and outputs. Based on the description in Table 16.26, the BRANCH_COND_GEN is nothing more than a comparator that has the ability to do signed and unsigned comparisons.

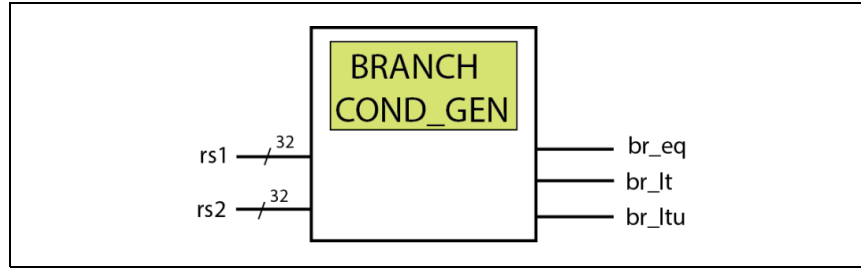


Figure 16.21: Black box diagram of the CU_DCDCR module.

Signal	Type	Comment
rs1	in	32-bit output of the register file (number 1 output)
rs2	in	32-bit output of the register file (number 2 output)
br_eq	out	1-bit output indicating if $rs1 == rs2$;
br_lt	out	1-bit output indicating if $rs1 < rs2$; result based on operands being signed
br_ltu	out	1-bit output indicating if $rs1 < rs2$; result based on operands being unsigned

Table 16.26: Description of register file inputs and outputs.

Table 16.27 shows a diagram of the B-type format, which supports the fact that the RISC-V bases all branch instructions on the result of a comparison between the **rs1** & **rs2** source registers. This means that you need to perform a branch based on an immediate value, you must first place that value into a register. The opcode field is the same for all B-type instruction; the hardware differentiates the branch-type instructions using the **funct3** opcode field.

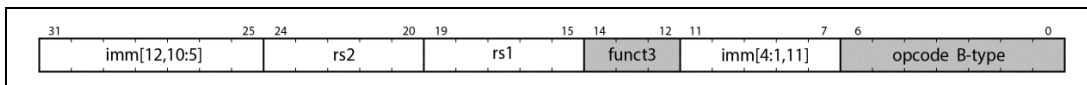


Table 16.27: B-type instruction format.

The branch-type instructions require that programmers first must place the two values it compares into registers. Placing values to compare into registers does seem somewhat limiting at first, but the **slt** and **slti** instructions provide somewhat of a workaround of this limitation. Since the slt-type instructions are not program flow instructions (meaning they can't branch), the BRANCH_COND_GEN does not make the comparison for those instructions and the CU_DCDCR does not have the option to implement a branch. The ALU module performs the comparison required by slt-type instructions.

Table 16.28 shows the relation between the BRANCH_COND_GEN outputs and the associated base branch instructions in the RISC-V ISA. What this table shows is the condition the CU_DCDCR checks in the context of each base branch instruction to determine whether it takes the branch or not. The CU_DCDCR is effectively controlling which value the PC loads based on the result of the comparison made in the BRANCH_COND_GEN. Table 16.28 also shows that for any given branch instruction, the CU_DCDCR only considers one of the three outputs of the BRANCH_COND_GEN. Note that the two possible signal values for each output support the six base branch-type instructions.

Instruction	True Condition Indicator		
	br_eq	br_lu	br_ltu
beq	1	-	-
bne	0	-	-
blt	-	1	-
bge	-	0	-
bltu	-	-	1
bgeu	-	-	0

Table 16.28: Output indicators associated with individual base branch instructions.

16.11 Then Control and Status Registers (CSR)

The control and status register (CSR) currently is only associated with the RISC-V MCU interrupt architecture. For this reason, we won't describe the CSR in this chapter, and delay that description until Chapter 18, which is the chapter on the RISC-V Interrupt architecture.

16.12 The RISC-V MCU Wrapper

I'm not actually sure where the term "wrapper" came from. Someone, possibly even me, may have made it up for all I can remember. Despite its dubious origins, the notion of a wrapper is rather important in the land of softcore MCUs. The sole purpose of the wrapper is to provide an interface between the MCU and the development board you implement it on. The MCU is a generic module in terms of its basic design, which means it is flexible enough to solve many different types of problems. The wrapper provides the interface between an outside world that has a given set of hardware (such as on a development board) and the generic MCU. In the context of this discussion, the notion of genericity refers to the MCU's generic I/O interface.

The notion of the wrapper is important for two reasons. First, because it helps you understand how to interface a softcore MCU such as the RISC-V MCU with a development board. Second, because it helps you understand how the input and output instructions work with the MCU and how the wrapper to actually implements I/O on the RISC-V MCU. For these reasons, you should make sure you understand everything all aspects of the RISC-V MCU wrapper.

The main purpose of a development board is to provide a modest set of input and output devices that you can use to interface with the RISC-V MCU. This being the case, we have three types of I/O on typical development board: 1) input devices (such as a buttons), 2) output devices (such as an LED), and 3) generic pins (meaning you can assign the pins as either being an input or an output and associate them with an external device). For this discussion, we only mention the actual physical devices on the development board, though you can use the generic pins to connect any external peripheral to the system such that the MCU can monitor/control them.

The mission for the Wrapper is to interface the RISC-V MCU with the development board. We currently use the Basys3 development board, which contains an FPGA-type PLD. Figure 16.22(a) shows a partial block diagram of the dev board; we refer to this as partial because there are some features on the board that we omit. The wrapper is the highest level of the design hierarchy that includes the RISC-V OTTER MCU. Figure 16.22(b) shows the black box diagram of the RISC-V OTTER MCU. Our mission is to use the Wrapper to interface with the RISC-V OTTER MCU in such a way as to create a working computer. There are many ways to do this; we present a generic approach with full explanation in this section.

Figure 16.22(a) shows the black box diagram of the Basys3 development board. The PLD on the development board is an FPGA, which interfaces with various input and output devices on the development board. This board includes 16 inputs connected to switches (SWITCHES), five inputs connected to buttons (BUTTONS) and an external clock signal (CLK). The board contains 16 output connected to LEDs (LEDS), eight outputs connected to the segments of four 7-segment display devices (SEGMENTS) and four outputs connected to the anodes associated with the four 7-segment display (ANODES). Figure 16.22 does not list the various generic signals available for input or output on the board. Additionally, the LEDS output are positive logic while the

SEGMENTS and ANODES are both negative logic. The switches in the “up” position and pressed buttons generate a ‘1’ to the onboard FPGA, while switches in the down position and unpressed buttons generate a ‘0’.

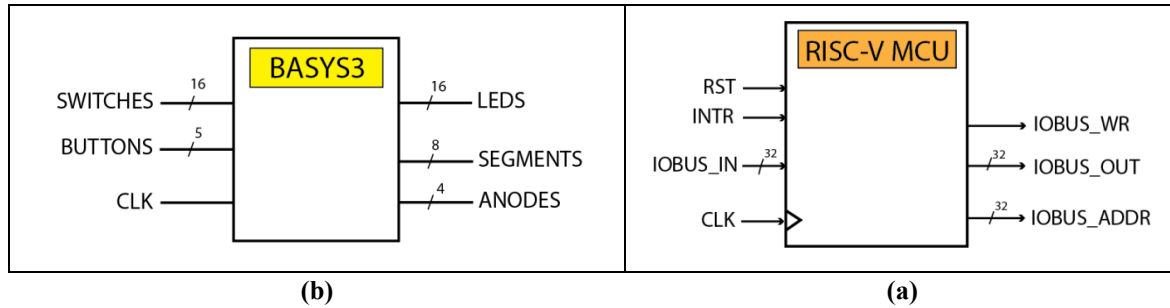


Figure 16.22: The block diagram for the Basys3 development board (a), and the RISC-V OTTER MCU.

We implement the Wrapper as a simple Verilog model; you can find the full model in the appendix. The approach we’ll take here is to describe the more important parts of the wrapper model by examining the Verilog model and relating that model to real world digital parts that you know and love. If you are using the RISC-V OTTER MCU or any other version of the RISC-V MCU, the information contained in this section allows you to understand the interface requirements for your given device. The approach to interfacing a softcore MCU on a development board is the same; the particular I/O available on a given development board is probably different.

16.12.1 Wrapper External Device Addressing

Because the underlying RISC-V MCU needs to interface with the external I/O devices, the Wrapper and RISC-V MCU need to agree on a “port ID” for each I/O device. The RISC-V MCU uses these port_IDs as “port addresses”, or simply “addresses” in the memory-mapped I/O architecture. These addresses are for the most part arbitrary, but they do need to fall into the I/O portion of the RISC-V MCU address space (memory map), which is 0x1100000 and above for the RISC-V OTTER. Figure 16.23 shows the list of port IDs for the I/O devices on the development board. We assign the port IDs using the Verilog `localparam` in order to facilitate future hardware changes.

```

// - INPUT PORT IDS -----
localparam SWITCHES_PORT_ADDR = 32'h11008000; // 0x1100_8000
localparam BUTTONS_PORT_ADDR = 32'h11008004; // 0x1100_8004

// - OUTPUT PORT IDS -----
localparam LEDS_PORT_ADDR     = 32'h1100C000; // 0x1100_C000
localparam SEGS_PORT_ADDR     = 32'h1100C004; // 0x1100_C004
localparam ANODES_PORT_ADDR   = 32'h1100C008; // 0x1100_C008

```

Figure 16.23: Fragment of wrapper showing assigned port_IDs.

16.12.2 Wrapper Input Circuitry

The RISC-V MCU has one signal (a bundle) that handles all the possible input to the RISC-V MCU from external devices. The `IOBUS_IN` signal is the 32-bit input to the RISC-V MCU that “accepts” data from the outside world and writes that data to a register in the register file. The RISC-V MCU is a general purpose MCU so it is quite versatile; part of that versatility includes being able to interface with a high number of external inputs, but not more than 32 at a time. The width of the `IOBUS_IN` signal governs how many inputs the RISC-V can input in one operation, which is driven by a load instruction executed by the MCU. In this way, the wrapper includes the circuitry to allow the RISC-V MCU to choose which input it wants input, which it does by using the proper port_ID as listed in Figure 16.23.

Anytime we need to “choose” something in hardware, we use a MUX. We thus use a MUX to determine which of the external input devices transfers their data from the MUX input to the MUX output, and thus connect to the

IOBUS_IN input on the RISC-V MCU. Figure 16.24 shows the Verilog code that the wrapper uses to handle external input devices. Here are the important features to notice about the model fragment in Figure 16.24:

- The code models a combinatorial circuit, which we know primary because of the “**always_comb**” choice of procedural block. We also know this because the body of the procedural block uses a case statement that includes a default case. The **always_comb** is actually a System Verilog construct; we could also implement the model using an **always** construct using Verilog.
- The variable expression in the case statement is the **IOBUS_addr** signal, which the I/O address output from the RISC-V MCU. This variable address is the absolute memory address associated with the load-type instruction output executed by the program running on the RISC-V MCU. Thus, the **IOBUS_addr** signal becomes the select input for the MUX modeled in Figure 16.24. Figure 16.25 shows the BBD associated with the interfacing the Wrapper to the RISC-V MCU; this diagram includes the input MUX.
- When the data on the **IOBUS_addr** signal matches a port_ID associated with an external input device, the MUX assigns the output from that device to the IOBUS_in signal. Because the **IOBUS_in** signal is 32-bits wide and inputs are often less bits, the MUX model clears all unused input puts by pre-assigning the IOBUS_in value to zero. The actual data input signal overwrites any pre-assigned value of zero.

```

always_comb
begin
    IOBUS_in=32'b0;
    case (IOBUS_addr)
        SWITCHES_PORT_ADDR : IOBUS_in[15:0] = switches;
        BUTTONS_PORT_ADDR  : IOBUS_in[4:0]  = buttons;
        default: IOBUS_in=32'b0;
    endcase
end

```

Figure 16.24: Fragment of wrapper code showing the input model.

Figure 16.25 shows a high-level diagram of the hardware associated with the RISC-V MCU wrapper. This diagram provides an overview of the wrapper hardware including the input MUX. Note that the input MUX uses the **IOBUS_addr** signal output from the RISC-V MCU as select signals to the MUX.

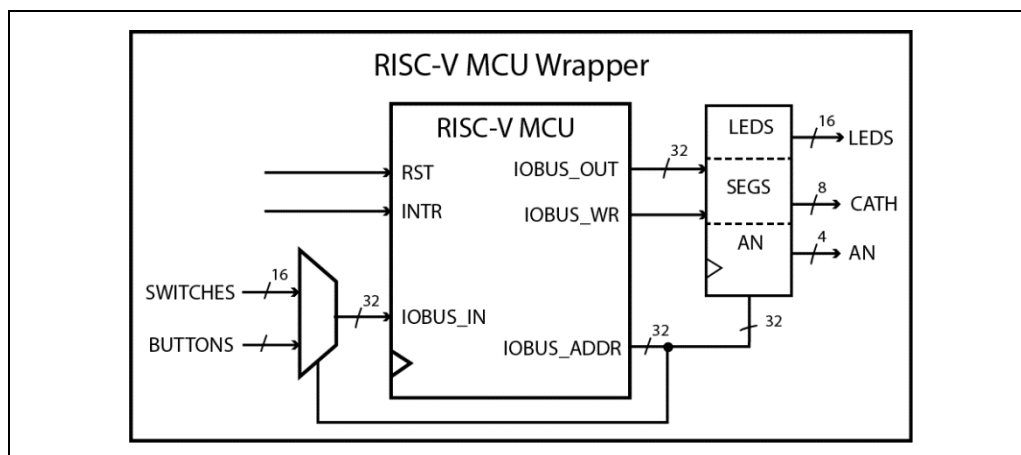


Figure 16.25: Block diagram of the RISC-V MCU wrapper and internal modules.

16.12.3 Wrapper Output Circuitry

The RISC-V MCU has one signal (a bundle) to handle all the possible output to external devices. The **IOBUS_out** is an output signal from the RISC-V MCU. The data associated with all store operations sent from a register in the register file. The **rs2** output of the register file serves as the **IOBUS_out** signal for the RISC-V MCU. The RISC-V MCU shares the addresses lines for output with the input data (**IOBUS_in**); this works because it is not possible to simultaneously perform both an input (load) and an output (store) in the RISC-V MCU hardware.

Despite the fact that the output circuitry uses the same “selection” signals as the input circuitry (Figure 16.25), the circuitry is significantly different. The reason is that data outputs from the RISC-V MCU module are “temporary”, meaning that data, address, and control signals for the output operation only exist for the execute cycle of the associated store instruction that generated them. These outputs are typically not useable for any circuit depending on the outputs. The solution is similar to the inputs, where the input data is “stored” somewhere, which means the RISC-V MCU stores data input from external sources into a register in the register file. The Wrapper must then include registers that store the data output from the RISC-V MCU as a result of executing store-type instructions configured to be output. Each output device on the dev board must have corresponding registers in the wrapper circuitry. Figure 16.25(b) includes three registers to handle dev board output, which we list on **AN**, **LEDS**, and **SEGS**.

Store instructions perform an absolute address calculation in hardware as part of the instruction. The absolute address then becomes an input to the memory module. One of the source registers (the **rs2** output of the register file) that is part of the store instruction provides the data associated with the store instruction. This source data becomes the data written to the memory or the data output to external devices based solely on the value of the absolute address associated with the store instruction. The hardware writes the data to memory if the absolute address is 0x0000FFFF or lower; otherwise, the data is “output” to external devices. There is no difference in the data output as the register file output (**rs2**) serves as both the data input to memory and the **IOBUS_out** signal.

The only difference in the memory writes and data outputs is in how the main memory module in the RISC-V MCU interprets the absolute address. The hardware writes the data on the **DIN2** input to memory if the write enable (**WE2**) is asserted and the value on the **ADDR2** input (the **rs2** output from the register file) is less than 0x00010000. If the write enable is asserted and the value on the **ADDR2** input is greater than 0x0000FFFF, the memory module asserts the **IO_WR** output. This output remains asserted for the duration of the execute cycle associated with the underlying store instruction.

The **IO_WR** signal is an output signal from the RISC-V MCU. When the RISC-V MCU executes an output instruction (a store instruction with an address value in I/O space), the MCU asserts this signal to indicate to external circuitry that the MCU is implementing an output operation. We often refer to the **IO_WR** signal as a “write pulse”, because the signal is only asserted for one clock cycle (the duration of the execute cycle of the associated store instruction). The external circuitry then uses the **IO_WR** signal as a write enable for devices such as registers. The RISC-V MCU Wrapper stores all data outputs in registers, which provides the data with “persistence”. This means when you write to a specific output device, the Wrapper circuitry latches that data to the external registers on the wrapper level using the **IO_WR** signal as the write enable to those registers. If we did not store the output data in registers, the output data would essentially disappear after the MCU completes executing the store instruction, which makes the data hard to use based on the relatively short time the MCU makes that data available.

Figure 16.26 shows the portion of the wrapper model that handles outputs. The code has two main purposes. First, it generates three registers, one for the LED, cathodes (segments), and anodes, which are the three output devices on the development board. Second, the “chooses” which register the data is written to using the **IOBUS_addr** signal to discern the proper external output device. Figure 16.25 shows the output hardware in a high-level flavor, but clearly shows that the latching of data output from the RISC-V MCU is a function of **IOBUS_addr**, **IO_WR**, **IOBUS_out**, and the clock edge. Here are a few more items to note about this code.

- We included declarations for the three required registers. We use a *logic* type for the declaration and provide a “r_” prefix in the associated label⁵. Using the “r_” prefix indicates to the human

⁵ The logic type is a feature of System Verilog, but not Verilog. If you were modeling this using Verilog, you would use either a register or wire-type.

reader that the model uses the declared signal as a register type, which is a great form of self-commenting that all good hardware modelers use.

- We use an “**always_ff**” block to model the actual input circuitry, which verifies the block actually models registers. The **always_ff** is another System Verilog construct that can easily be replaced by a Verilog **always** construct. The fact that we use the **posedge** function in the sensitivity list ensures that the model generates synchronous sequential elements.
- The model only latches data to the registers when the **IOBUS_wr** signal is asserted. The **IOBUS_wr** is the external connection associated with the **IO_WR** signal generated by the main memory when the MCU executes an output instruction.
- The **IOBUS_addr** signal is the address value associated with the underlying store instruction. If this address matches one of the port_IDs (port addresses) associated with an output device, the data on the **IOBUS_out** signal (which is the signal from the register file’s rs2 output) is latched into the correct register on the next active clock edge.
- Overall, the code in Figure 16.26 models three registers and a generic decoder. The **IOBUS_addr** signal becomes the select input to the decoder; the decoder selects which register latches the **IOBUS_out** data.

```

// - register for dev board output devices -----
logic [7:0] r_segs; // register for segments (cathodes)
logic [15:0] r_leds; // register for LEDs
logic [3:0] r_an; // register for display enables (anodes)

always_ff @ (posedge s_clk)
begin
  if (IOBUS_wr == 1)
    begin
      case (IOBUS_addr)
        LEDS_PORT_ADDR: r_leds <= IOBUS_out[15:0];
        SEGS_PORT_ADDR: r_segs <= IOBUS_out[7:0];
        ANODES_PORT_ADDR: r_an <= IOBUS_out[3:0];
        default: r_leds <= 0;
      endcase
    end
  end
end

```

Figure 16.26: Fragment of wrapper code showing the output model.

Example 16.3: Maximum Control and Status Bits

How many unique status bits of can the current architecture read? Also, how many bits of unique output can the current architecture write.

Solution: The best place to start on a problem like this is to examine the top-level RISC-V architecture diagram, which we conveniently provide in Figure 16.27. The first thing to note is that is that the RISC-V has 100 inputs and outputs, 98 of those (all but the RST and CLK) have to do with I/O. Another thing to note is that the RISC-V designers made the MCU as flexible and extensible as possible, meaning that if you needs to control either five or 500 signals, the RISC-V architecture can easily handles and do so in a generic manner.

Input: There are 32 bits associated with input (IOBUS_IN), which means every different input operation can input 32 unique bits. The question is how many unique sets of 32 bits can the architecture manage? Recall that the various inputs connect to a MUX in the wrapper; the IOBUS_ADDR signal acts as the control inputs to the MUX. The next question is how many unique address values can the IOBUS_ADDR generate? The answer is embroiled with the notion of the RISC-V MCU’s memory mapped I/O (MMIO). The RISC-V has a 32-bit

address space, but the addresses range $[0x00000000, 0x0000FFFF]$ is associated with memory access instructions, which leaves the remaining addresses associated with I/O. Each of the unique addresses can input 32, or 2^5 unique bits, so the final answer is:

$$(2^{32} - 2^{16}) * 2^5 = \text{a lot of bit (you do the math)}.$$

Output: Calculating the number of unique output bits is similar to the number of unique input bits. The IOBUS_OUT signal controls the number of outputs, which is 32. This means for any single output, the RISC-V can output 32 unique bits. The output also uses the IOBUS_ADDR to “decode” the bits the output are sent to. The IOBUS_ADDR is constrained by the RISC-V MCU’s MMIO, which gives the effective address space for outputs the same as the effect address space for inputs: $[0x00010000, 0xFFFFFFFF]$. This range provides $(2^{32} - 2^{16})$ unique values, so the total number of output bits the RISC-V can control is $(2^{32} - 2^{16}) * 2^5 = \text{many bits}$.

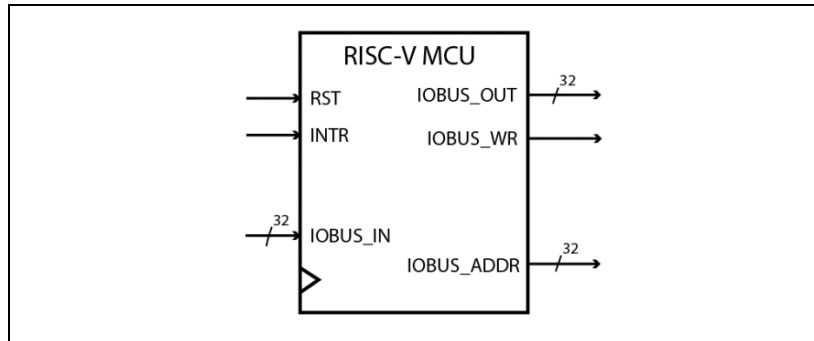


Figure 16.27: The Control Unit FSM black box diagram.

Example 16.4: Maximum Control and Status Bits

The RISC-VMCU physical memory address space was increased from 16 bit to 18 bits. What is the resulting number of bits the RISC-V MCU can both input and output?

Solution: In this problem, the address space grows by two bits up to 18 bits. This means that the memory address range is now $[0x00000000, 0x0003FFFF]$, which leaves the I/O address range based on 14 bits: $[0x00040000, 0xFFFFFFFF]$. Based on the RISC-V architecture, the input and output can both manage the same number of unique bits, which is now based on a 14-bit address space. The final number of bits for both input and output are $(2^{32} - 2^{18}) * 2^5$. That’s a lot of control (unwarranted editorial).

16.13 Chapter Summary

- The RISC-V OTTER MCU is a relatively complex digital circuit that we can easily subdivide into a set of smaller modules.
- *Mealy's First and Only Law of Computer Programming*: If you understand the hardware of the computer your program will run on, then you can write better programs.
- The RISC-V OTTER MCU has two modules that control the basic operations of the underlying hardware: the CU_FSM and the CU_DCDR. The CU_FSM is a finite state machine (and thus a sequential circuit) that sequences through the states associated with instruction execution. The CU_DCDR is a decoder (and thus a combinatorial circuit) that provides a set of signals that controls the operation of various MUXes in the MCU.
- Most instructions require two clock cycle for for execution, but load-type instructions require three clock cycles. Instruction execution includes names for the states in the underlying FSM, which are fetch and execute cycles for all instructions and a writeback state for load-type instructions. The fetch cycle roughly *fetches* an instruction from program memory, the execute cycle roughly executes that instruction, and the writeback cycle retrieves data from memory for load-type instructions.
- The program counter (PC) stores the address of the current instruction being executed. The PC is implemented as a 32-bit register with external support on the current RISC-V implementation. The PC addresses physical memory in the RISC-V OTTER, but because program memory uses 32-bit instruction, the program memory only requires the 14 most significant bits of the PC address.
- The PC supports basic computer operations by loading a new value into the PC after executing an instruction. The new value can either be the address of the next instruction (normal operation) or the address associated with a branch-type instruction (both conditional and unconditional).
- The RISC-V MCU main memory represents the total address space for the computer. Although main memory is considered to be $2^{32} \times 8$, physical memory is only $2^{16} \times 8$, where physical memory is the memory where data can actually be stored. The other part of the address space is reserved for other operations such as I/O. The physical memory stores the program and other data; the other data includes items such as the stack.
- The RISC-V MCU uses memory-mapped I/O (MMIO). The RISC-V MCU interprets all memory accesses above the physical memory limit 0x000FFFFF as I/O.
- The IMMED_GEN module creates 32-bit values from the smaller values and often strangely configured values associated with the immediate fields in the RISC-V instructions.
- The BRANCH_ADDR_GEN module uses a combination of register data, PC values, and values output from the IMMED_GEN module in order to create 32-bit absolute address. The values the BRANCH_ADDR_GEN creates are loaded into the PC for conditional and unconditional branch instructions.
- The REG_FILE module stores the 32 32-bit general purpose register in the RISC-V MCU. All bit crunching operations in the RISC-V MCU are done using registers in the register files. All memory load operations are from main memory to the register file register; all memory store operations are from register file register to main memory. All input operations are from the outside world to registers; all output operations are from register to the outside world.
- The ALU module performs all bit-crunching operations. Although the name implies arithmetic and logic type operations, ALUs typically do other operations as well. The ALU inputs two 32-bit operands and outputs a 32-bit result based on the selected operation.
- The BRANCH_COND_GEN module controls the branches associated with branch-type instructions in the RISC-V MCU (which does not include slt-type instructions). This module inputs the two register outputs from the REG_FILE module and generates three output signals based on comparisons of these signals in the BRANCH_COND_GEN module. The CU_DCDR uses these three signals to control whether the MCU takes a branch or not.

- The Wrapper is a model that interfaces the MCU to a given development board. Development boards generally have a given set of input and output device; the Wrapper makes these devices available to the I/O operations of the underlying MCU.
 - Development board inputs are “selected” via a MUX to enter the MCU; the selection signals are the I/O address signals output from the MCU. Development board outputs are registered on the Wrapper level so that they are persistent and can more easily be used by external hardware devices. The I/O address signals in conjunction with the IO_WR pulse output from the MCU control which register will receive the data output from the MCU.
-

16.14 Chapter Exercises

- 1) Briefly describe why Mealy's One and Only Law of Computer Programming is patently obvious.
- 2) Briefly describe how we use the notion of hierarchical design to understand the RISC-V MCU.
- 3) Briefly describe why the CU_FSM and CU_DCDR are separate modules.
- 4) Briefly describe the main responsibility of the CU_FSM.
- 5) Briefly describe the function that the **memRDEN1** and **memRDEN2** signals serve.
- 6) Briefly explain why the CU_FSM contains a clock input but the CU_DCDR does not.
- 7) Briefly describe why the CU_FSM contains a **memWE2** signal but not a **memWE1** signal.
- 8) We state that the program memory is not writable, but it actually is writeable. Briefly describe why you are able to write to program memory and briefly describe how exactly to do it.
- 9) Each of the CU_DCDR outputs have a special commonality; what is that commonality?
- 10) Briefly explain how long the CU_DCDR's output remain unchanged for a given instruction.
- 11) The outputs of the CU_DCDR in the fetch cycle still have the outputs associated with the previous instruction. Briefly describe why this is so and why it is OK.
- 12) Briefly describe whether the CU_DCDR knows anything about instruction cycles.
- 13) Briefly describe why the CU_DCDR does not need to do anything different for load-type and all other instructions.
- 14) How many clock cycles does it require for the following RISC-V assembly language code fragment to execute from the starting at the **start** label and going through the **done** label?

```

start:    add    x10,x0,x0
          addi   x10,x10,8

loop:    beq    x10,x0,done
          lw     x20,0(x21)
          sw     x21,4(x23)
          sw     x21,4(x23)
          addi   x10,x10,-1
          slt   x23,x24,x35
          j     loop

done:    nop

```

- 15) How many clock cycles does it require for the following RISC-V assembly language code fragment to execute from the starting at the **go** label and going through the **stop** label?

```

go:      add    x10,x0,x0
          addi   x10,x10,0x12

loop:    beq    x10,x0,stop
          lw     x20,0(x21)
          lw     x22,0(x31)
          addi   x10,x10,-1
          addi   x21,x21,1
          addi   x31,x31,4
          xor    x23,x24,x25
          j     loop

stop:    nop

```

- 16) How many clock cycles does it require for the following RISC-V assembly language code fragment to execute from the starting at the **go** label and going through the **stop** label?

```

go:      add    x10,x0,x0
         addi   x10,x10,14

loop:    sw     x23,0(x21)
         lw     x22,0(x31)
         addi   x10,x10,-1
         ori    x21,x21,1
         addi   x31,x31,4
         xor    x23,x24,x25
         beq   x10,x0,loop

stop:    nop

```

- 17) Briefly describe why the PC is a register and not a counter in the RISC-V OTTER MCU architecture.
- 18) Briefly describe why the PC is 32-bits wide but only 14 of those bits are used to access instructions in program memory.
- 19) Briefly describe why programmers can use either **jal** or **jalr** instructions to call subroutines.
- 20) Briefly describe why programmers can use a **jalr** instruction but not a **jal** instruction to return from a subroutine.
- 21) Briefly describe why the program memory is sometimes listed as 14k x 32, and other times listed as 16k x 8.
- 22) Briefly describe why the immediate fields in the branch instruction formats are stored using a strange ordering.
- 23) Briefly describe what entity forms the immediate values associated with branch instruction.
- 24) Briefly describe what entity forms the absolute branch addresses from the signed immediate values associated with branch-type instructions.
- 25) Briefly describe how it is that the RISC-V MCU can branch twice as far as the immediate value associated with a branch instruction seems to indicate.
- 26) The RISC-V OTTER MCU memory module serves three functions: what are they?
- 27) What is the byte and word capacity of the RISC-V OTTER MCU main memory?
- 28) If the stack pointer grew larger than 0x0000FFFF, briefly describe what would happen if a program attempted to push a value onto the stack.
- 29) Briefly describe the relationship between main memory and physical memory.
- 30) Briefly describe whether program memory is writeable under program control.
- 31) Briefly describe the potential problem associated with changing data in program memory.
- 32) Briefly describe whether you can push and pop data from the code segment.
- 33) Briefly describe whether the memory reads and memory writes on the RISC-V MCU are synchronous or not. Your answer should include both the main memory and the register file.
- 34) Memory access instructions in the RISC-V MCU don't officially use the lower two bits of the memory address lines. This being the case, briefly describe how the hardware is able to perform reads of individual bytes from any four-byte chunk of memory data.
- 35) Briefly describe why the RISC-V instruction set includes five load-type instructions but only three store-type instructions.
- 36) Briefly describe what or who determines whether a memory access-type instruction will perform a memory access or an input/output operation.

- 37) We generally speak of the PC as being the address of the current instruction being executed; briefly describe why this definition is not always 100% accurate.
- 38) Briefly describe the purpose the IO_WR signal serves to the RISC-V MCU hardware. Be careful, this is a trick question.
- 39) Briefly explain whether the RISC-V assembler knows the difference between a memory access instruction and an I/O instruction.
- 40) Briefly explain whether the RISC-V hardware (not including the main memory module) knows the difference between a memory access instruction and an I/O instruction.
- 41) The current RISC-V OTTER outputs the IO_WR signal from the main memory module. Briefly explain how you could modify the RISC-V architecture such that the IO_WR signal was output from one of the control unit modules.
- 42) Briefly describe the main purpose of the IMMED_GEN module.
- 43) Briefly describe why some of the immediate value fields have strange bit orderings.
- 44) Briefly explain why the final 32-bit immediate values associated with B-type and J-type instructions always encode the LSB as zero.
- 45) Briefly explain why the `jalr` instruction does not include the PC in the calculation while the similar `jal` instruction does.
- 46) Briefly describe the main purpose of the BRANCH_ADDR_GEN module.
- 47) In the context of branch-type instructions (conditional and unconditional), the RISC-V MCU hardware converts relative offsets encoded as part of the instructions into absolute address. Briefly describe which modules in the RISC-V OTTER MCU hardware are responsible for this conversion.
- 48) Briefly describe why we refer to the register file as a *multiport RAM*.
- 49) Briefly describe whether the register file is a synchronous or asynchronous device.
- 50) Briefly describe the main responsibilities of the ALU module.
- 51) Briefly describe whether the ALU directly provides any information regarding the status of the operations it performs.
- 52) Arithmetic shifts support the shifting of which type of number representations?
- 53) Briefly describe why shift lengths are limited to 32 bit positions in the RISC-V OTTER MCU.
- 54) Briefly describe what would happen if you attempted to shift more than 32 bit positions in the RISC-V OTTER MCU.
- 55) List the two significant differences between branch-type instructions and slt-type instructions.
- 56) Both the branch-type instructions and the slt-type instructions perform comparisons; briefly describe where those comparisons occur in the underlying RISC-V MCU hardware.
- 57) Briefly describe any limits the slt-type instructions have regarding their ability to make comparisons.
- 58) Briefly but completely describe the purpose of the MCU wrapper.
- 59) Briefly describe why the Wrapper does not register the inputs to the MCU as it does the outputs.
- 60) What limits the amount of data that can be input to the RISC-V MCU in a single operation?
- 61) Briefly describe what causes data to be stores in the registers on the wrapper level.
- 62) What three items need to happen to allow data from the RISC-V MCU to a register in the Wrapper?
- 63) Briefly describe how it is possible for the IOBUS_addr signal to be shared by both input and outputs.
- 64) Would it be possible to use the current Wrapper for another softcore MCU. Briefly but completely explain your reasoning.

- 65) Briefly but completely explain the main differences between the input and output portions of the MCU wrapper.
 - 66) The MCU can only control a fixed number of input and output bits. Describe what determines the number of input and output bits can be controlled by the MCU.
 - 67) How many total input bits can the current RISC-V OTTER MCU control? Show the calculation for this question.
 - 68) How many total output bits can the current RISC-V OTTER MCU control? Show the calculation for this question.
 - 69) If the RISC-VMCU physical memory address space was increased from 16 bit to 24 bits, what is the resulting number of bits the RISC-V MCU can both input and output?
 - 70) If the RISC-VMCU physical memory address space was increased from 16 bit to 30 bits, what is the resulting number of bits the RISC-V MCU can both input and output?
 - 71) If the RISC-VMCU physical memory address space was decreased from 16 bit to 12 bits, what is the resulting number of bits the RISC-V MCU can both input and output?
 - 72) If the RISC-VMCU physical memory address space was decreased from 16 bit to 8 bits, what is the resulting number of bits the RISC-V MCU can both input and output?
 - 73) Show a completed timing diagram based on the current RISC-V MCU wrapper that indicates exactly when the data is latched into the output registers.
-

16.15 Chapter HDL (Verilog) Exercises

- 1) Show the modifications necessary to the wrapper code so that you can add two more input devices: a set of six buttons (map them to address 0x44) and a set of seven switches (map them to address 0x77).
 - 2) Show the modifications necessary to the wrapper code so that you can add three more input devices: a set of five LEDs, another four-digit 7-segment display, and a set of 16 LEDs. You can choose any port addresses you want for this question.
-

17 RISC-V Instruction Details

17.1 Introduction

All of the previous chapters that dealt with the RISC-V MCU did so at primarily a programming level. We purposely limited our mention of hardware details in an effort to not frighten programmers who have no knowledge of the hardware implements an actual computer. This chapter starts delving into some of the hardware aspects and other details of RISC-V MCU instructions and operations. We delve into the details of the underlying hardware of the RISC-V MCU in a later chapter.

Main Chapter Topics

- **HARDWARE-BASED STACK IMPLEMENTATIONS:** This chapter describes the approach to implementing stack ADTs in digital hardware.
- **INSTRUCTION FORMATS:** This chapter describes the RISC-V instruction types and their various formats including opcodes and field codes.
- **SPECIAL INSTRUCTION HANDLING:** This chapter describes the basic operations of miscellaneous pseudoinstructions and their relation to the base instruction they translate to.

Why This Chapter is Important

This chapter is important because it describes some of the low-level details regarding RISC-V instructions and instruction execution.

17.2 Hardware-Based Stack Implementations

The stack is an abstract data type that the RISC-V MCU uses for specific types of data storage including the implementation of nested subroutines. We covered the basic functionality of a stack in Section 12.2, but that coverage intentionally did not mention stack implementation associated with hardware. This section covers the same basic concepts but from the aspect of how we typically implement stacks in hardware.

Figure 17.1 shows an example of a stack implemented in hardware, or more precisely, using a structured memory-type device such as a RAM. The example shows the highest memory locations of memory with 32k storage locations. In Figure 17.1, we use a small arrow as the stack pointer to indicate the top of the stack and listed the value in the box labeled “SP” (for “stack pointer”) below. We made this stack to be similar to the example in Section 12.2 to highlight the major implementation differences; you may want to go back and review that section, or equivalently stare out the window. Here are the descriptions of the state changes in Figure 17.1. We don’t need to state the exact width of the data for this example.

- Image 1: the stack in its empty state. The stack pointer indicates the top of the stack (the box with the letters *SP* next to it). In its initial state, the stack pointer is officially not pointing to the memory associated with the stack. We initialize the stack pointer to point outside of the actual memory.
- Image 2: the stack after one item has been pushed onto the stack. The figure uses a small arrow in addition to the stack pointer box to indicate the top of the stack.

- **Image 3:** the stack after four items (three more values since image 2) have been pushed onto the stack. The items on the stack were pushed in the following order: 34, 29, 19, and 17. Note that the stack pointer is pointing at the last thing that was pushed on the stack (17).
- **Image 4:** the stack after one item is popped off the stack. Note that the item popped off the stack is not actually removed; the item is still there but the stack pointer is adjusted to point to a new top of the stack. If we were to push another item onto the stack after this point, it would necessarily overwrite the number 17 with new data.
- **Image 5:** the stack after three items (two since image 4) are removed from the stack. Once again, we don't remove items from the stack; we simply adjust the stack pointer.

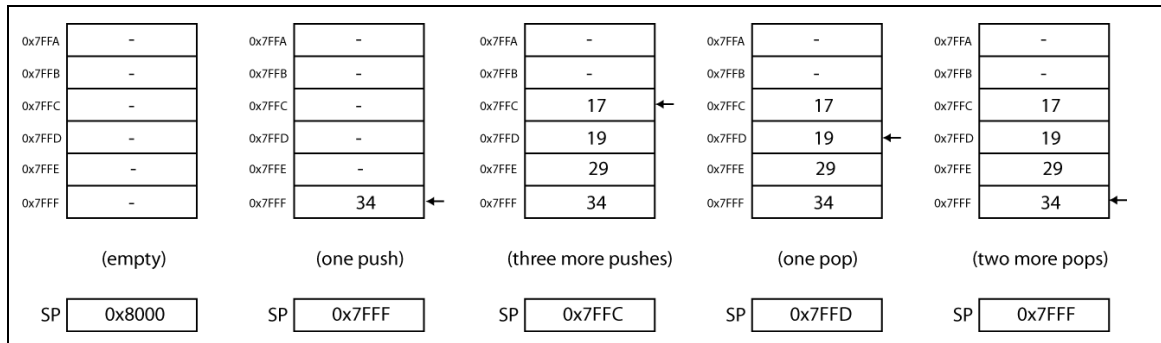


Figure 17.1: Implementation of a hardware-based (structured memory) stack.

One final but important characteristic exhibited by the stack in Figure 17.1 is that we consider the stack to “grow” in the negative direction as we push items onto it. In other words, when we push data onto the stack, the stack pointer (which is an address value pointing at the stack area in main memory) value becomes smaller. Conversely, when we remove items from the stack, the stack pointer increases in magnitude. There is no particular reason why most MCUs do it this way other than tradition; we could implement stacks the other way just as easily.

17.3 Instruction Types and Formats

The best place to start any discussion on RISC-V hardware is with low-level descriptions of the instructions themselves. Recall that an assembly language is a set of mnemonics that represent instructions. We humans use the instructions to control the underlying hardware. That being the case, the instructions are nothing more than a set of bits that the hardware uses to know what instruction needs executing and how exactly to execute that instruction.

There are about approximately 40 base instructions and another 20 or so pseudoinstructions in the RISC-V MCU instruction set. While this seems like many instructions, it's not as bad as it initially seems. First, we're only required to understand the ins and outs of the base instructions because the assembler translates the pseudoinstructions. Additionally, we use various approaches to exploit the many similarities between the instructions to further expedite our understanding of both the instruction set and how we implement those instructions in the underlying hardware.

17.3.1 Field Codes and Opcodes

We divide all of the 40 base instructions into six different types. The thing that differentiates these instruction types is how they arrange the underlying bits. Because the instructions have different numbers and types of operands, it makes intuitive sense that the underlying bits that form those instructions will also be different. Table 17.1 shows the six RISC-V formats. We'll look more closely at these formats when we look into individual instructions, but there are a few general items to notice about Table 17.1.

- Although each instruction comprises of 32 bit, we arrange those bits in an organized manner by dividing them into groups according to their purpose. There are two types of groups, which we refer to as “field codes” and “operational codes” (or “opcodes”). These two different groups have

different purposes. The opcodes define the particular instruction being executed and cannot vary for a given instruction. The field codes are variable for a given instruction. The field codes define either register values or immediate values. While the field codes defining registers are always 5-bits wide, the immediate values vary depending on the instruction.

- We designate the opcodes in Table 17.1 using shading.
- The field codes and opcodes have both common widths and common locations in the instruction across the set of instruction types. Organizing the field codes in this way makes the underlying RISC-V hardware less complex.
- The numbering on the immediate value field codes can sometimes be strange. Once again, the RISC-V designers chose this organization to simplify actual RISC-V MCU implementations.
- We designate the register-based field codes in Table 17.1 by either an “**rd**”, “**rs1**”, or “**rs2**”, where **rd** is the destination register and **rs1** & **rs2** are the source registers. The number of register-based operands depends on the instruction type. We give the register-based field codes common names.

Instr Type	Instruction Format
R-type	
I-type	
S-type	
B-type	
U-type	
J-type	

Table 17.1: RISC-V Instruction types and associated formats.

17.4 Notable Handling of Specific Instructions

Some the RISC-V instructions are notable because of the way programs use them and because of their somewhat unique hardware implementations. The “uniqueness” of these instructions primarily refers to their non-intuitive usage and “lack” of direct use. Programmers use these instructions quite often, but in an indirect manner as these instructions are primarily what the assembler uses to implement various and common pseudoinstructions.

17.4.1 Add Upper Immediate to PC Instruction: **auipc**

The RISC-V ISA includes several pseudoinstructions involved in program flow control. These instructions then necessarily involve using the PC. The primary purpose of the **auipc** instruction is to load a copy of the current program counter to a register, where then other instructions can use that value. The primary use of the **auipc** instruction is as part of the **call** pseudoinstruction (the other part of the **call** pseudoinstruction is a **jalc** instruction) and the **la** pseudoinstruction (the other part being an **lw** instruction).

The **auipc** instruction loads the sum of the current PC and a modified immediate value into the destination register. The instruction sign-extends the immediate value and shifts it left by 12-bit locations before being added to the PC value. Table 17.2 provides an RTL description of the **auipc** instruction while Table 17.3 shows lower-level implementation details. The efficacy of this instruction relates to its usage as part the **call** pseudoinstruction, so we save insights into the operation of the **auipc** instruction when we describe the **call** pseudoinstruction in Section 17.4.3 and the **la** pseudoinstruction in 17.4.6.

The RTL for the **auipc** instruction in Table 17.2 is misleading. The instruction essentially creates a 32-bit value from an immediate value and the PC. The instruction makes that value by placing the 20-bit immediate value in the 20 left-most bits of the register, and then adding the PC value. The notion of the shifting operation implies that the lower 12-bits are zero before the addition operation. The underlying RISC-V hardware does more of a reassignment of 20-bit immediate value for the lower 20 bits to the upper 20 bits; no shifting occurs. It is the programmer's responsibility to handle overflows of the addition operation. If the immediate value associated with the instruction is zero, the instruction effectively moves the PC to the destination register, which is actually one way we commonly use the instruction.

Instr Type	Instruction Form	Instruction RTL	Example Usage
U-Type	auipc rd,imm	$rd \leftarrow PC + (\text{sext}(\text{imm}) \ll 12)$	auipc x8,imm

Table 17.2: Usage and description of the **auipc** instruction.

Type	Instruction Type Format
Instr	Instruction Format
U-type	
auipc	

Table 17.3: Type and Instruction format for the **auipc** instruction.

17.4.2 Load Upper Immediate Instruction: **lui**

The **lui** instruction is similar to the **auipc** instruction. It's once again one of those instructions that programmers don't use often in a direct manner, but use often in an indirect manner as a part of useful pseudoinstructions. The assembler translates the **li** pseudoinstruction into **lui** instruction.

The **lui** instruction loads a modified immediate value into the destination register. The instruction sign-extends the immediate value and shifts it left 12-bit locations before loading it to the destination register. Table 17.4 provides an RTL description of the **lui** instruction while Table 17.5 shows lower-level implementation details. The efficacy of this instruction is related to its usage as part the **li** pseudoinstruction, so we save insights into the operation of the **lui** instruction for Section 17.4.5, where we describe the **li** pseudoinstruction.

The RTL for the **lui** instruction in Table 17.2 is misleading similar to the way the **auipc** instruction is misleading. The instruction essentially creates a 32-bit value from an immediate value by placing the 20-bit immediate value in the 20 left-most bits of the register. The notion of the shifting operation implies that the lower 12-bits in the destination register are always zero. The underlying RISC-V hardware does more of a reassignment of immediate value; no shifting occurs. If the immediate value associated with the instruction is zero, the instruction effect is to clear the destination register.

Instr Type	Instruction Form	Instruction RTL	Example Usage
U-Type	<code>lui rd,imm</code>	$rd \leftarrow \text{sext}(imm) \ll 12$	<code>lui x8,imm</code>

Table 17.4: Usage and description of the `lui` instruction.

Type	Instruction Type Format
Instr	Instruction Format
U-type	
<code>lui</code>	

Table 17.5: Type and instruction format for the `lui` instruction.

17.4.3 Calling Subroutines: The `call` Pseudoinstruction

Programmers use the `call` pseudoinstruction to transfer program flow control to another area of the program we refer to as a subroutine. The basic operation of a subroutine call is to load the address of the first instruction in the subroutine into the PC, but at the same time, storing the address of the instruction following the `call` instruction. We refer to the address of the instruction following the `call` instruction as the return address, as that is where program control transfers to after completing execution of the instructions in the subroutine. The subroutine code uses a label to mark the address of the first instruction in the subroutine.

The `call` instruction has two primary responsibilities. First, it must formulate the absolute address of the subroutine from the label value. Absolute address formation is another exercise in using the RISC-V instructions and assembler to convert the relative addresses encoded in the instructions¹ into a 32-bit address that the hardware loads into the program counter as part of the translated call instruction sequence. Second, the `call` instruction must store the return address in a register, which the return from subroutine instruction (`ret`) later uses to transfer program flow control back to the instruction following the `call` instruction listed in the program code.

Table 17.6 shows an overview of the `call` pseudoinstruction including the associated RTL statements. The RTL statement in Table 17.6 is slightly misleading in that the information makes it seem like there is only one instruction required to execute the `call` pseudoinstruction. As you'll see next, this is not the case.

Instruction Form	Instruction RTL	Example Usage	Comment
<code>call label</code>	$rd \leftarrow PC + 8$ $PC \leftarrow \&label$	<code>call my_sub</code>	Transfers program control to instruction associated with <code>my_sub</code> label.

Table 17.6: The overview of the `call` pseudoinstruction.

Table 17.7 shows the `call` instruction in two forms and includes some other usage information. Table 17.7 shows that the assembler translates the `call` pseudoinstruction into an `auipc & jalr` instruction. We're primarily interested in the form of the instruction in the second row, which is the one we primarily use when programming. The `call` pseudoinstruction form in the second row is a special case of the `call` form in the first row where the destination register defaults to `x1`.

¹ `call` is a pseudoinstruction, but we commonly reference it as an instruction to save typing keystrokes.

Instruction Form	Equivalent Base Instruction(s)	Example Usage	Comment
<code>call rd, lab</code>	<code>auipc rd, hi(lab)</code> <code>jalr rd, lo(lab)</code>	<code>call x5, subrut</code>	Jump to instruction associated with label; Store current address in rd
<code>call lab</code>	<code>auipc x1, hi(lab)</code> <code>jalr x1, lo(x1)</code>	<code>call subrut</code>	Jump to instruction associated with label; Store current address in x1

Table 17.7: The two forms of the call pseudoinstruction and equivalent base instructions.

Table 17.8 provides all the gory details as to how the `auipc` & `jalr` instructions work together to execute a `call` pseudoinstruction. Note that the following description is for the `call` form in the second row of Table 17.7. Here is a full description of the code in Table 17.8:

- The `call` pseudoinstruction has an associated label: `Instr`. The comment above the `call` pseudoinstruction states that the value of this label is `0xFEBA`.
- The assembler translates the `call` pseudoinstruction into the `auipc` and `jalr` instructions in the right column of Table 17.8. These two instructions perform three distinct functions (one for `auipc` and two for `jalr`):
 - 1) The `auipc` instruction stores the upper 20 bits of the address of the subroutine in the upper 20 bits of register `x6`. The current PC value is `0x0000FEBA`, which is the address of the `call` instruction in the left column. The value stored in `x6` after this instruction is `0x0000F000`.
 - 2) The `jalr` instruction stores the current PC value plus four (`PC+4`) in `x1`, which is the address of the “instruction after the `call` pseudoinstruction”. But since the `call` pseudoinstruction generates two base instructions, the address loaded into `x1` is the address of the instruction after the `jalr` instruction, which is effectively the address 8 greater than the address of the `call` pseudoinstruction as it appears in the source code. This is why the RTL statement in Table 17.6 is misleading. The return address is officially the instruction after the `jalr` instruction in the translated code.
 - 3) The `jalr` instruction completes the subroutine address formulation started in the `auipc` instruction by adding the value in `x6` to the lower 12 bits of the address of the first instruction of the subroutine (thus creating an absolute address) and storing that value in the PC.
- Lucky for us programmer types that the assembler handles most of the ugly details. Namely, the assembler generates and assigns the right-most operands of both the `auipc` & `jalr` instructions.

Pseudoinstruction Usage	Pseudoinstruction Translation
<code>#Instr=0x0000FEBA</code>	
<code>Instr: call my_sub</code>	<code>inst1: auipc x6, 0x0000F # x6 = 0x0000F000</code>
<code> nop</code>	<code>inst2: jalr x1, x6, 0xEBA # PC <- (x6 + 0xEBA)</code>

Table 17.8: Example of the call pseudoinstruction translation.

17.4.3.1 Subroutine Call Timing

Calling subroutines is another common type of program flow control operation. There is no dedicated calling instruction in the RISC-V architecture, but there is a `call` pseudoinstruction that programmers can use. The `call` pseudoinstruction translates to two base instructions: `auipc` & `jalr`. Section 17.4.3 described the

operation of the **call** pseudoinstruction and underlying base instruction; this section provides an example timing diagram with verbose description. Figure 17.2 shows an example timing diagram associated with a **call** pseudoinstruction; here are the gory details:

- This example shows the calling of a subroutine names “sub_rut”; the address of the first instruction in this subroutine is 0x4328, which arbitrary. The address of the **call** instruction is 0x90. The assembler translates the call instruction to an **auipc** instruction (address 0x90) followed by a **jalr** instruction (0x94).
- The PCWrite signal asserts at the beginning of the execute cycle, which causes the PC to advance to the next instruction as the diagram indicates with the (1) note. The actual instruction at 0x8C does not matter but it is not a program flow control-type instruction.
- The note at (6) indicates the machine code associated with the **auipc** instruction become available starting at the execute cycle for the **auipc** instruction because part of the fetch cycle includes asserting the read enable for instruction memory.
- The **auipc** instruction loads the upper 20 bits of the address of the subroutine to the upper 20 bits of x6 as note (2) indicates. The **auipc** instruction essentially grabs the 20 MSBs of the subroutine address. The value of x6 does not matter before note (2), but it is loaded with the five most significant nibbles of the subroutine address as the control unit transitions from a execute to fetch cycle.
- The assembler generates the **jalr** instruction shown at the top of the diagram. The left-most operand indicates the register where the MCU stores the return address. The hardware uses the values associated with the other two operands to complete the formation of the address of the first instruction in the subroutine. The “0x294” value in the **jalr** instruction is the lower 12-bits of the relative offset from the address of the current instruction to the address of the subroutine: (0x00004328 – 0x00000094). Yes, this seems like a lot of trouble, but the assembler handles all the math for you.
- The **jalr** instruction has two responsibilities as the notes at (3) & (4) show. The note at (3) indicates the address of the instruction after the **call** pseudoinstruction is stored in x1. Recall that x1 is the designated register for holding return addresses. The value in x1 is the address of the first instruction that will execute after returning from the subroutine. The documentation for the **call** instruction indicates that the MCU stores “PC+8” in x1, but this is because the assembler translates the **call** instruction into two base instructions, which is eight greater the address of the **call** pseudoinstruction as it appears in the assembly language source code. The note at (4) indicates the new PC address is the address of the first instruction of the subroutine, which indicates program control has transferred to the subroutine.
- The instruction bits for the first instruction in the subroutine become available during the execute cycle as the note at (5) indicates.

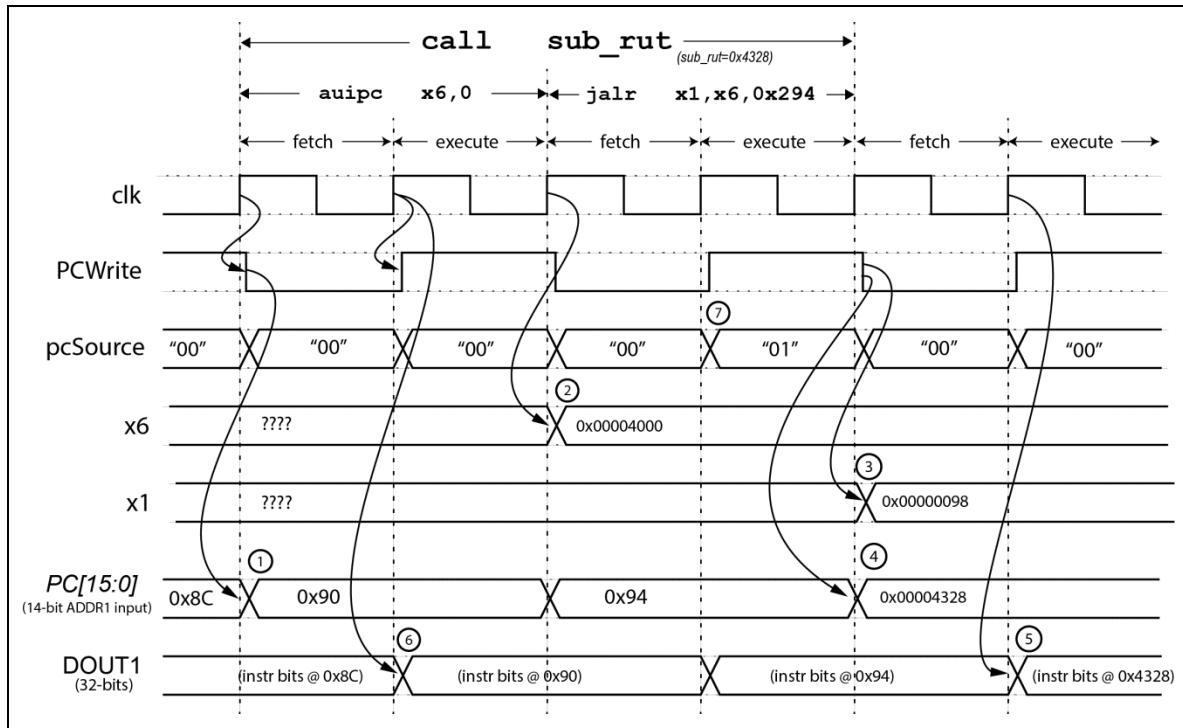


Figure 17.2: Example timing diagram for the `call` pseudoinstruction.

17.4.4 Returning from Subroutines: The `ret` Pseudoinstruction

The `ret` pseudoinstruction is another program flow control operation, typically associated with a `call` pseudoinstruction. Recall that the assembler translates the `call` pseudoinstruction into an `auipc` and `jalr` instruction. While the assembler could have used a `jal` instruction as part of the `call` translation, it uses a `jalr`. The same is not true for the returning from subroutines. When returning from subroutines, the assembler must use a `jalr` instruction because the return address is stored in a register. The main difference between the `jal` & `jalr` subroutine is in the fact that the absolute address calculation to determine the value loaded into the PC includes a register. When programmers use the `ret` pseudoinstruction, the register used in the calculation defaults to `x1`. Table 17.9 shows all the important information for the `ret` pseudoinstruction.

Instruction Form	Instruction RTL	Example Usage	Equivalent Base Instruction	Comment
<code>ret</code>	$PC \leftarrow x1$	<code>ret</code>	<code>jalr x0, 0 (x1)</code>	Transfers program control to address in <code>x1</code>

Table 17.9: The overview of the `ret` pseudoinstruction.

17.4.4.1 Subroutine Return Timing

We can best describe the operation of the `ret` pseudoinstruction by using a timing diagram. Figure 17.3 shows a timing diagram that uses the `ret` pseudoinstruction; this diagram is somewhat special because it works with the `call` instruction timing diagram in Figure 17.2. Here are the nitty-gritty details of the `ret` of the timing diagram in Figure 17.3:

- The assembler translates the `ret` pseudoinstruction to a `jalr` instruction, which Figure 17.2 indicates in the top of the diagram directly above the `jalr` instruction. The only responsibility of the `jalr` instruction is to load the value in the return address, `x1` or `ra`, into the PC. Note that the

jalr instruction uses `x1` as a base address and then includes a zero offset. The assembler does with translation for us programmers.

- Register `x1` contains the return address from Figure 17.3. This was the address two instructions after the **call** pseudoinstruction because the assembler translated the **call** pseudoinstruction to two base instructions. The address in `x1` is effectively the instruction two instructions locations after the **call** pseudoinstruction.
- Note (1) shows normal sequential instruction execution with the addition of 4 to the current PC. The value of `0x4400` is arbitrary. Note (2) shows when the instruction bits associated with the instruction become available at the start of the execute cycle.
- The **jalr** instruction must select the absolute address associated with **jalr** input to the PC. Note (3) shows that the `pcSource` changes to “01” as a result of the MCU decoding the bits associated with the **jalr** instruction after the start of the execute cycle.
- The control unit in the RISC-V hardware decodes the instruction and sends out the appropriate control signals; two of those signals are `PCWrite` and `pcSource`. These two signals direct the MCU to copy the address in `x1` into the PC, as the note at (4) shows. The value in `x1` was loaded there by the **jalr** instruction associated with the **call** pseudoinstruction.
- Note (5) shows that the instruction bits associated with the instruction at the address in the PC become available after entering the execute cycle.

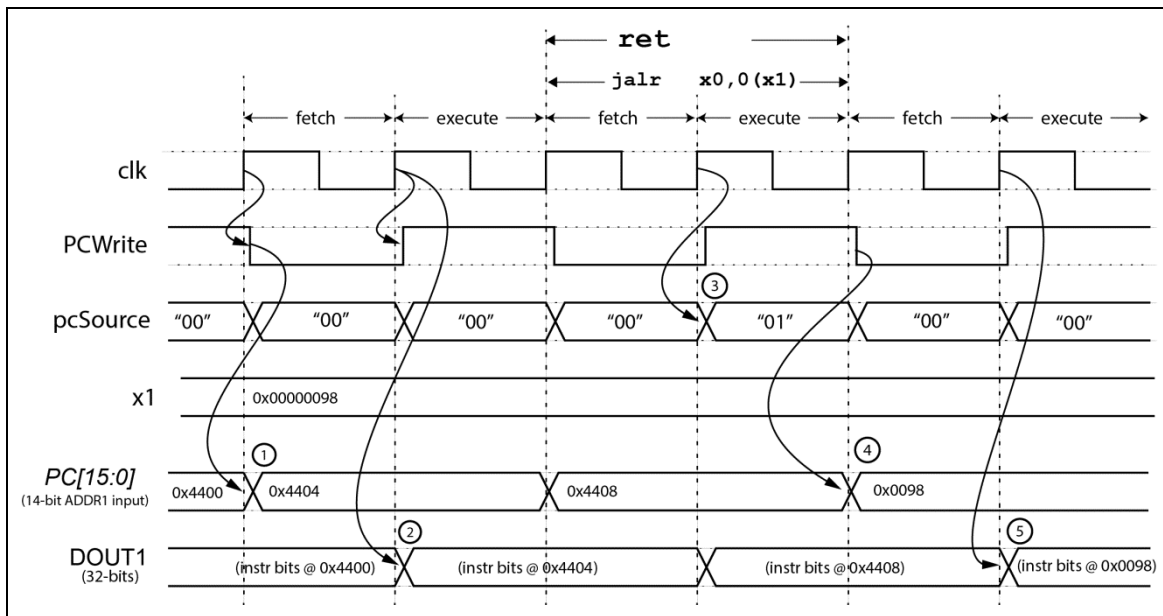


Figure 17.3: Example timing diagram for the **ret** pseudoinstruction.

17.4.5 Loading Immediate Value: **li**

The **li** pseudoinstruction effectively loads an immediate value into a register. RISC-V MCU registers are 32-bit wide, but the various instruction forms represent immediate values using significantly less bits. The assembler translates the **li** pseudoinstruction into an **addi** instruction if the assembler can represent the immediate value using the 12-bit immediate field in the **addi** instruction. If the assembler can't represent the immediate value with a 12-bit value, the assembler translates the **li** pseudoinstruction to an **lui** base instruction followed by an **addi** instruction. Table 17.10 shows an overview of the **li** instruction.

Instruction Form	Instruction RTL	Example Usage	Comment
li rd,imm	$rd \leftarrow imm$	li x8,20	Immediate value loaded into destination register

Table 17.10: The overview of the **li pseudoinstruction.**

Although the assembler handles the details of whether the **li** instruction translates to one or two base instructions, it's instructive to know the mechanics of the assembler's translations. Table 17.11 shows an example of two **li** pseudoinstructions in the left column; the right column shows how the assembler converts those instructions to base instructions based on the magnitude of the immediate operand in the **li** pseudoinstruction. Here are some extra items worthy of noting about Table 17.11:

- For the first row in Table 17.11, the associated immediate value can fit into a 12-bit signed value range [-2048,2047] (the 12-bit range threshold is associated with the **addi** instruction). Because the immediate value can fit into a 12-bit range, the assembler translates the **li** pseudoinstruction into a single **addi** instruction.
- For the second row in Table 17.11, the associated immediate value is does not fit into a 12-bit signed range. This larger immediate value thus causes the assembler to translate the **li** pseudoinstruction into two base instructions: **lui** & **addi** as follows:
 - 1) The **lui** instruction encodes the upper 20 bits of the immediate value into the upper 20 bits of the destination register x8, and clears the lower 12 bits of x8.
 - 2) The **addi** instruction places the lower 12 bits that were not included from the **lui** instruction into the lower 12 bits of the x8, which was also the destination register in the **lui** instruction.

Pseudoinstruction Usage	Pseudoinstruction Translation
li x8,19	addi x8,x8,19 # x8=19
li x8,0x12345678	inst1: lui x8,0x12345678 # x8=0x12345000 inst2: addi x8,x8,678 # x8=0x12345678

Table 17.11: Example of the **li pseudoinstruction translation.**

17.4.6 Load Address Pseudoinstruction: **la**

The **la** is a pseudoinstruction used to translate program labels into to numerical values and store those values in registers. Programmers typically use labels as symbolic addresses, which allow jump and branch instructions to transfer program control to instructions associated with label. This symbolic representation of addresses allows for 1) easy program modifications, and 2) absolves programmers from dealing with relative and absolute address calculations. One common use of labels to locate the base addresses of look-up-tables (LUTs). Table 17.12 shows an overview of the **la** instruction usage. The RTL in Table 17.12 uses the "&" symbol, which is the "address of" operator in the C programming language. In this context, it refers to the fact that labels in RISC-V programs are address values of data (instruction or actual data) in memory.

Instruction Form	Instruction RTL	Example Usage	Comment
la rd, label	rd ← &label	la x8, my_label	Numerical value of my_label copied to x8

Table 17.12: The overview of the **la pseudoinstruction.**

The assembler translates the **la** pseudoinstruction into an **auipc** base instruction followed by an **addi** base instruction. The **la** pseudoinstruction uses a two-step process to obtain the label value:

- 1) The **auipc** base instruction places the value of the current program counter into a register.
- 2) The **addi** base instruction adds an offset to the register value loaded by the **auipc** instruction. The offset is a relative value, specifically the relative offset from the address of the **auipc** instruction to the value (data or instruction) associated with the label in question.

Table 17.13 shows an example of the **la** instruction with comments. Here are a few items to note about the **la** pseudoinstruction.

- The **auipc** instruction is only interested in the PC, which is why it the assembler places a zero in the immediate operand in the instruction.
- The **addi** instruction adds a relative offset to the PC valued stored in the register. The assembler makes the calculation and assignment of the result into x10. The assembler calculates and assigns the immediate value to the immediate operand in the **addi** instruction, which is: (address of **auipc** instruction) – (value of label).

Pseudoinstruction Usage	Pseudoinstruction Translation
My_lab: nop	
la x10, My_lab	auipc x10, 0 # x10 ← PC + 0
	addi x10, x10, -4 # x10 ← x10 - 4

Table 17.13: Example of the **la pseudoinstruction translation.**

17.4.6.1 Assembler Handling of Labels

The assembler assigns a value to every label in a program; this value is effectively an absolute memory address. The memory stores two types of data: instructions and data. This means that a label can either be the address of a specific instruction in program memory (code segment or text segment) or the addresses of a data (data or stack segments).

The value that the assembler assigns to labels is effectively the output of a counter used in the assembler in the assembly process. When the program encounters data (in the data segment) or instructions (in the code segment), this internal counter advances. For example, each byte of data in the data segment advances the counter by one; each halfword and word of data advances the counter by two and four, respectively. When the assembler encounters a label in the code, the assembler assigns the current value of that counter to that label. When the assembler encounters multiple labels without encountering data, all the labels receive the same value. Programs specify data (assigned values) or space (unassigned values) in the data segment using various assembler directives.

While specifying data can advance the internal assembler counter by one, two, or four, instructions always advance the counter by four. Labels in the code segment are treated exactly the same as labels in the data segment, including the ability to have multiple labels (one per line though) without encountering an instruction.

The assembler translates the **la** pseudoinstruction into two base instructions: **auipc** & **addi**. The good part about this is that the assembler does the work for the programmer, but it's a good to know exactly how the assembler does this. The code fragment in Table 17.14 provides an example to explain this mechanism. Here is the full explanations including the gory details:

- The program has the “dog” label associated with the instruction on line (02). The assembler assigns the “dog” label the value of the internal counter, which is 0x00000080; this value is the address in the code segment where the nop instruction on line (02) is stored.
- Two more generic instruction happen in the program on lines (02-03); nothing big here.
- The assembler encounters the **la** instruction on line (06), which copies the value associated with the “dog” label into x10. The assembler does this by translating the **la** instruction into an **auipc** instruction followed by an **addi** instruction; the approach here is to use existing base instructions to load the address of the instruction associated with the “dog” label into a register. Once that value is in the register, programs can use this value for other very useful purposes.
- The **auipc** instruction used in this context effectively loads the current PC value to a register. It does this by encoding a zero into the immediate value associate with the **auipc** instruction. The PC value that is encoded is the value of the **la** instruction as it appears in the program, which is the same address as the **auipc** instruction after the assembler translates the instruction.
- The assembler then needs to “adjust” the current value in x10 to be the address of the instruction associated with the dog label, which are three instructions before the **la** instruction in the program. The assembler makes this adjustment by adding a relative offset from the address of the **la** instruction (or the **auipc** instruction) to the address of the instruction associated with the **dog** label: the relative offset value is -12. It's a negative number because it's going backwards in the code (instruction addresses are becoming smaller). It's 12 because there are three instructions where each instruction is four bytes wide. It adds this offset value to the value in x10 using the **addi** instruction. What the **addi** instruction effectively does is creates an absolute address from the base PC address and the relative offset. I'm sure glad the assembler takes care of these details for me; very clever, but I have better things to do.

```
(00) #~~~~~ code fragment ~~~~~
(01) .text
(02) dog:  nop          # generic instruction: addr=0x00000080
(03)     nop          # placeholder instructions
(04)     nop
(05)
(06)     la    x10,dog  # place associated value of dog (0x00000080) into x10
(07)
(08) #----- the assembler translates the la instruction to the following: -----
(09) #
(10) #     auipc x10,0    # zero is the immediate value
(11) #
(12) #     addi  x10,-12  # -12 is the relative offset from the la instruction
(13) #                   # to the instruction associated with the dog label
(14) #
(15) #~~~~~ code fragment ~~~~~
```

Table 17.14: Code fragment example using the **la instruction.**

17.4.7 Special Operations: the **slt**-Type Instructions

The RISC-V ISA uses bases branching operations on the result of the comparison of the value in two registers. The RISC-V hardware makes the branching decision based on the values of the branch instruction operands and directs the loading of the appropriate value to the PC. The RISC-V ISA uses the set-if-less-than-type instructions to make comparisons between two registers or a register and an immediate value and store the result of the comparison in a register without the option of making a branch or not.

Table 10.9 provides an overview of the set if less than (**slt**) instructions. There are two main types of **slt**-type instructions. All **slt**-type instructions set the destination register (writes ‘1’ to the register) if the result of the

comparison is true; the differences lie in the comparisons made by the instructions. The immediate forms of the instructions (**slti** & **sltiu**) compare a register to an immediate value, while the register-immediate forms of the instructions compare two register values. Additionally, the instructions either interpret the two operands differently, as both unsigned values (**sltu** & **sltiu**) or signed values (**slt** & **slti**).

Table 10.9 uses some special vernacular in the associated RTL to describe the instructions. First, it uses “<_u” and “<_s” for unsigned and signed comparisons, respectively. Second, it uses a C programming language type operator to describe the result of the comparison. The “?:” is an arithmetic if operator. This operator includes an expression on the left of the question mark, and a value on each side of the colon; the RTL statements in Table 10.9 use a comparison in place of the expression. If the comparison evaluates to true, the operator assigns the value on the left side of the colon (‘1’) to the destination register; otherwise, the operation assigns the value on the right side of the operator (‘0’). Thus, the destination register is either set or cleared as a result of executing any one of these slt-type instructions.

The immediate forms of the slt-type instructions represent the immediate operand in a 12-bit field in the instruction format. The hardware interprets these values as signed values, which gives an effective range of [-2048,2047]. The RISC-V MCU hardware sign-extends these values prior to the comparison. Table 10.9 indicates sign-extension of the immediate value with using the “sext(imm)” notation in the RTL statement.

Instr Type	Instruction Form	Instruction RTL	Example Usage	Comment
R-Type	slt rd,rs1,rs2	$rd \leftarrow (rs1 <_s rs2) ? 1 : 0$	slt x10,x5,x21	signed compare
I-Type	slti rd,rs1,imm	$rd \leftarrow (rs1 <_s sext(imm)) ? 1 : 0$	slti x8,x9,0xF0	signed compare 12-bit signed imm
R-Type	sltu rd,rs1,rs2	$rd \leftarrow (rs1 <_u rs2) ? 1 : 0$	sltu x5,x6,x16	unsigned compare
I-Type	sltiu rd,rs1,imm	$rd \leftarrow (rs1 <_u sext(imm)) ? 1 : 0$	sltiu x7,x8,25	unsigned compare 12-bit signed imm

Table 17.15: The two forms associated with the four logic instructions.

17.5 Chapter Summary

- The stack is an abstract data type that the RISC-V MCU uses to store data. Stacks are a common module in most computer architectures and are typically implemented using a structured memory device such as a RAM. There are many approaches to implementing stacks, but the most efficient approach in digital hardware is a structured memory device.
 - The **auipc** and **lui** instructions are similar in that they transfer an immediate value to a register. These instructions are not typically used directly in programs; they instead are part of pseudoinstructions such as **li** and **la**. The main difference between the **auipc** and **lui** instructions is that the PC value modifies the result for the **auipc** instruction.
 - The **li** pseudoinstruction places an immediate operand into a register. The **li** pseudoinstruction translates to one or two base instructions depending on the sign and magnitude of the immediate value.
 - The **la** pseudoinstruction places the value associated with a program label into a register. Programmers typically don't keep track of the various addresses of items in programs; they instead use the **la** pseudoinstruction to work with only the addresses they require without ever actually knowing that address.
 - The slt-type instruction are similar to branch instructions in that they perform a compare operation, but differ because, unlike branch instructions, the result of a slt-type instruction can never be a branch.
-

17.6 Chapter Exercises

1. Briefly describe what exactly the stack pointer is and what it does in terms of hardware-based stack implementations.
2. Briefly describing the notion of stacks growing and shrinking in terms of digital hardware.
3. Briefly describe the notion of stack overflow and underflow in terms of digital hardware.
4. Briefly describe what occurs if the stack pointer value exceeds its designated bit width.
5. Briefly describe what determines how many stacks a RISC-V based MCU can “easily” have.
6. Briefly describe the drawbacks of having a high number of easily implemented stacks using the RISC-V architecture.
7. Briefly describe why stacks implemented in hardware do not alter data that is popped off the stack.
8. Briefly describe why the hardware has no need to implement pseudoinstructions.
9. Briefly describe who or what decides how many instruction types a given computer ISA has.
10. List the six types of RISC-V instructions.
11. Briefly explain whether pseudoinstructions fall into the category of the six different types of RISC-V instructions.
12. Briefly explain the main difference between a field code and an opcode.
13. Briefly describe who or what determines the field code and opcode configuration for a given instruction set architecture.
14. Briefly describe what differentiates the six different RISC-V instruction types.
15. Briefly describe why there is significant overlap between with the field codes and opcodes in the RISC-V instruction types.
16. Briefly describe why the immediate-type field codes sometimes have really wanky arrangements.
17. Briefly describe how it is possible that you the programmer can write programs until the cows come how but never have a **lui** or **auipc** instruction in your source code.
18. Briefly describe if the **auipc** and **lui** instructions actually do any shift operations as their RTL descriptions seem to indicate.
19. What are the two primary responsibilities of the **call** pseudoinstruction?
20. Briefly describe why the **call** pseudoinstruction adds 8 to the PC rather than 4, where 4 seems like it would be the addresses of the instruction following the **call** pseudoinstruction.
21. Briefly describe the responsibilities of the two base instructions associated with a **call** pseudoinstruction.
22. Briefly describe why the **call** pseudoinstruction translated to two base instruction while the **ret** pseudoinstruction translates to only one base instruction.
23. Briefly describe what determines whether the **li** pseudoinstruction translates into one or two base instructions.
24. Briefly describe from an efficiency standpoint whether it is better to use a **mv** pseudoinstruction or a **li** pseudoinstruction to clear a register.
25. Briefly describe why there is not one **li**-type pseudoinstruction or instruction that loads a full 32-bits at one time.
26. Briefly describe when the **la** pseudoinstruction can find the addresses of data in both program and data memory.

27. Briefly describe whether it is possible to branch to an address in data memory.
 28. Briefly describe how branch instructions and set-if-less-than-type instructions are similar.
 29. Briefly describe the primary difference between branch instructions and set-if-less-than-type instructions.
-

18 RISC-V MCU Interrupt Architecture (Hardware)

18.1 Introduction

We previously discussed the notion of interrupts and real-time programming in Chapter 13. While that chapter was designed such that pure programmers could understand it, the discussion in this chapter is for people who have a solid understanding of standard digital design topics. We focus the discussion in this chapter on the low-level hardware implementation details of the RISC-V OTTER MCU interrupt architecture and assume our gentle readers as well-versed with the theory of interrupts as they relate to MCUs. This chapter is somewhat stand-alone, but the best approach would be to become familiar with the topics in Chapter 13 before reading this chapter.

The notion of interrupts forms the heart of most embedded systems. The interrupt mechanism essentially provides hardware with the ability to “call” subroutines, which means that devices outside of the MCU can affect program operation. The main purpose of interrupts is to reduce the response time to external inputs, which allows the MCU to avoid other less efficient approaches to I/O, which thus allows makes the architecture into a real-time system. There are many other advantages to using interrupts that we cover in both Chapter 13 and this chapter.

Main Chapter Topics

- **HARDWARE DETAILS OF INSTRUCTION EXECUTION:** This chapter provides pertinent hardware details regarding the RISC-V MCU interrupt architecture.
- **INTERRUPT SUPPORT HARDWARE:** This chapter describes the structure in the RISC-V hardware that directly supports the interrupt architecture.
- **PROGRAM FLOW CONTROL:** This chapter describes interrupts in the context of program flow control, which is an inherent quality of interrupts.
- **INTERRUPT TIMING ISSUES:** This chapter describes the various timing issues involved with interrupts.
- **INTERRUPT INTERFACING ISSUES :** This chapter describes common interrupt interfacing issues such as interrupt signal noise and signal duration requirements.

Why This Chapter is Important

This chapter is important because it describes the low-level architecture details of the RISC-V MCU interrupt architecture.

18.2 RISC-V MCU Interrupt Overview

The concept of interrupts is relatively simple due to their similarity to subroutines. You can generally connect external peripheral devices to an MCU in such a way as they can do what we refer to as “generate an interrupt”. While this may sound complicated, it simply means that the device has the ability to assert a signal; this signal is understood to be connected to the interrupt input on the MCU. When an external peripheral device “generates an interrupt”, the microcontroller stops what it is doing and starts processing a special subroutine known as the *interrupt service routine*, or ISR. When processing of the ISR is complete, the microcontroller resumes to its regularly scheduled programming. Simply stated, the ISR is nothing more than a subroutine initiated by hardware. The section covers the details of how the hardware processes the interrupts.

Because interrupts are similar to subroutines, the hardware portions of the RISC-V MCU's interrupt architecture is minimal. Because the hardware already supports subroutines, most of the required hardware for interrupt processing is already in place. The hardware we describe in this chapter supports the interrupt cycle and interrupt-related instructions.

18.2.1 The RISC-V Interrupt Input

The RISC-V MCU has an input that allows external devices to indicate to the MCU that they require some type of attention. When the signal on this input is asserted, the MCU goes through a specific process in order to give the external device the attention it's requesting. Figure 18.1 shows the black box diagram for the RISC-V MCU; the INTR signal is the interrupt input. The current RISC-V OTTER MCU contains only one interrupt; if there were two interrupts, there would most likely be two different interrupt-type inputs to the RISC-V MCU.

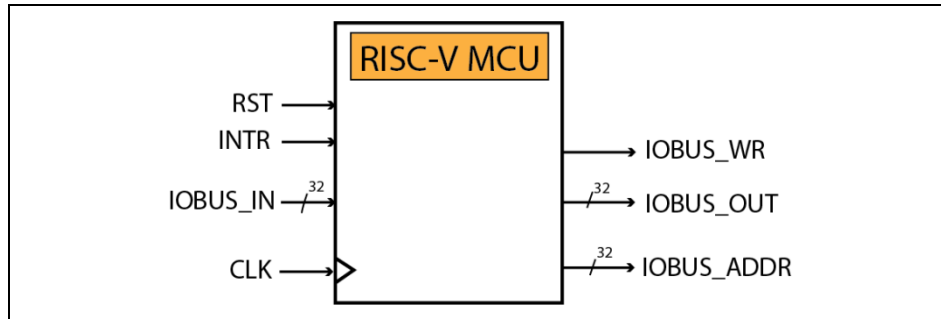


Figure 18.1: Black box diagram for the RISC-V MCU.

18.2.2 The Interrupt Cycle

The MCU responds to an asserted signal on the INTR input by entering an interrupt cycle. As with other cycles associated with the MCU (fetch, execute, and writeback), the interrupt cycle is associated with a given number of states in the FSM that controls the RISC-V MCU. A MCU's particular interrupt architecture determines the number of states in the interrupt cycle. The interrupt cycle is thus a blanket term for all the “special” events that must happen when the MCU responds to an external interrupt signal. In other words, the interrupt cycle does the special operations to support the firmware entering the interrupt service routine; instruction execution controls the exits from ISRs (no special states required).

Figure 18.2 shows the state diagram for the RISC-V FSM including the interrupt cycle. This is essentially the FSM in Figure 16.3 but now with support for interrupts. We've opted to describe the RISC-V OTTER MCU state diagram in two different ways: first without interrupts, and then with interrupts. The approach underscores the notion that interrupts are a *feature* in the MCU; our intention was not to overload you with information by presenting interrupt topics simultaneously with basic computer architecture and programming concepts. The following items list the high-level details of the interrupt cycle as it relates to the RISC-V MCU's FSM.

- Transitions from the fetch state are unconditional, the fact that the state diagram is now supporting interrupts does not change that.
- The state diagram does not show the fact that acting on an interrupt depends on a bit in an external register (the CSR) being set. In other words, the FSM can only enter the interrupt cycle if the interrupts are “enabled” by that external bit. The hardware controlling the enabling and disabling of interrupts is external to the FSM so we do not include it in the state diagram. We discuss the specifics of this in Section 18.3.
- The state diagram supports interrupts, which means that the FSM can transition to the interrupt state. The FSM can transition to the interrupt state from either execute state (for most instructions) or the writeback state (for load-type instructions).
- All instructions will complete execution before entering the interrupt state. If hardware is currently executing a load-type instruction and there is a pending interrupt, the FSM transitions to the writeback state to complete execution of the load-type instruction. Once again, any instruction that

the FSM is implementing must complete before having the possibility of entering an interrupt cycle.

- The interrupt state in the FSM is responsible for controlling the various operations required to support the RISC-V interrupt architecture. We'll save the details for a later section, but generally, the control units send out the control signals that implement the various operations associated with the interrupt processing, which is similar to the processing of a subroutine.
- The interrupt cycle we speak of is associated with “going into an interrupt”; note that there are no special FSM states associated with exiting an interrupt cycle. Exiting interrupts is a notion associated with exiting the interrupt service routine, which the MCU does under program control (the `mret` instruction). We describe the full details in Section 18.4.3.
- As the state diagram shows, it appears possible that the FSM can go immediately back into an interrupt cycle after it receives an interrupt. For reasons you'll see later, one function of the interrupt architecture that we'll discuss later prevents this from occurring.
- The RISC-V MCU happens to have an interrupt cycle comprised of a single state. In general, the amount of “stuff” that needs to be done to support the interrupt architecture determines the length of the interrupt cycle. The MCU happens to be able to do everything it needs to do to implement the interrupt architecture in a single state; this would not necessarily be true of other MCUs.
- The FSM includes an asynchronous reset signal, RST. This is signal connects to the reset signal on the RISC-V OTTER MCU.

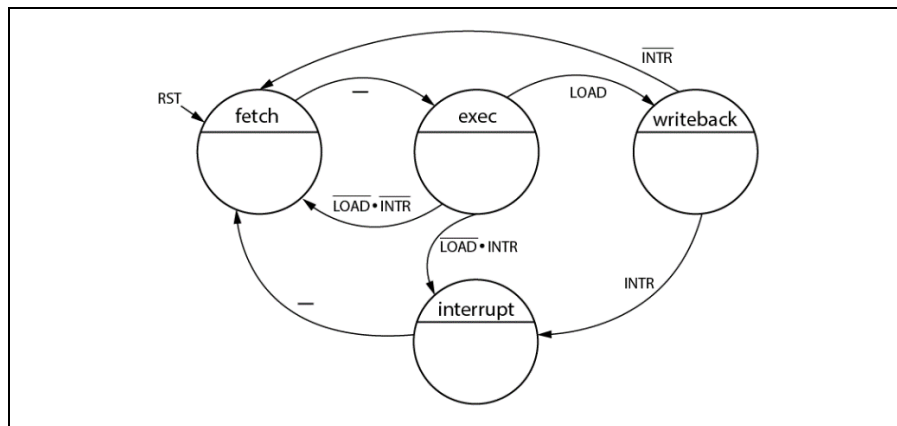


Figure 18.2: The RISC-V MCU control unit state diagram.

18.3 Interrupt Support Hardware

Implementing the interrupt architecture on the RISC-V MCU requires two new modules and modifications to existing modules. The support hardware has two basic functions. First, it provides a form of control by allowing the MCU to either act on or ignore the asserted interrupt signal. Second, it provides circuitry that basically mimics the RISC-V MCU's support for subroutines. The next sections describe this hardware.

18.3.1 The Interrupt Masking Circuitry

The RISC-V MCU can ignore pending interrupts under program control (using instructions). In this context, a pending interrupt is an asserted signal external to the interrupt that has the ability to cause the MCU to go into an interrupt cycle. Figure 18.3 shows the RISC-V OTTER MCU circuitry that controls the interrupt enable; here are the pertinent things to notice about this diagram:

- Entering an interrupt cycle depends on two conditions. First, some external device must assert the interrupt signal, which we list as the INTR signal in Figure 18.3. Second, the interrupts must be “unmasked”; the MIE signal in Figure 18.3 controls interrupt masking.

- The MIE signal is the output of a register that programmers can configure under program control. This register is one of the registers in the CSR module, which we describe in Section 18.3.2. The MIE signal of two inputs to an AND gates and essentially acts as a switch that either blocks the INTR signal or allows the INTR signal to pass through to the CU_FSM.
- When the AND gates blocks the signal, the output of the AND gate is always zero; under this condition, the CU_FSM will never receive an interrupt. We refer to this blocking condition as the interrupts being “masked”, or disabled. In this case, the INTR signal may be asserted, but it can’t pass through the “dead” gate. When the signal is allowed to pass through this gate, we refer to this condition as the interrupts being “unmasked”, or simply, enabled.
- The INTR external input signal in Figure 18.3 is the same INTR signal in the state diagram of Figure 18.2.

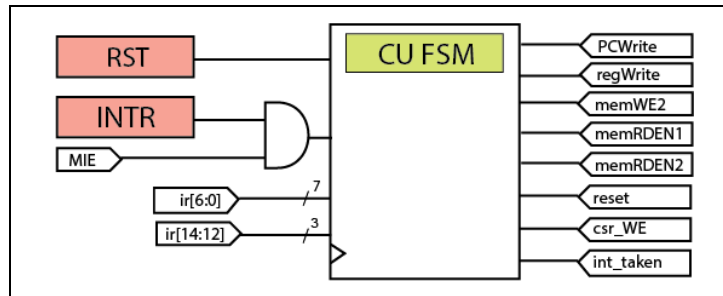


Figure 18.3: The interrupt masking control circuitry.

The controlling signal to the AND gate represents the output of a register in the CSR module. We refer to MIE as a register, but it is actually a flip-flop, which is a 1-bit register. Programmers can write to this register under program control, where setting and clearing the register is equivalent to unmasking and masking the interrupts, respectively. The masking control register is one of three registers in the CSR module, which we refer to as CSR[mie]. We describe the CSR register in the next section in more detail. The notion of “ie” such as in “mie” is common in MCU-related discussion; the acronym stands for “interrupt enable” and generally represents a positive logic signal.

18.3.2 The Control and Status Registers (CSRs)

The control and status register (CSR) is a module that controls various operations associated with the RISC-V OTTER interrupt architecture. The CSR module contains three registers: 1) the **mie**, 2) the **mepc**, and 3) the **mtvec**. Each of these registers has a distinct function supporting the interrupt architecture. Table 18.1 provides a brief summary of the three CSR registers. We describe these registers in more detail in the following sections.

Register	Width	Addr	Description
MIE	1	0x304	Interrupt enable. Interrupts are masked (disabled) when MIE=0 and unmasked when MIE=1 (enabled).
MEPC	32	0x341	Holds the return address to be loaded into PC upon return from interrupt, which is indicated in code with the mret instruction.
MTVEC	32	0x305	Holds the vector address (first instruction of ISR), which the hardware loads into PC upon entering the interrupt cycle.

Table 18.1: CSR register names and descriptions.

Figure 18.4 shows the interface to the CSR module. Table 18.2 provides a brief description of the input and output signals associated with the CSR module.

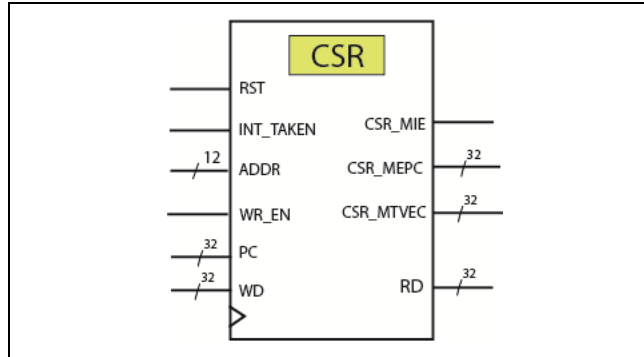


Figure 18.4: The CSR Module Interface.

CSR Interface Description	
Input Signals	Description
RST	Resets register values in the CSR. This input is output by the CU_FSM
INT_TAKEN	Indicates that the CU_FSM is in the interrupt cycle. This input is output from the CU_FSM.
ADDR	The address value which acts as a register select for reading and writing CSR registers. This input is the 12 MSBs from the current instruction.
WR_EN	The write enable for the CSR registers. This input is output from the CU_FSM.
PC	The current PC, which loads into CSR[mepc] when the MCU acts on an interrupt.
WD	The data to write to CSR registers. The input is output from rs1 of the register file.
Output Signals	Description
CSR_MIE	The current value of CSR[mie], which is the interrupt enable bit. This output is a control input to the CU_FSM.
CSR_MEPC	The current value of CSR[mepc], which the return address that loads into the PC when returning from an interrupt (upon issuing a mret instruction).
CSR_MTVEC	The current value of CSR[mtvec], which is the address of the ISR. This address is loaded into the PC when the MCU enters an interrupt cycle.
RD	The value of a CSR register as selected by the ADDR input. This signal is loaded into a selected register in the register file.

Table 18.2: Description of CSR input and output signals.

18.3.2.1 The mie Register

The mie register, or CSR[**mie**], controls whether the external interrupt signal is passed to the RISC-V MCU's control unit FSM. The output of this register is a control input to a simple AND gate. The CSR[**mie**] is set or cleared under program control using the **csrrw** instruction, an instruction mnemonic that roughly stands for CSR read and write.

Table 18.4 shows an overview of the **csrrw** instruction; consult the RISC-V assembler manual for full details. The **csrrw** instruction has the ability to simultaneously read the value in a CSR register and write that register with a new value. For simply writing the individual CSR registers and not reading the same register, programmers can use x0 for the destination register as the usage column in Table 18.4 shows. The source

register (**rs1**) provides the value to write to the CSR[**mie**] register. The **csr** operand is effectively an address that the CSR module uses to differentiate the three registers in the CSR module. The **csrrw** is a base instruction.

Inst r	Form	Example
csrrw	csrrw rd,csr,rs1	<pre>csrrw x0,0x304,x8 # loads value in x8 into CSR[mie] csrrw x7,0x304,x8 # loads value in x8 into CSR[mie] # loads value in CSR[mie] into x7</pre>

Table 18.3: Instruction usage for the **csrrw instruction.**

Table 18.4 shows the underlying instruction format for the **csrrw** instruction. Note that the instruction is not one of the six standard RISC-V instruction types. The CSR module uses the 12-bit **csr** field to choose the register in the CSR module to write to. Programmers use the **csrrw** instruction to configure each of the three registers in the CSR module. The instruction differentiates between the three registers in the CSR module by including a value in the **csr** field in the **csrrw** instruction that the CSR module treats as an address.

Instr	Format
csrrw	

Table 18.4: Instruction format for the **csrrw instruction.**

18.3.2.2 The **mtvec** Register

The **mtvec** register, or CSR[**mtvec**], stores the location in program memory of the interrupt service routine. The CSR[**mtvec**] is a 32-bit register that we officially refer to as the “interrupt vector address”, or simply “interrupt vector”. When the MCU enters the interrupt cycle, the hardware loads the interrupt vector address into the PC under control of the interrupt cycle, which causes program control to “vector” to the ISR. Programmers are responsible for loading the **mtvec** register under program control using the **csrrw** instruction.

18.3.2.3 The **mepc** Register

The **mepc** register, or CSR[**mepc**], stores the address of the instruction that follows the instruction that was the hardware was executing directly before the MCU entered the interrupt cycle. The CSR[**mepc**] register is analogous to the return address (**ra**) associated with standard subroutine calls. After the MCU completes execution of the ISR, the MCU returns to the instruction at the CSR[**mepc**] address by loading this address into the program counter. The RISC-V OTTER hardware loads this value into the PC when it executes an **mret** instruction. Programmers have the ability to write the CSR[**mepc**] under program control, but they generally have no need to do so as the hardware writes this register as part of the interrupt cycle and reads this register as part of the **mret** instruction.

18.4 Interrupts and Program Flow Control

The notion of program flow control in interrupt processing is similar to the program flow control associated with subroutine processing. The processing of the instructions in the interrupt service routine represents normal instruction processing with no new details. What we are interested are the steps the hardware takes upon acknowledging interrupts and returning from interrupts. The following sections describe those details as they relate to the underlying MCU architecture.

18.4.1 Interrupt Initialization

The RISC-V MCU requires two steps for initialization. First, programmers must load the CSR[**mtvec**] register with the address of the ISR. The notion of the “address of the ISR” means the address of the first instruction in the ISR. Programmers load the CSR[**mtvec**] under program control using the **csrrw** instruction.

The second step required in initialization is the unmasking the interrupts. Programmers unmask interrupts by writing a ‘1’ to the CSR[**mie**] register also using the **csrrw** instruction.

Instr	Format
csrrw	

Table 18.5: Instruction format for the **csrrw instruction.**

18.4.2 Acting on Interrupts

Normal program flow control is “interrupted” (for lack of a better word) when two conditions happen: 1) when the interrupts are unmasked (enabled), and, 2) then the value on the interrupt input pin (INTR) on the MCU is asserted. When these conditions are met, the MCU goes into an interrupt cycles.

When the MCU goes into an interrupt cycle, the following things happen as part of that interrupt cycle. The control units issue the appropriate control signals to ensure the operations required by the interrupt cycle complete before starting execution of the first instruction in the ISR. The control units are then responsible for issuing the control signals that implement the following, which they complete as part of the interrupt cycle. Note that all of these items occur simultaneously, and are synchronous with the rising clock edge that ends the interrupt cycle:

- The MCU completes execution of the current instruction, which is after the execute state for most instructions or after the writeback state for load-type instructions. Keep in mind that the signal connected to the RISC-V MCU can change asynchronously in relation to the MCU’s system clock, means interrupts can occur during any phase of the instruction cycle.
- The hardware stores the address of the instruction following the instruction that the MCU was executing when the MCU received the interrupt in CSR[**mepc**]. This instruction address is effectively the current output of the PC.
- Simultaneously to the previous bullet, the MCU loads the interrupt vector (CSR[**mtvec**]) into the PC, which is the interrupt vector address. The next instruction that executes after the interrupt cycle will then be the first instruction in the ISR.
- Also simultaneously to the previous two bullets, the MCU hardware clears the CSR[**mie**] register, which masks interrupts. This, interrupts are masked automatically as a function of hardware and stay masked until they are unmasked under program control.

There are two other items regarding interrupt processing work noting here. First, the hardware is not responsible for saving the operating context before entering the ISR. As with subroutines in the RISC-V MCU, all context saving is done under program control by pushing registers used by the ISR onto the stack, and popping them off the stack before returning from the interrupt. Second, interrupt nesting is not possible on the RISC-V MCU based on the notion there is only one register to store the return address (CSR[**mepc**]). Nesting interrupts is possible on other MCUs, but not currently in the RISC-V MCU.

18.4.2.1 Interrupt Cycle Timing

Examining an example timing diagram is always a good path to gaining a complete understanding of the interrupt cycle. Figure 18.5 provides such a timing diagram. The diagram in Figure 18.5 shows the pertinent signals associated with the interrupt cycle, not including control signals.

The timing diagram in Figure 18.5 makes the following assumptions:

- The CSR[**mtvec**] was pre-loaded the value 0x58, which is the address of the first instruction in the interrupt service routine.
- Some external device generated the interrupt signal (**intr**). This **intr** signal connects to the MCU, which is actually the INTR signal of Figure 18.3.

- None of the instructions are load-type instructions, which means the diagram has no need to include writeback cycles.

Here is a detailed description of the timing diagram in Figure 18.5.

- The interrupt signal (**intr**) asserts asynchronously for approximately 2½ clock cycles; the duration is long enough to cause the MCU to go into an interrupt cycle because the interrupt input was asserted at the end of the execute cycle and the interrupts were unmasked (CSR[mie]=1).
- We don't know what instruction the MCU was executing when we entered the interrupt cycle, but we know it was not a load-type instruction because it entered the interrupt cycle instead of going onto the writeback state.
- The MCU enters the interrupt cycle at the end of the execute cycle associated with the 0x3C instruction. Part of the execute cycle includes advancing the PC, so the PC is now pointing at the instruction that would have been executed had the MCU not entered the interrupt cycle. This instruction is now the first instruction that the MCU executes after it returns from the ISR, which officially makes this instruction's address the "return address".
- Part of the interrupt cycle include storing the ISR return address, which is 0x40, into the CSR[mepc] register, which happens when the interrupt cycle exits and goes onto the fetch cycle. This loading happens because the CU_FSM asserts the **csr_WE** signal as part of the interrupt cycle.
- Another part of the interrupt cycle is to clear the CSR[mie] bit, which masks the interrupts and thus prevents other interrupt from occurring until CSR[mie] is set under program control. This CSR clears this bit as a result of the CU_FSM asserting the **int_taken** signal as part of the interrupt cycle.
- Normal processing continues after exiting the interrupt cycle (and entering the ISR). Be sure to note that the instruction at address 0x58 is the address of the first instruction in the ISR; instructions listed after the interrupt cycle are in the interrupt service routine. Because no instruction has a writeback state, none of the ISR instructions are load-type instructions.

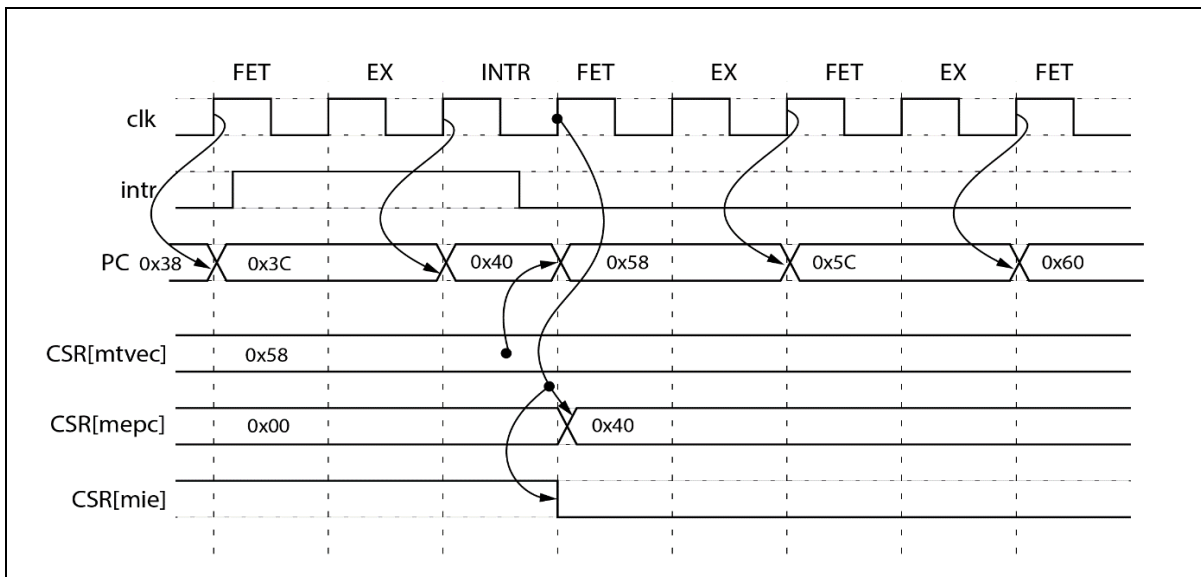


Figure 18.5: Timing Diagram for entering an interrupt cycle.

18.4.3 Returning from Interrupt Processing

When the ISR is complete, the MCU returns program control to the instruction after the instruction that it was executing when it received the interrupt, which the hardware stored in CSR[mepc] as part of the interrupt cycle.

The program indicates the ISR is complete by issuing a special return-type instruction for interrupts: **mret**. The **mret** instruction is base instruction with a format different from the standard six RISC-V instruction formats. For this description, we do not include any context restoration information as context saving and restoring must be done under program and is thus not a function of hardware. The complete sequence of events is as follows:

- The program alerts the MCU hardware to the fact that it has completed processing of the interrupt service routine by issuing an **mret** instruction. The **mret** causes the hardware to load the address in CSR[mepc] into the program counter. Recall that CSR[mepc] was loaded with the return address as part of the MCU's interrupt cycle.

Table 18.6 shows an example of the **mret** instruction; Table 18.7 shows the underlying bit format for the **mret** instruction. As Table 18.7 shows, there are no field codes in the **mret** instruction. Note that the **mret** instruction shares the same opcode as the **csrrw** instruction.

Instr	Form	Example
mret	mret	mret # return from ISR: PC ← CSR[mepc]

Table 18.6: Instruction usage for the return from interrupt instruction.

Instr	Format
mret	

Table 18.7: Instruction format for the **mret** instruction.

18.4.3.1 Return From Interrupt Timing

Returning from interrupts refers to the notion that the interrupt service routine (ISR) has completed and program flow control returns from the foreground process to the background process. The program indicates the end of the ISR by issuing an **mret** instruction, which returns program control to the instruction following the instruction that the MCU was executing when it acted on the interrupt (entered the interrupt cycle). Returning from interrupt processing is inherently different from starting interrupt processing in that there is no special state associated with exiting the ISR. Thus, control units process the **mret** instruction using the same fetch-execute cycle as with all other non-load-type instructions.

Figure 18.6 show an example of the timing associated with returning from interrupts. This image pairs with the image in Figure 18.5 in the timing diagram in Figure 18.6 represents the returning from the ISR that was entered using the timing diagram in Figure 18.5. Here are the most interesting things to note about Figure 18.6.

- The instructions at addresses 0x78 → 0x84 are all part of the instructions in the ISR. The final instruction in the ISR is the **mret** instruction at address 0x84.
- The execute cycle of the **mret** instruction is responsible for loading the CSR[mepc] value into the PC. Recall that the CSR[mepc] was loaded with the address of the instruction following the instruction that was executing when the MCU acted on the interrupt (entered the interrupt cycle). Had the MCU not entered the interrupt cycle, the MCU would have executed the instruction at address 0x40 following the instruction at address 0x3C. The timing diagram shows the CSR[mepc] value loaded into the PC at the end of the execute cycle. The MCU effectively reads the value in CSR[mepc] and loads that value into the PC; being a read operation, the value in CSR[mepc] does not change.
- The CSR[mie] value remains low. Recall that the hardware cleared CSR[mie] as part of the interrupt cycle, which masked the interrupts. The fact the CSR[mie] is low in Figure 18.6 indicates that no instructions in the ISR unmasked the interrupts; recall that CSR[mie] can only change under program control by executing a **csrrw** instruction.

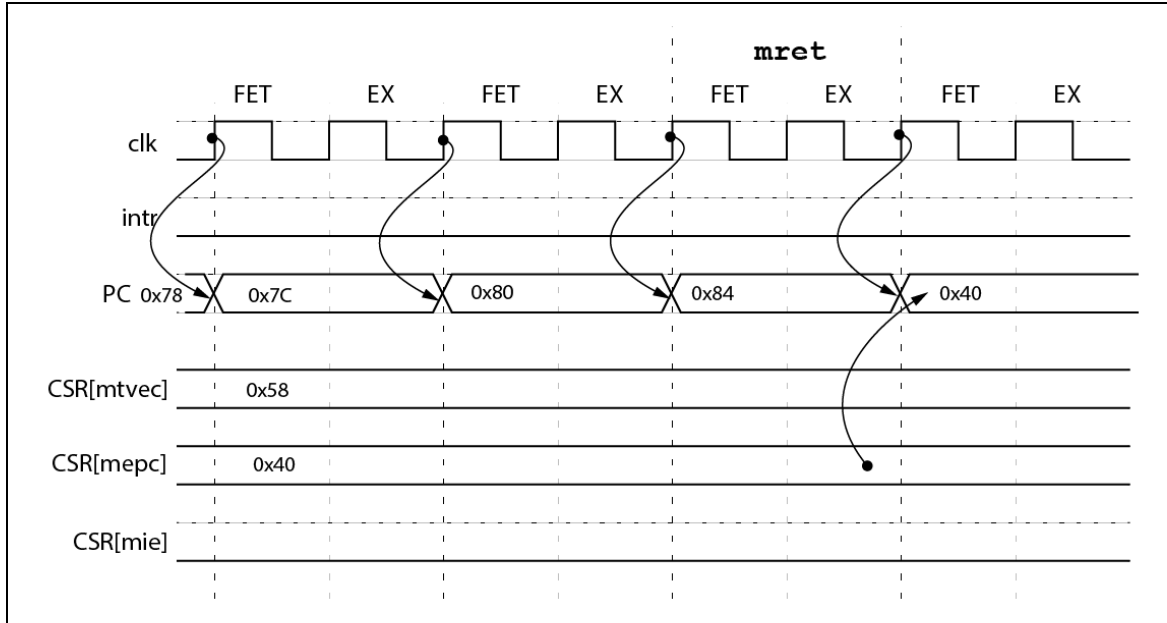


Figure 18.6: Timing Diagram showing an example of exiting an ISR.

Example 18.1: Interrupt Processing Timing

What is the fewest number of clock cycles required to unmask the interrupts after the RISC-V MCU acts on an interrupt? Why is this a killer important question?

Solution: The problem does not provide many details, so it is up to the people reading the problem to figure out what the problem is really asking. And here are the details.

Some external device made the RISC-V MCU go into an interrupt cycle. Part of the interrupt cycle includes the hardware automatically masking the interrupt, which it does by loading a '0' into the CSR[mie] register. Therefore, what this question is asking is what is the minimum number of clock cycles required to unmask the interrupt. The only way to unmask the interrupts is under program control by issuing a **csrrw** instruction. The fastest way to do this is to make the **csrrw** instruction the first instruction in the interrupt service routine.

The **csrrw** instruction is a two-cycle instruction. As part of the execute cycle on the **csrrw** instruction, the **csr_WE** signal asserts and writes enabling data to CSR[mie]. At the end of the execute cycle, the hardware latches the data into the CSR[mie] register and the interrupts are once again enabled.

So in total, the interrupts were only disabled for two clock cycles, which was the fetch & execute associated with the **csrrw** instruction. This information is important because if the signal that generated the interrupt stays asserted, the RISC-V will immediately enter an interrupt cycle again based on the assertion event associated with the interrupt. In most every case, this would be a problem. The moral of the story is to constrain the assertion time-width of an interrupt signal because leaving signal asserted for too long could cause multiple interrupt cycles to be entered for the same interrupt-event. Specific to the timing diagram in Figure 18.7, if the **intr** signal was asserted at the end of the execute cycle (or writeback cycle for load-type instructions) associated with the instruction following the **csrrw** instruction that unmasked the interrupts, the MCU would go into another interrupt cycle for the same asserted **intr** signal, which is problematic for two reasons. First, it would overwrite the CSR[mepc] and the original return address would be lost. Second, it probably not what you want to do in the first place.

The timing diagram in Figure 18.7 shows what the words above were trying so desperately to say; here is the fascinating highlights of Figure 18.7.

- The **intr** line is the interrupt input on the RISC-V MCU (not on the CU_FSM). This input asserts sometime during the first listed fetch cycle but is not acted on until the end of the execute cycle, which means the instruction associated with the left-most FET & EX labels in the diagram is not a load-type instruction.
- The FSM senses the asserted interrupt because the **CSR[mie]** is at a '1' level, thus enabling the asserted interrupt signal **intr** to be sensed by the CU_FSM. The FSM then enters the interrupt state, or the interrupt cycle.
- As a result of entering the interrupt cycle, three things happen: 1) interrupts are masked (**CSR[mie]=0**), 2) the PC loads **CSR[mtvec]**, and, 3) the current PC is stored in **CSR[mepc]**.
- The notion here is that the instruction at address 0x58, which is the first instruction in the ISR, re-enables the interrupts. That instruction is the listed **csrrw x0,mie,x8** instruction, which does several things including asserting the **csr_WE** signal that hardware to write a '1' to **CSR[mie]**. The result is that the **CSR[mie]** becomes a '1' at the next clock edge, which transfers the FSM from an execute to a fetch cycle. You can see from Figure 18.7 that the interrupts can be re-enabled under program control in two clock cycles.

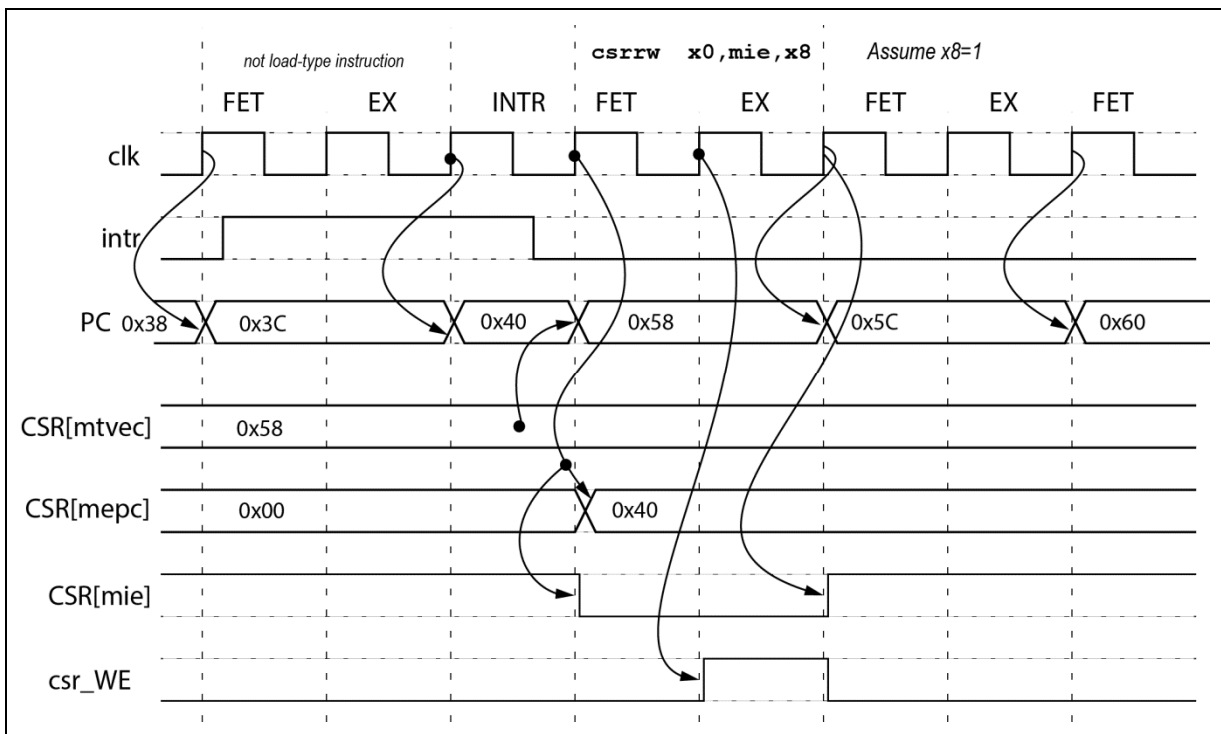


Figure 18.7: Timing diagram showing how fast you can unmask interrupts once in the ISR.

Example 18.2: Minimum Interrupt Pulse Requirement

What is the shortest interrupt pulse width (in terms of clock cycles) that the interrupt signal can be and still guarantee the RISC-V MCU will respond to that interrupt?

Solution: When dealing with the signal connected to the MCU’s interrupt input, we always must consider the internal workings of the MCU in order to guarantee that the width of the signal is long enough so that the MCU is able to act on the interrupt. For the RISC-V MCU, this means that the interrupt must be present on the MCU’s interrupt input at the end of the execution of a given instruction. Recall here that we designed the MCU’s control unit such that the MCU always completes the instruction that it is executing when it receives the interrupt before it acts on the interrupt.

For this problem, we must consider the worst-case timing for the solution. All RISC-V instructions execute in two or three clock cycles, so the worst-case timing would be associated with the load-type instructions, which execute in three clock cycles. Figure 18.8 shows the timing diagram, which provides a basis for description of this example’s solution. Here are the interesting particulars:

The note at (1) show the interrupt signal asserting immediately after entering the fetch state for a load-type instruction. For the MCU to react to this asserted signal, the interrupt must still be asserted by the end of the writeback cycle for the instruction, which note (2) indicates. This diagram shows that the interrupt pulse must be at least three clock cycles to guarantee the signal is present at the completion of the instruction that requires to the most clock cycles to execute.

Also worthy of note is the notion that the MCU asserts the **int_taken** signal upon entering the interrupt cycle as note (3) indicates. Leaving the interrupt cycle causes the **int_taken** signal to de-assert as note (4) indicates. The CU_DCDR and CSR use this signal as part of the RISC-V’s interrupt architecture implementation.

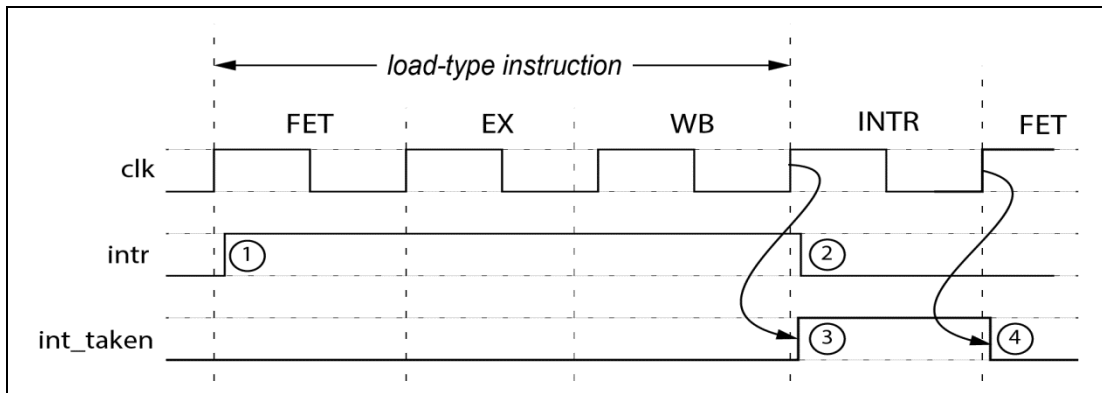


Figure 18.8: Timing diagram showing the minimum interrupt pulse width.

Example 18.3: Maximum Interrupt Pulse Limitation

What is the longest duration of an interrupt pulse width (in terms of clock cycles) that the interrupt signal can be such that it guarantees the RISC-V MCU will not respond to that interrupt more than one time?

Solution: The two problems with the interrupt signal connected to the MCU’s interrupt input is that it can be too short or too long. We’ve covered the case where it is too short; this example covers the issue of what interrupt pulsewidth can be too long. What this question is really asking is how long can the interrupt signal be without it being too long in the worst case. The timing diagram in Figure 18.9 shows the worst case scenario.

- In this case, the **intr** signal is asserted immediately after entering the fetch state, as the note at (1) indicates. We’re looking for the shortest interrupt signal, so our example states that the instruction being executed when it receives the interrupt is a non-load-type instruction, as these instructions execute in two clock cycles.

- The asserted `intr` signal causes the MCU to enter the interrupt cycle, which in turn causes the `int_taken` signal to assert as note (2) indicates. The `int_taken` unasserts after leaving the interrupt cycle. Entering the interrupt cycle causes the RISC-V hardware to automatically disable the interrupts as note (5) indicates.
- Our example then assumes the first instruction in the ISR is a `csrrw` instruction that renables the interrupts. This is a two cycle instruction that results in the `csr_WE` signal being asserted as note (4) indicates, which in turn causes the interrupts being unmasked as note (5) indicates.
- The interrupts are now enabled as note (6) indicates. The next instruction after the `csrrw` is another non-load-type instruction. The constraint here is that the interrupt signal must unassert before the end of the execute cycle associated with this instruction to ensure the MCU will not enter another interrupt cycles based on the same interrupt.
- And the final answer is: six clock cycles. If the interrupt was seven clock cycles, it could still be asserted at the end of the execute state associated with the instruction following the instruction that unmasked the interrupts (`csrrw`).

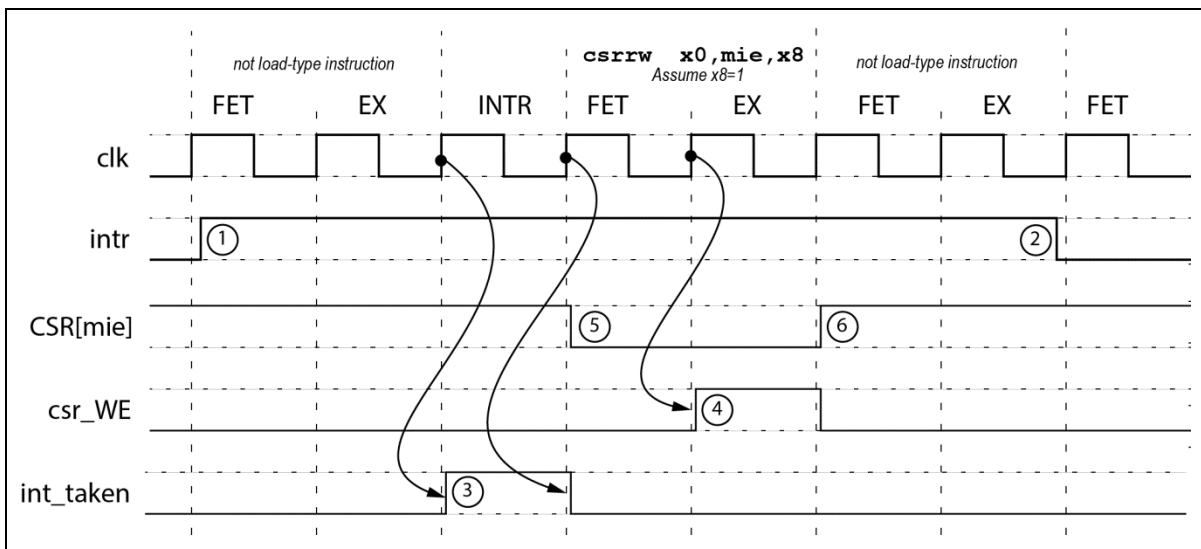


Figure 18.9: Timing diagram showing safe limitation on the interrupt pulse width.

18.5 Other RISC-V Interrupt-Related Hardware Modifications

Section 18.3 described the major modifications required by the RISC-V MCU to support interrupts. This section provides a description of the “less major” modifications. These modifications are in the PC support and the control units (CU_FSM and CU_DCDR).

18.5.1 Program Counter (PC) Support

The program counter needs to support the additions to the program flow control associated with the RISC-V interrupt architecture. While the PC itself is not modified, the MUX controlling the data inputs expand to include two extra jump locations. These two jump values are `mtvec` and `mepc`, are the interrupt vector address and the return from ISR address, respectively. The number of MUX inputs increases from four to six, which necessitates an increase in select control input width (`pcSource`) from two to three. Figure 18.10(a) shows the PC without interrupt support while Figure 18.10(b) shows the PC with interrupt support.

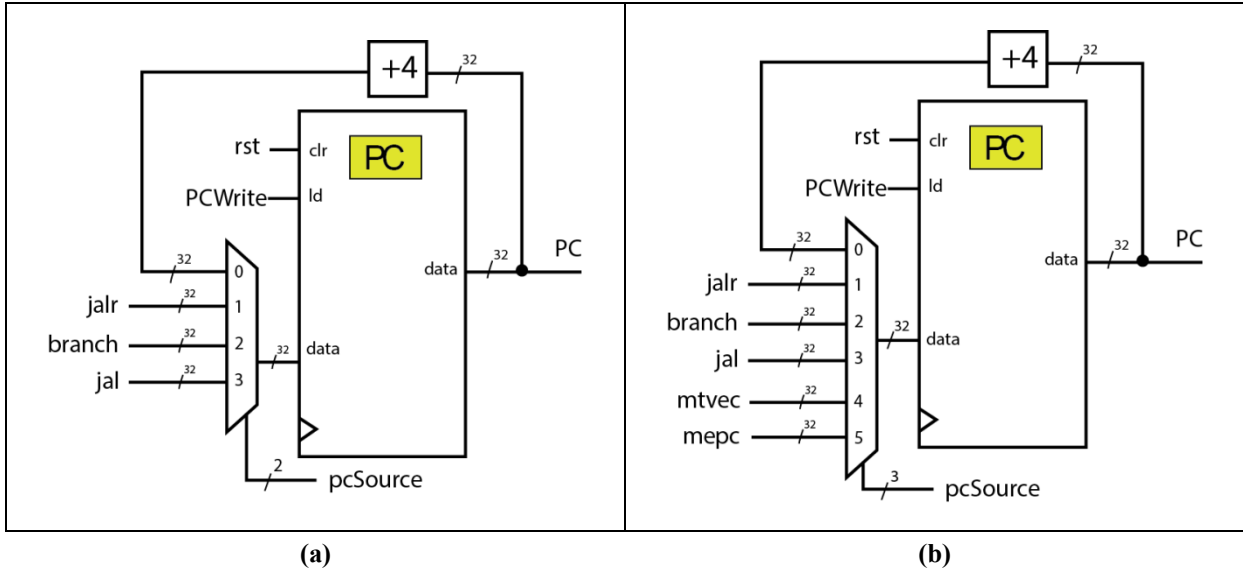


Figure 18.10: The PC-related hardware for no interrupts (a) and with interrupt support (b).

18.5.2 Control Unit Support: FSM & DCDR

The two control units require a modest amount of modifications in order to support interrupts. We group changes to the CU_FSM and CU_DCDR into one discussion because the changes closely involve both topics. Figure 18.11(a) shows the control units without interrupt support and Figure 18.11(b) shows the control units with interrupt support.

Modifications to the CU_FSM fall into two major areas. First, the state diagram needs to be modified in order to support the newly added interrupt cycle (interrupt state). This state controls the parts of the interrupt architecture having to do with acting on interrupts. The CU_FSM also needs to be modified in order to support two interrupt related instructions: **csrrw** & **mret**. The CU_DCDR needs to be modified such that it outputs the correct **pcSource** when the CU_FSM is in the interrupt cycle. Table 18.8 lists the required changes with an added amount of description.

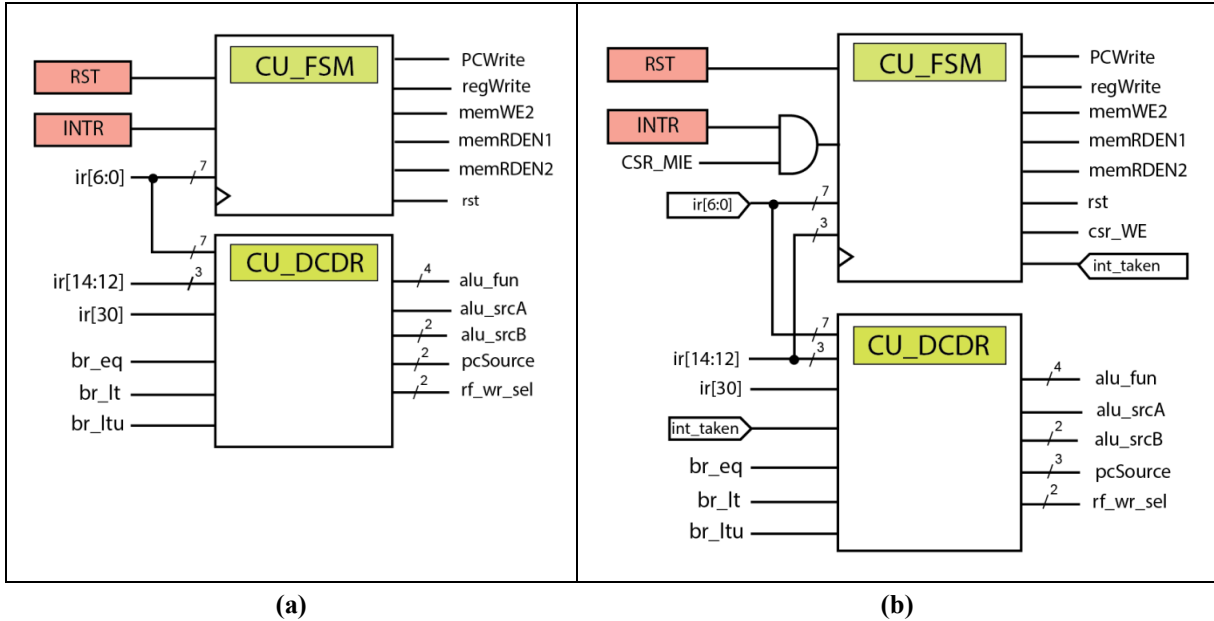


Figure 18.11: The PC-related hardware for no interrupts (a) and with interrupt support (b).

Interrupt Support Mod	Wimpy Explanation
INTR masking control	The INTR input, which was previously connected directly to the CU_FSM, is now connected to a AND gate. The other input to the AND gate is CSR_MIE with is the current state of the CSR[mie] register. The AND gate acts as a switch controlled by the CSR_MIE input, which allows passage of the INTR signal to the CU_FSM.
CU_FSM: csr_WE	The CU_FSM now controls writing of data to the CSR register under program control using the csrrw instruction. The csr_WE output of the CU_FSM is the write enable signal for the registers in the CSR module.
Int_taken	The int_taken is an output of the CU_FSM and an input the CU_DCDR, which is why we left the connection symbols in the diagram.
Ir[14:12]	The ir[14:12] is the funct3 inputs associated with the instruction formats. The two new instructions added to the ISA to support interrupts (csrrw & mret) both share the same opcodes and are thus differentiated using the funct3 opcode. The funct3 opcode were previously only input to the CU_DCDR, but are now input to the CU_FSM.
CU_DCDR: pcSource	The pcSource signal is the select signal for the PC MUX, which expands from two bits to three bits. This expansion allows the MUX to support the added data inputs associated with the mtvec & mepe inputs.

Table 18.8: Description of control unit changes to support interrupts.

18.6 Interrupt Signal-Related Timing Issues

Typical system clock signals for MCUs are relatively fast compared to how quickly you can press and release a hardware actuator device such as a button. This brings up two serious issues that the system designer must deal with in order ensure the overall circuitry (both hardware and firmware) will work properly under all possible conditions. The two issues are: 1) “noise” on the interrupt signal, and 2) the duration of the pulse physically connected to the interrupt input on the MCU.

18.6.1 Interrupt Signal Noise

One type of noise that will affect the operation of interrupt processing is switch bounce. Recall that switch bounce is a characteristic associated with mechanical actuators such as buttons and switches. These mechanical properties of switches generally prevent them from being directly connected to the interrupt inputs on MCU hardware. Mechanical actuators connected to MCU typically are “debounced” in hardware or firmware before being connected to the MCU’s interrupt input.

The problem with switch bounce in the context of MCUs involves timing issues. The specific problem is that typical bounce times are in the millisecond range while the MCU is operating in the nanosecond range. This means that if you press the switch once, the switch contacts can actually “bounce” a few times before arriving at a steady state value, which means that a single button press can generate a separate pulse from each switch bounce. Each of the bounces can generate a separate interrupt, which is generally not intended from a single button press.

You can’t solve with this issue under program control outside of writing a debouncer in firmware. The main drawback of firmware debouncers is the notion that simple debouncers are computationally expensive because they typically require polling loops in their implementations. Debounce firmware that does not use polling loops require other MCU resources such as timer interrupts. The problem with timer interrupts is the fact that there is only one interrupt on the RISC-V MCU and no timers (typically an internal peripheral). Lastly, if you have many buttons that need debouncing, such as with a keyboard, you’ll want to think of an external hardware solution.

18.6.2 Interrupt Signal Duration

There is another issue you need to deal with in addition to noise on the interrupt input issue. The pulse width of signal interrupt signal connecting to the MCU must comply with two parameters: if the signal is too short or too long, it’s highly probably the interrupts will not work as expected. Here are the details.

1. **Interrupt Pulse Too Short:** If the interrupt pulse is too short, the MCU may not recognize it and the interrupt will effectively go away without the MCU entering into an interrupt cycle. Recall that the interrupt must be present at the end of the execute cycle for most instruction or at the end of the writeback cycle for load-type instructions. An interrupt signal can thus be $2\frac{1}{2}$ clock cycles in duration and still not be long enough to ensure the MCU will act on it in the case of load-type instructions. For most instructions, a pulse width two clock cycles wide is sufficient, but not for load-type instructions. Therefore an interrupt pulse with a minimum width of three clock cycles guarantees the pulse will be caught by the MCU’s FSM, but too short for that interrupt to cause the MCU to enter the interrupt cycle more than once.
2. If the interrupt pulse is too long, it can cause the MCU to enter the interrupt cycle more than once for a single interrupt. The issue here is that we try to keep interrupt service routine short, which means that when we exit the ISR and unmask the interrupts, the MCU could re-enter the interrupt cycle if the interrupt signal was still asserted. Additionally, MCU’s have relatively fast clocks; even a “non-short” ISR executes extremely fast compared to something such as a human button press. In essence, the MCU may attempt to process more interrupts than what actually arrived. Note that automatic interrupt masking when entering the interrupt cycle does not solve this issue.

In actuality, chances are good that when you press the button, the button will remain pressed long enough for the MCU to process the interrupt and return to normal program execution. This presents the situation that when the MCU exits the interrupt service routine and the interrupts are unmasked, the interrupt from your last button press will still be there because you have not yet lifted your finger. In this situation, the hardware notices that the interrupt line is still asserted and enters back into a second round of interrupt processing for the same interrupt (namely the initial single button press). In effect, the MCU would service the same interrupt multiple times, which probably is not what you want.

The pulse-width of the one-shot’s output acts independently of the input to the one-shot. In this way, the input signal can remain high for an indefinite period while the output signal remains only briefly asserted before it returns to zero. The final result is that the pulse is long enough to cause the MCU to go into an interrupt cycle (meaning the pulse will be present at the end of the execute cycle) and that the pulse will be not so long that it gone before the end of the next execute cycle. In particular, these issues are:

To avoid the situation where a single “event” can generate multiple interrupts, you can connect the signal that indicates a device needs attention to a “mono-stable multivibrator”, commonly known as a “one-shot”. As the name implies, this device has one stable state, and one non-stable state, or temporary state. The “on” state is the non-stable state, which means it’s only temporarily in that state. The stable state is the off state. When you connect a button to a one shot, the output of the one-shot is only asserted for a fixed length of time, which officially makes it independent of the length of time the button is pressed for. The input of the one-shot connects to the output of the device generating the interrupt signal; the output of the one-shot connects to the RISC-V MCU’s interrupt input. The one-shot circuitry thus provides a relatively short pulse output to the MCU input; this pulse is short enough to ensure that the MCU will only process one interrupt per button press.

18.7 Interrupt Architecture Summary

As you can see from the previous sections, the interrupt architecture of the MCU entails a definite sequence of steps. This sequence of steps ensures a smooth transition to and from the interrupt service routine, as well as protecting the pre-interrupt operating context of the MCU. Here is a brief summary of the steps involved with the acting on an interrupt, executing the interrupt service routine, and returning to the regularly scheduled processing.

- The RISC-V MCU detects an asserted signal on the interrupt input (assume the interrupt is not currently masked on the RISC-V MCU)
- Execution of the current instruction completes and the RISC-V MCU goes into an *interrupt cycle*
- The Interrupt Cycle:
 - The hardware automatically masks the interrupt
 - The address of the instruction what would have been executed next is stored CSR[mepc]
 - The program counter is loaded with CSR[mtvec]
- Execution of the ISR completes with the issue of an **mret** instruction
 - The **mret** instruction loads the address stored in CSR[mepc] into the PC
- Execution resumes at the instruction following the one that was executing when the RISC-V MCU received the interrupt

Table 18.9: Summary of the RISC-V interrupt architecture.

18.8 Chapter Summary

- The interrupt architecture is a term we use to describe all the hardware and hardware-induced operations associated with the processing interrupts. The interrupt architecture is one of the first things you should examine when dealing with a new MCU or CPU, as interrupt driven programs have many distinct advantages over programs that are not interrupt driven.
 - The RISC-V MCU contains a state machine that is responsible for the decoding and execution of instructions and to implement the interrupt cycle. The interrupt cycle on the RISC-V OTTER MCU consists of a single state.
 - The RISC-V OTTER MCU interrupt architecture uses a set of three registers to implement the interrupt architecture. These three registers reside in the CSR module; we refer to them as the CSR[**mie**], the CSR[**mtvec**], the CSR [**mepc**].
 - The interrupt signal is a signal from a device external to the RISC-V MCU. When this signal asserts, it can cause the RISC-V MCU to enter an interrupt cycles. The RISC-V MCU can choose not to act on active interrupts if the interrupts are *masked*; in these case, the hardware ignores the asserted external interrupt signal. The interrupt masking hardware consists of a single bit, CSR[**mie**] that shares an AND gate input with the external interrupt signal. If the CSR[**mie**] bit is a zero, the AND gate output is a zero; otherwise, the AND gates passes the external interrupt signal to the CU_FSM.
 - Interrupts are similar to subroutines and they are thus part of the RISC-V MCU program flow control. Interrupts differ from subroutines in that they require initialization of the interrupt vector address and unmasking the interrupts, both of which are done under program control.
 - The RISC-V OTTER MCU “acts” on interrupts by entering the interrupt cycle. The interrupt cycle then does the following under hardware control: 1) mask the interrupt (CSR[**mie**]), loads a return address into CSR[**mepc**], and loads the interrupt vector address (CSR[**mtvec**]) into the PC. Returning from in interrupts are down with the **mret** instruction, which loads the return address from CSR[**mepc**] to the PC.
 - In order to support interrupts, the MUX that provides the PC with an address expands to include CSR[**mepc**] (for returning from interrupts) and CSR[**mtvec**] (branching to the ISR). Interrupt support for the CU_FSM includes an interrupt state and the addition of the **funct3** opcodes as inputs. The CU_FSM also controls the CSR with the **int_taken** signal and the CSR_WE signal. The **int_taken** signal also notifies the CU_DCDR that the CU_FSM is in the interrupt cycle.
 - The interrupt signal input to the CU_FSM must have certain properties in order for the RISC-V MCU interrupts to work properly. The interrupt signal can’t be too short of the CU_FSM may miss it because the it only looks for it on the clock edge before transitioning back the fetch cycle. The interrupt signal can’t be too long or the signal may be still asserted when the program unmask the interrupt under program control. The interrupt signal should be debounced if connected to a mechanical switch and connected to a one-shot if the signal could stay asserted for a long amount of time.
-

18.9 Chapter Exercises

- 1) Interrupt service routines are very much like subroutine; briefly describe their main difference.
 - 2) Briefly describe the purpose of the *interrupt cycle*.
 - 3) Briefly explain why MCUs do or don't require interrupt cycles.
 - 4) Briefly describe why transitions from the fetch state to the execute state are always unconditional; be sure to include a comment regarding interrupts in your answer.
 - 5) Briefly describe what determines the number of FSM states there need to be in an interrupt cycle. Briefly describe how the CU_FSM knows whether to enter an interrupt cycle or not.
 - 6) Briefly describe the four conditions that must be present in order for the CU_FSM to enter an interrupt cycle.
 - 7) Briefly describe how the MIE signal acts as a troll that allows the external interrupt signal to pass through to the CU_FSM or not.
 - 8) Briefly explain the following statement: *The CU_DCDR only knows about only one state in the CU_FSM.*
 - 9) Briefly describe whether it is possible to write a value to the CSR[mepc] register under program control.
 - 10) Briefly describe why programmers don't have a pressing need to ever write the CSR[mepc] register.
 - 11) Briefly describe why the RISC-V OTTER MCU complete execution of the current instruction before acting on an interrupt.
 - 12) Briefly describe the three things that occur in hardware as part of the interrupt cycle.
 - 13) Briefly describe why there is a special state (the interrupt cycle) for entering interrupts, but no special state for exiting interrupt service routines.
 - 14) Briefly describe how the CU_DCDR knows that the MCU is currently in an interrupt cycle.
 - 15) Briefly describe the CU_DCDR's responsibilities during an interrupt cycles.
 - 16) Briefly describe any context saving mechanism that the RISC-V MCU hardware is responsible for.
 - 17) Briefly describe what a programmer must do to allow the nesting of interrupts on the RISC-V OTTER MCU.
 - 18) Briefly describe why the RISC-V OTTER MCU can't support nested interrupts.
 - 19) The interrupt signal should be at least three clock cycles in duration to ensure that it "works" properly. Briefly describe the reason for three clock cycles.
 - 20) Briefly describe the problem with the interrupt signal being too short.
 - 21) Briefly describe the problem with the interrupt signal being too long.
 - 22) Briefly describe the difference between an **ret** and **mret** instruction.
 - 23) I decided that I wanted to use a **ret** instruction to return from an interrupt service routine. I did this by placing a return address in the return address register before issuing a **ret** instruction. Briefly describe whether this approach could work or not, and if it could work, under what conditions.
-

19 Miscellaneous RISC-V MCU and Other Architecture Details

19.1 Introduction

While the previous chapters discussed many aspects of the RISC-V MCU, there are a few more topics we need to introduce to provide the overall big picture. When we say “big picture”, we mean the big picture in terms of both the MCU and basic computer architectures in general. In truth, we could not easily introduce a few subjects out there earlier because we had not yet provided you with the background to facilitate that discussion. This chapter ties together many of the issues regarding the MCU and computer architecture in general.

Main Chapter Topics

- **OVERVIEW OF RISC AND CISC ARCHITECTURES:** This chapter describes the RISC and CISC architectures including their main accepted differences.
- **STANDARD ARCHITECTURES:** This chapter introduces the standard architecture types of Harvard and Von Neumann architectures.
- **OVERVIEW OF LEVELS OF MEMORY:** This chapter describes the notion of memory levels as they relate to basic computer systems.
- **SEVEN-SEGMENT DISPLAY MULTIPLEXING:** This chapter describes the popular digital design topic of display multiplexing.

Why This Chapter is Important

This chapter is important because it describes some the non-architectural but still important details involving the RISC-V MCU.

19.2 RISC vs. CISC Architecture Types

Out there in computer land, you’ll find that people attempt to model computer architectures as one of two different types. Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). The names probably meant something at one time, but they’re now something that you should not take literally.

Since the dawn of computers, or even before, there has been an ongoing argument of which architecture is “better”: RISC or CISC? To understand the parameters of the RISC vs. CISC argument, you must understand the current accepted differences between RISC and CISC architectures. Table 19.1 lists the commonly accepted characteristics and differences between RISC and CISC architectures.

RISC Architectures	CISC Architectures
<ul style="list-style-type: none"> • All instructions execute in the same number of clock cycles • Instructions are relatively simple compared to CISC architecture • System clock speed is relatively fast • Instruction set has relatively few addressing modes • Has a relatively large register file 	<ul style="list-style-type: none"> • Instructions require varying numbers of clock cycles for execution • Instructions are relatively complex compared to RISC architectures • System clock speed is not overly fast • Instruction set has relatively many addressing modes • Has a relatively small register file

Table 19.1: The characteristics of RISC & CISC architectures.

Generally speaking, the instructions on a RISC machine relatively simple, which allows them to execute in a few number of clock cycles. Conversely, instructions on a CISC machine can be complicated and thus require more instruction cycles or longer clock periods to execute. In the end, to complete the same task, programs written for a RISC architecture are longer (meaning more instructions) because the instructions are simple so there must be more of them to complete the same task. The same program functionality implemented on a CISC architecture is shorter (less instructions) because each instruction can generally do more stuff. Nevertheless, because each instruction is doing more stuff, the system clock period typically must be longer. The general thought here is that it takes more instructions for a RISC computer to perform a given task than it does for a CISC computer. However, the RISC instructions, because of their simplicity, allows for a higher clock speed. This is a classic trade-off in computer-land.

So how do we classify the RISC-V MCU architecture? Wow, that's a tough on. As the name implies, we consider it a RISC architecture because it contains most of the characteristics of a RISC architecture. The only characteristic that it violates is that the current RISC-V implementation does not implement all instructions in the same amount of clock cycles. Recall that the load-type instructions require three clock cycles to complete. Hey, what's one instruction?

The RISC vs. CISC thing is quite important. But then again, you've probably worked on many different computers without knowing whether the underlying architecture was a RISC or CISC. The thing to note here is even though you didn't know this information, you were able to work with that computer. Often times in computer land we simply do all our work at a high level, such as with a higher-level computer programming language. We write our programs and let the compiler do the grunt work. Once again, if you know something about the underlying architecture, you'll be able to write programs that are more efficient. This notion becomes even more important as the complexity of the underlying hardware increases. This knowledge includes both the architecture and the instruction set; this knowledge is something pure programmers could not use. Don't ever be a pure programmer.

19.3 Standard Computer Architectures

Any study of computer architectures arrives at the notion of two common architectures: the Harvard architecture and the Von Neumann¹ architecture. Everyone who studies computer architectures should be aware of the characteristics of these two architectures. The problem is that these definitions are not carved in stone and leave some amount of gray area as to their interpretation. Someone created these definitions a long time ago, so even though they're becoming harder to use as a label for an architecture, we still use them. This is a standard interview question that someone is going to ask you if you state you took an architecture course.

The best way to understand the characteristics of Harvard and Von Neumann architectures is to examine high-level models side-by-side. Figure 19.1(a) shows examples of a Harvard architecture and Figure 19.1(b) shows an

¹ The Von Neumann architecture is also known as the Princeton architecture. The story goes that Harvard and Princeton had some type of computer design competition; these two architectures were the result of that competition.

example of an Von Neumann architecture. Both diagrams show the CPU module with two submodules of the *Control Unit* and *ALU*. The main difference is evident in the Harvard architecture has a different memory for both instructions and data while the Von Neumann uses a single memory for instructions and data.

The definition is somewhat more detailed than that though. The true definitions have to do with the datapath of data from the CPU to the memories. In rough terms, if there are separate paths from the CPU to the data and instruction memory, then we consider that to be a Harvard architecture. If there is one datapath from the CPU to memory, then data and instructions must share the datapath.

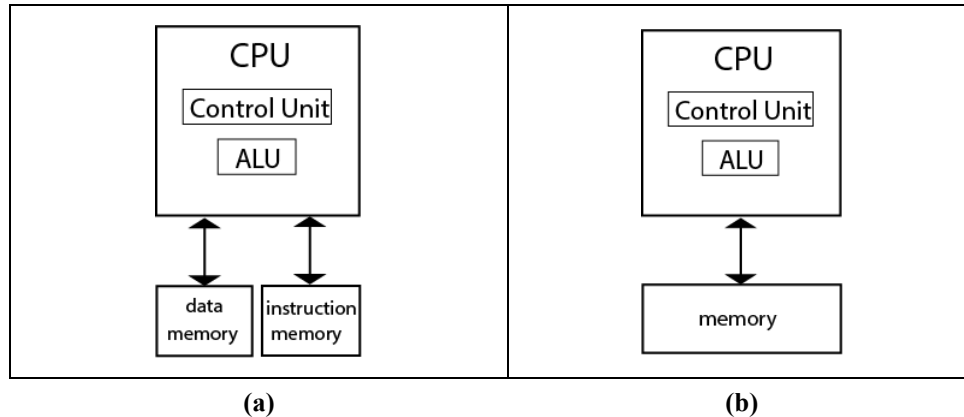


Figure 19.1: Diagrams of Harvard (a) and Von Neumann architectures (b).

Yes, we still talk about Harvard and Von Neumann architectures. If one of these was clearly better than the other one, then that is what everyone would use and we would not care about it, so that is clearly not the case. The main ramifications of Harvard vs. Von Neumann has to do with advanced architectures. If there is only one memory for data and instructions, the architecture can face memory bandwidth limitations. Advanced architectures generally use a *pipeline* for instruction execution, which roughly means that instruction execution is divided into distinct sections such that all the sections can be simultaneously executed. Simultaneous execution of multiple parts of an instruction can require, for example, that a memory be read (such as an instruction for a fetch cycle) at the same time as memory is written to (such as with a store-type instruction).

When you pick up a new MCU, looking at the general architecture is always one of the first things you must do. Often times MCU datasheets describe their processor as either being Harvard or Von Neumann, but modern architectures often run into the gray areas of these definitions. The best approach is to understand general computer architectures at a high level, which allows you to relatively quickly understand any new architecture you see.

19.4 Levels of Memory

A typical computer system has many types of memory and many memory entities. Even a relatively simple computer such can have several relatively large structured memories (such as register files and main memory), and several special register memories (program counter and special use registers supporting interrupts). Computer architects are always attempting to classify items in computer architecture, and one of the primary targets is the structured memories. For this discussion, we're interested in the two writable structured memories in the RISC-V MCU architecture, which are the register file and the main memory.

Computer architects often speak of “levels of memory” in the context of structured memories in a given computer system. In this way, they consider a typical computer model to have multiple *levels* of memory. We differentiate these different levels of memory by one thing: how long it takes to do something useful with the memory in those levels. We're careful here not to classify these different memories solely by access times, as we typically do with structured memories because there are other special usage parameters involved. The RISC-V MCU has two structured memories: the register file and the main memory; these are the memories we'll base this discussion on, knowing that we could also have large memories external to the RISC-V MCU as well.

We consider the register file to be a *lower level* of memory than the main memory, but not because it has a faster access time. For this discussion, we don't care about access times; what we do care about is the amount of time it requires to do something useful with the data in those memories. Recall that all useful operations in the RISC-V MCU occur with instructions that access the register file, which means that if we have data in a register, we're immediately ready to operate on it. This differs from data in the main memory in that if we want to operate on data stored in the main memory, we first must load it into a register in the register file. The notion of loading the value from memory into the register file requires an extra instruction (some type of load instruction), and thus doing something useful with the data in main memory is "slower" (takes more time) than doing something with data already in a register. Because of the extra time it requires to do meaningful work with the data in main memory compared with data in the register file, we consider main memory a higher-level of memory. And of course, what follows is the notion that the register file is a lower-level memory.

A typical computer system such as your laptop computer has many different levels of memory. For example, the lowest level of memory may be some type of general-purpose registers such as the register file in the RISC-V MCU. More complex computer architectures typically have many more levels of memory; so for something like your laptop, the hierarchy of memory starts with low with registers, then goes to various cache memories (for data and/or instructions), main memory, external RAM, hard-drive, thumb drive, tape drive, etc.

The general thought with levels of memory is that the lower the level, the more expensive it is, the faster it is in terms of usage and/or access, and the less your system has of it. You certainly see this with the RISC-V MCU, as there are several reasons why the register file memory is more "expensive" than the main memory. Note that if register file memory did not have associated expenses, the system architects would have provided a lot more of it. On the other end, memory in hard drive is cheap, plentiful, and requires a relative long time and a variable amount of time to access. It's great that solid-state drives (SSD) are becoming cheap enough and large enough for people to invest in; having a spinning drive in your system is like having a campfire in your living room. Each level of memory has its place in a computer system; exactly what place that is, is something computer architects deal with constantly.

The final note here is that the RISC-V has the ability to interface with external memory peripherals the standard I/O. We typically classify these memories as higher levels of memories depending on the particular memory. Keep in mind that when we think of memory, we tend to think of parallel interfaces. Because having large parallel interfaces uses up many resources, many discrete memory devices have serial interfaces, which would necessarily have slower access times than similar memories with parallel interfaces.

19.5 Switch Bounce

Every mechanical actuator device such as a button or switch has a physical characteristic known as *bounce*, or *switch bounce*. This means that if you press a button, for example, the button contacts usually "bounce" a few times before arriving at a steady state value². The notion of bounce here means that the output of the switch goes on and off (or off and on) a few times before attaining a steady state (does not change any more). The result is that a single button press can generate a separate pulse from each switch bounce. Because the contacts can bounce for up to 50ms (which is a long time in MCU-land), they can cause unwanted effects in a MCU because MCUs typically operate in tens of nanoseconds range.

Figure 19.2 show an oscilloscope output of switch bounce. The top trace in Figure 19.2 shows an idealized off-to-on signal transition while the bottom trace shows the actual transition. Notice the glitches on the bottom trace. Realize that the time scale of the oscilloscope output is rather high (in the millisecond range), which means that each of the glitches in the lower trace of Figure 19.2 are actually pulse of a longer duration. This duration is long enough to allow MCUs to complete tasks associated with the switch activation. This means that although the switch activation in Figure 19.2 only intended for one activation, a MCU connected to the circuit sees eight 0→1 transitions, with only the final 0→1 transition being the one of interest. The concept of bouncing from a mechanical actuator holds true for high-to-low transitions also.

² Steady state in this context means the contacts have stopped bouncing.

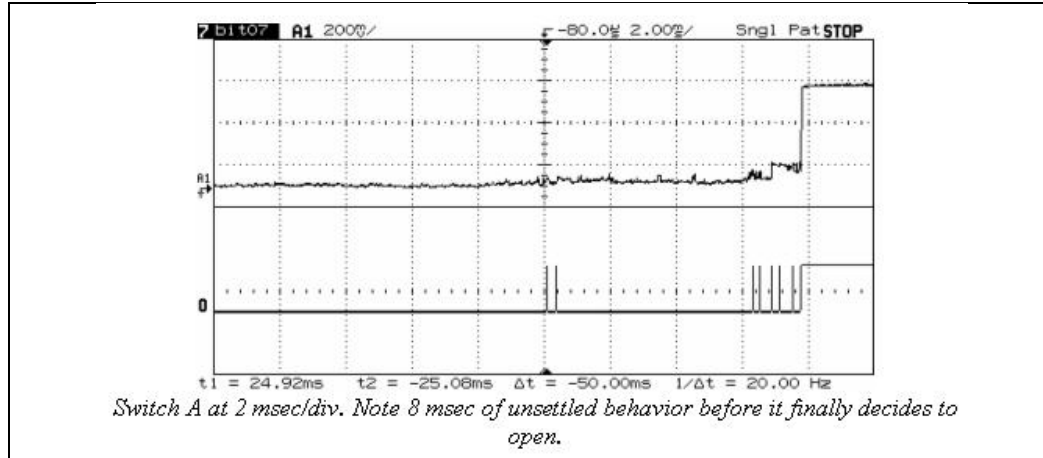


Figure 19.2: Oscilloscope display capture showing actual switch bounce.

Any device that utilizes a mechanical switch (or button) most likely uses some type of “debouncer”, particularly if the switch is in a MCU controlled circuit. System designers must debounce all mechanical switches, but they have two choices for debouncing: hardware or firmware. The model for both hardware and firmware debouncers is the same: the debounce mechanism works by first detecting a change in the signal value (off-to-on or on-to-off, for example), waiting for a specific amount of time, and then checking the signal value again. If the signal value of the switch after the delay indicates the signal is “off”, then the transition on the signal must have been noise, so the debouncer does not pass the signal. On the other hand, if the value on the signal is still in the “on” state after the delay, the debouncer passes the “on” value of the signal. The good news is that the debouncer only passes along a clean signal; the bad news is that the debouncer introduces a delay in the circuit, which increases the response time.

The solution for these types of noise is to apply a “debouncer” to the switch outputs. You must debounce all switches, but you have the choice of debouncing them in hardware or firmware. Figure 19.3(a) shows a block box diagram of a debouncer circuit. There are many approaches to implementing a debounce circuit; the BBD in Figure 19.3(a) represents a digital approach, where we use the clock signal input to “time” the delay associated with the debounce circuit. The timing diagram in Figure 19.3(b) shows an example of the ideal and actual outputs of the debounce module in Figure 19.3(a).

Figure 19.3(b) arbitrarily shows an example of a signal transitioning from low-to-high, which is a function of how the particular hardware and how the hardware designer connects the device in the circuit. The top trace in Figure 19.3(b) shows the idealized switch activation; the middle trace shows the actual characteristics of the switch activation, and the bottom trace shows the debounced switch activation.

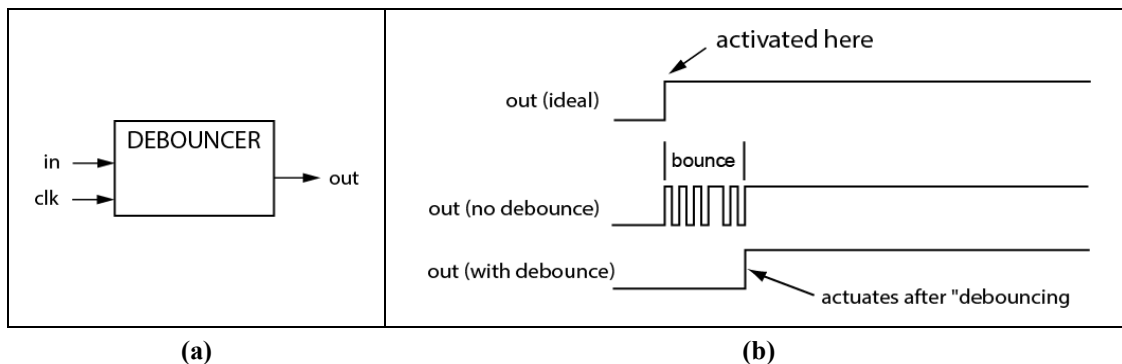


Figure 19.3: A debouncer circuit BBD (a) and the associated timing diagram (b).

19.6 Monostable Multivibrators (One-Shots)

Often times in digital systems we need to have control over signals in order to ensure they perform as designers intend them to in a digital circuit. We generally need to take an output provided by one circuit and operate on that signal such that it conforms to the input requirements of another circuit. We sometimes refer to this as synthesizing a new signal from a given signal; other times we refer to this as filtering the signal. Debouncing a signal is an example of this type of operation.

When humans interface with computers, there are always special interface issues that designers need to deal with. The problem is that computers are fast, while humans pressing buttons are relatively slow. Even the fastest possible human button press looks like forever at the speeds typical MCUs run at. This difference in speed can cause problems if you don't properly handle it.

The notion of a human pressing a mechanical actuator (such as a button) can mean two things. If the "event" in question the notion that someone pressed a button, or is the associated "event" the fact the button continues to be pressed. The problem lies in the case where the event of interest is where someone presses a button (changes state). To ensure the fact that the "button press" event is not interpreted as a "button continues to be pressed" event, we modify the output of the button using a monostable multivibrator, which is longhand for "*one-shot*".

As the name implies, the one-shot device has one stable state, and one non-stable state, or temporary state. The "actuated" state is the non-stable state, which means it's only temporarily in that state. The stable state is the "not-actuated" state, which is the state the one-shot resides in when it's waiting for an event to happen. When you connect a button to a one-shot, the output of the one-shot is only asserted for a fixed length of time, which officially makes it independent of the length of time the button is actually pressed for. The one-shot circuitry thus takes a pulse of unknown length (including the signal transition) and transforms it into a pulse of known length.

Figure 19.4(a) shows an example of digital one-shot circuitry³ while Figure 19.4(b) shows examples of two representative timing diagrams. Once the signal is actuated (goes high or low), the one-shot is activated. The output of the one-shot becomes a pulse independent of the duration that the input signal is high (rising-edge signal) or low (falling-edge signal). The output pulse width requirements are dictated by the input requirements of the circuit that the one-shot connects to. Any one-shot worth dealing with includes the ability to configure the pulse-width. We refer to the high state associated with the rising-edge pulse of the debounce circuit's output as the unstable state (because it's momentary) while the low state is the stable state. There is only one stable state, hence, the circuit exhibits mono-stability.

Another item worth noting about the circuit in Figure 19.4(a) is the fact that it contains a clock input. Digital one-shots work by using internal counters to time the input sampling delay; the counter is a sequential circuit, which is driven by the clock input. One interesting artifact from this design is that the digital one-shot inherently synchronizes the *in* input to the system clock of the given circuit. The input to the one-shot can thus be asynchronous, but the output is always synchronous.

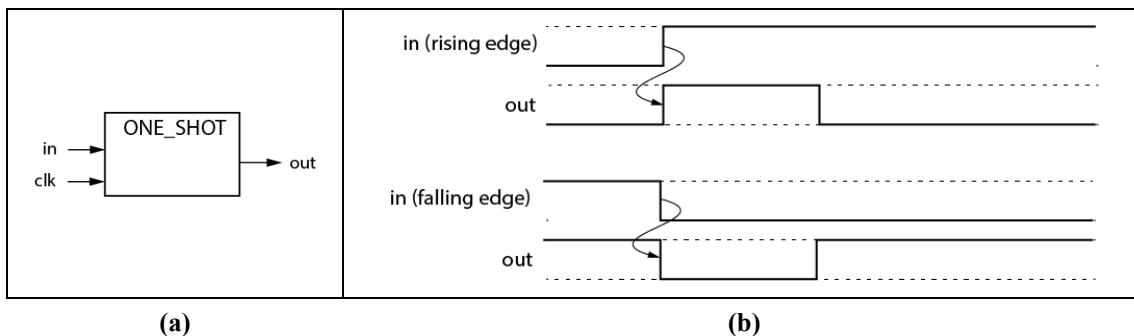


Figure 19.4: A one-shot circuit BBD (a) and an example timing diagram (b).

Figure 19.5 shows a diagram showing a circuit that is both debounced and one-shotted. The signal labeled (A) represents the signal from the button. This signal shows a button press and the actual reaction on the signal due

³ We can implement one-shots as purely analog circuits as well.

to switch bounce. The signal labeled (B) represents the ideal output of the button press. Note differences between signal (A) and (B) are the toggling of the switch after the initial actuation. The signal labeled (C) represents the classic debounced button, which shows the switch actuates only after the switch has complete it bounce routine. The debounced characteristic in (C) is fine for some applications, but not for signals that connect directly to the RISC-V MCU's interrupt input. The signal in (D) shows what the MCU requires, which is essentially a signal that is both debounced and connected to the one-shot.

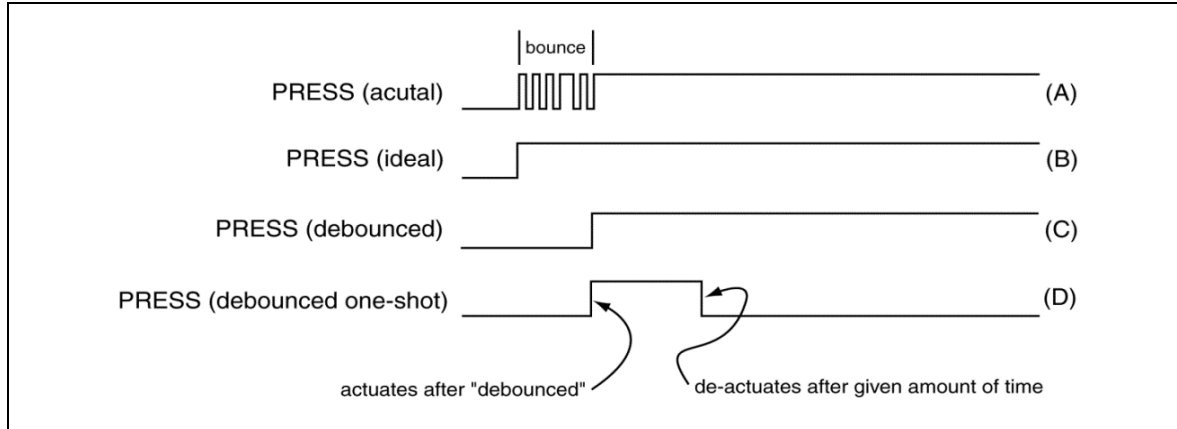


Figure 19.5: Timing diagram showing a signal both debounced and one-shotted.

19.7 Seven-Segment Display Multiplexing

The seven-segment display is one of the most common display devices in the universe; we generally use these devices to display decimal numbers. The seven-segment display can display any of the digital digits (0-9) using, wait for it, seven segments, which are typically LEDs (but sometimes are LCDs). The displays are relatively simple so it is an attractive approach to displaying numbers, particularly numbers with a relatively large number of digits. Additionally, seven-segment displays do an adequate job of displaying alpha hex digits (a-f), though it's a mix of upper and lower-case letters.

There are two main reasons hardware uses seven-segment display multiplexing. First, it saves inputs. Imagine a four-digit 7-segment display that included decimal points. If the displays did not use multiplexing, they would require 32 separate pins (outputs) to properly drive the device, which is many outputs, but embedded systems programming considers outputs relatively expensive in embedded designs. A four-digit multiplexed display would require only 12 outputs to drive it, which is 20 less inputs than an equivalent non-multiplexed display.

We represent individual decimal numbers by turning on specific sets of the segments. For this discussion, we'll refer to 7-segment displays implemented with LEDs. Referencing the seven segments is done by assigning unique letters to each of the segments. Figure 19.6(a) shows the most common listing of these segments. Figure 19.6(b) shows a 7-segment display creating the illusion of a '0' by lighting all the segments except segment 'g'. Figure 19.6(c) shows segments a, b, c, d, and g lit to simulate the number '3'. Most 7-segment displays actually have eight segments because they typically include a decimal point with each set of seven display segments.

The seven-segment display can use fewer inputs because of the way it handles the individual digits. In a seven-segment display, all of the segments are driven simultaneously (a-g and the decimal point) meaning that when you turn on one segment, all the segments are potentially activated. You control which segment actually turns on by actuating the correct gate device associated with each display⁴. Driving displays in this manner has the added benefit of requiring less power because only one display is on at a given time⁵. Actuating a segment on most 7-segment displays is a two-step process. You need to both turn on the LED and actuate the individual 7-

⁴ Seven-segment displays come in either "common anode" or "common cathode" configuration, which you can effectively consider an on/off switch for a given display. Turning on a single segment on a single display requires that you both drive the segment and turn on the proper display.

⁵ This is not a super strong reason as was the first, but people often consider it significant.

segment display. Both of these actuation steps involve sending a logical ‘1’ or ‘0’ to the device. You must consult the reference manual for the particular display you’re working with to figure out how it works.

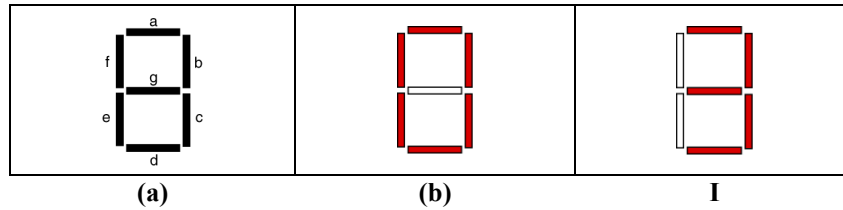


Figure 19.6: The amazing 7-segment display (a), and ‘0’ (b), and a ‘3’ (c).

The approach is to actuate each individual seven-segment display sequentially in a circular manner thus ensuring each display activates for the same amount of time before going onto the next display. This multiplexing action takes advantage of the human visual system’s characteristic of *retinal persistence* in order to make the display appear as if the individual digit displays activate simultaneously while in fact only one digit of the display actuates at a time.

You can implement display multiplexing in either hardware or firmware; this section describes a firmware implementation⁶. Imagine a development board that contains four seven-segment displays; each display contains eight segments (including a decimal point). There is one signal per each segment for all of the four displays on the board; the activation of a single segment on a single digit display involves actuating that segment and actuating the display enable (anode) for that digit. Each individual segment of the seven-segment displays on the development boards connect to each other, so writing to one individual segment of a display is actually writing to that segment on all four seven-segment displays

To make the displays appear as if they are constantly on without the appearance of flicker, you need to leave each display actuated for a given amount of time using a firmware delay function such as the one in Figure 19.7. The idea is to do no further processing for a set period of time after firmware actuates a display before going on to actuate the next display.

The firmware delay in Figure 19.7, however, is not an efficient use of the microcontroller’s resources since the program execution is in a tight loop that effectively does no meaningful processing. It is for those reasons that we often refer to delays such as these as *dumb loops*. It is, however, a viable firmware-based approach to providing a time delay.

```

#-----
# Subroutine: delay_ff
#
# Delays for a count of FF. Unknown how long that is but it
# is plenty of time for display multiplexing
#
# tweaked registers: x31
#-----
delay_ff:
    li    x31,0xFF    # load count (relative big value)
loop:    beq   x31,x0,done # leave if done
        addi  x31,x31,-1 # decrement count
        j     loop      # rinse, repeat
done:    ret           # leave it all behind
#-----

```

Figure 19.7: A standard delay subroutine.

There are two reasons why we need to use a delay. First, because the MCU is so fast, we need to provide enough time for the LED to turn completely one. Second, the way we need to multiplex the displays require that each display be off for at least a small amount of time relative to the amount of time they display is on. To make the number appear as bright as possible, we want to ensure that the percentage of time the display is off compared to when the display is one is relatively small.

⁶ You hopefully implemented a 7-segment display in your introductory digital design course.

Figure 19.8 shows a flowchart that models the approach to multiplexing two 7-segment displays in firmware using a common cathode-type seven-segment display. Note that code in Figure 19.8 shows one pass of the algorithm; the complete algorithm endlessly repeats the flowchart in Figure 19.8.

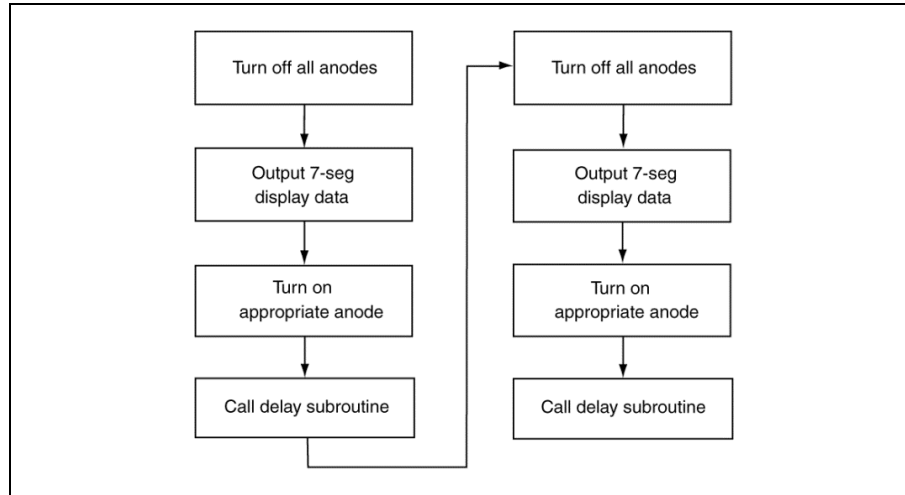


Figure 19.8: Process flow for firmware multiplexing algorithm.

19.7.1 Undesirable 7-Segment Display Effects

Multiplexing 7-segment displays can have on of several undesirable effects if not done properly. Table 19.2 shows a description of several types of effects, their causes, and some ideas on how to correct the issues. If your particular implementation is having issues, you can use Table 19.2 to help solve those issues.

Effect	Cause	Comment
Dim display	Multiplex delay too short	If the segment display is not actuated for enough time, the ration of display off/on time is too small. Increasing the delay fixes this issue.
Flickering	Multiplex delay too long	The operation of the multiplexed display requires that it fool the human visual system (HVS). In this case, the HVS is not getting fooled good enough. Decreasing the delay fixes this issue.
Ghosting	Segment data displayed at wrong time Multiplex delay too short	There are two causes of this problem. First, the wrong segment data is sent to the display that is on for a short time causes some segments to be at different brightness levels than other displays. Fix this by ensuring that displays are off before writing new segment data. Second, if all the segments are mostly dim, it indicates the multiplex delay is too short.
Differing Digit Intensity	Individual displays are not on for equal amounts of time.	This generally means there is an error in the algorithm that implements the multiplexing.

Table 19.2: Common problems when multiplexing displays in firmware.

19.7.2 Lead-Zero Blanking

Lead-zero blanking (LZB) is a simple notion that is typically associated with 7-segment displays. The issue is what to do with “lead zero”, which refers to the left-most zeros in a multi-digit display of a decimal number less than ten. Although placing zeros in to the left of a digital number does not change the value of that number, if

makes the number hard for humans to read. The better option is to “blank” the lead zeros, which means to not display them. For example, if you had the number 32 on a four-digit display, it is better to blank the lead zero so that the display shows “_ _ 3 4” rather than “0 0 3 4”. Then again, if the number you need to display is “0”, you want to display one “0” because a blank display would cause people to wonder if the display was actually working or not.

19.8 Chapter Summary

- Despite the MCU being a relatively simple device, there are timing issues associated with the instructions that you must understand in order to understand the overall operation of the RISC-V MCU. If you're just a programmer, you don't really need to understand these timing details. But if you know/understand anything regarding the underlying RISC-V MCU hardware, you must understand basic timing issues.
 - The RISC-V MCU "wrapper" provides an interface between the RISC-V MCU and a development board. The RISC-V MCU wrapper is a relatively simple HDL model that interfaces the RISC-V MCU I/O with the various input (such as buttons and switches) and output devices (such as LEDs) on the development board. The highlights of the wrapper include a MUX for the development board's inputs and a decoder and register for the development board's outputs.
 - Reduced instruction set computers (RISC) and complex instruction set computers (CISC) are the two main classifications that we try to place computer architectures into. RISC architectures have instructions that execute in the same number of clock cycle, relatively large register files, few addressing modes, and relatively simple instructions. CISC architectures have all the opposite characteristics. Programs written for RISC architecture generally have more instruction than the same program for a CISC architecture, but the RISC instructions generally execute in a smaller amount of time.
 - The notion of levels of memory refers to how fast a system can access (read and write) that memory. Generally speaking, lower levels of memory have faster access times, are but lower storage capacity than higher levels of memory.
 - One-shots, also known as monostable multivibrators, filter signals to make them more effective to systems with special constraints on the input. The MCU interrupt input has special constraints in that the interrupt signal can cause problematic behavior if the signal is too short or too long. This one-shot solves this problem.
 - Switch bounce is a known characteristic of all mechanical switches. When switch contact is initiated or uninitiated, the switch contacts can be unstable (touch and retouch many times) before arriving at a steady state. The switch bounces issue must be handled in firmware or hardware to make the system work in a predictable manner.
 - There are two "standard" computer architectures that many computer people refer to: the Harvard and Princeton (usually referred to as Von Neumann architecture) architectures. The Harvard architecture uses a separate memory for instructions and data, while the Von Neumann architecture uses the same memory for both instructions and data.
 - Seven-segment display multiplexing is a method to have seven-segment displays show many numbers but only use a minimal amount of input pins. Input and outputs are expensive on computer devices; the main purpose of a multiplexed display is to reduce the number of I/O required to drive the device. In a multiplexed display, only one digit actuates at any given time. The displays take advantage of the human visual system's retinal persistence to make it appear that all the display devices are on at the same time. It does not have leaving a single digit display on for a given amount of time, then switching to another digit to repeat the process. When this is done fast enough, the human visual system sees more than one number at a given time.
 - Seven-segment displays typically use *lead zero blanking*, which means they do not display higher-order digits if those digits are zero and do not change the value of the number. If the number to display is zero, the display shows that zero (does not blank it).
-

19.9 Chapter Exercises

- 1) In your own words, explain which is better (or if one is not better) RISC or CISC architectures.
- 2) Briefly describe why system clock speeds in RISC architectures are typically faster than the clock speeds in CISC-based computers.
- 3) Briefly but completely explain why modern computer architectures often blur the accepted definitions of RISC and CISC architectures.
- 4) Briefly describe whether the RISC-V OTTER MCU is a RISC or CISC computer.
- 5) What is the alternative name for the Von Neumann architecture?
- 6) Briefly describe which standard computer architecture better supports the notion of *pipelining* and why.
- 7) Briefly describe why it is a good idea to discern the type of architecture the you're working with early when working with a new computer architecture.
- 8) Briefly describe whether the computer you're working on now has a Von Neumann or Harvard architecture.
- 9) If lower levels of memory are faster, briefly but completely explain why there is a need for higher levels of memory.
- 10) Briefly describe the drawbacks of simply increasing the register file size in a computer architecture simply because it's a *faster* memory because it is a lower-level of memory.
- 11) Briefly describe why the notion of *levels of memory* is not necessarily dependent upon the data access times for a given structured memory.
- 12) List the three typical characteristics of a *lower level of memory*.
- 13) What is a lower level of memory: a tape drive or a hard drive? Briefly explain.
- 14) In the RISC-V OTTER MCU, which module represents the lowest level of memory and briefly explain why.
- 15) All mechanical switches have bounce when activated. Briefly explain whether silicon switches have switch bounce also.
- 16) Briefly describe why debouncing switches and buttons is more important for systems utilizing MCUs.
- 17) Briefly describe at least two drawbacks to using a switch debouncer in your circuit. Include both hardware and firmware debouncer in your answer.
- 18) If you had a circuit with many switches but only one debouncer, briefly describe the conditions where you could effectively debounce each of the switches using one debouncer.
- 19) Briefly describe how digital one-shots provide synchronization to the output of the one-shot.
- 20) What is another name for a mono-stable multivibrator?
- 21) What is another name for a bi-stable multivibrator?
- 22) What is another name for an a-stable multivibrator?
- 23) If you used both a debouncer and one-shot modules in your circuit, briefly describe if the order they appear in your circuit matters or not.
- 24) Briefly describe the two main reasons why we use multiplexed seven-segment displays.
- 25) Briefly describe the basic operation of a multiplexed display.
- 26) Briefly describe why in firmware multiplexing applications we speak of a ratio of time the display is on/off.
- 27) Briefly describes what happens when the 7-segment display on/off ratio is too small.
- 28) List the two reasons why we need to leave the display on for a given amount of time.

- 29) Seven-segment displays are useful for two main reasons, list those reasons.
 - 30) Write a closed form formula showing the number of inputs required to drive a seven-segment display for any number of digits two or greater.
 - 31) Briefly describe the two main reasons we use *lead zero blanking* in a seven-segment display.
 - 32) Briefly describe why we never blank all zeros in a seven-segment display if the number to be display is zero.
-

20 RISC-V MCU Timing Issues

20.1 Introduction

In order to truly understand the RISC-V MCU, you must have a solid grasp on various issues regarding the RISC-V hardware. Computer programming is an exercise in writing text that some software (assembler and/or compiler) translates into machine code that drives the computer. That is not the end of the story. All operations on a computer, such as executing instructions, require time to complete. Because the underlying computer hardware implements the timing and actions taken by each instruction, we have a tendency to concentrate on programming as independent entity. But the truth is that to become complete excellent hardware designers as well as great programmers, we must understand all aspects of how the underlying hardware executes instructions. One of those important aspects of instruction execution is the associated timing diagrams.

This chapter introduces timing characteristics of instructions. The bad news is that timing diagrams can potentially become really complex based on the number of *important* signals associated with a computer. But the good news is that there are a limited number of *types of instruction execution* we need to deal with. This of course means that there are many similarities between executing certain types of instructions, where these types generally follow the standard instruction types in the RISC-V OTTER MCU ISA.

This chapter takes a higher-level approach to timing diagrams in an effort to save time and space. We present example that contain a limited number of signals, but these signals are the more important signals associated with the execution of those given instructions. The signals we utilize fall are a classic set of both data and control signals, where some of the data signals include addresses.

Main Chapter Topics

- **RISC-V MCU TIMING ISSUE:** This chapter shows the execution of RISC-V MCU instructions as a function of time.
- **RISC-V MCU DATA & CONTROL TIMING:** This chapter shows a subset of the data changes and corresponding control signals in timing diagrams as a function of instruction execution.

Why This Chapter is Important

This chapter is important because it provides important insights into RISC-V MCU instruction execution by the use of timing diagrams.

20.2 RISC-V OTTER MCU Timing Problems

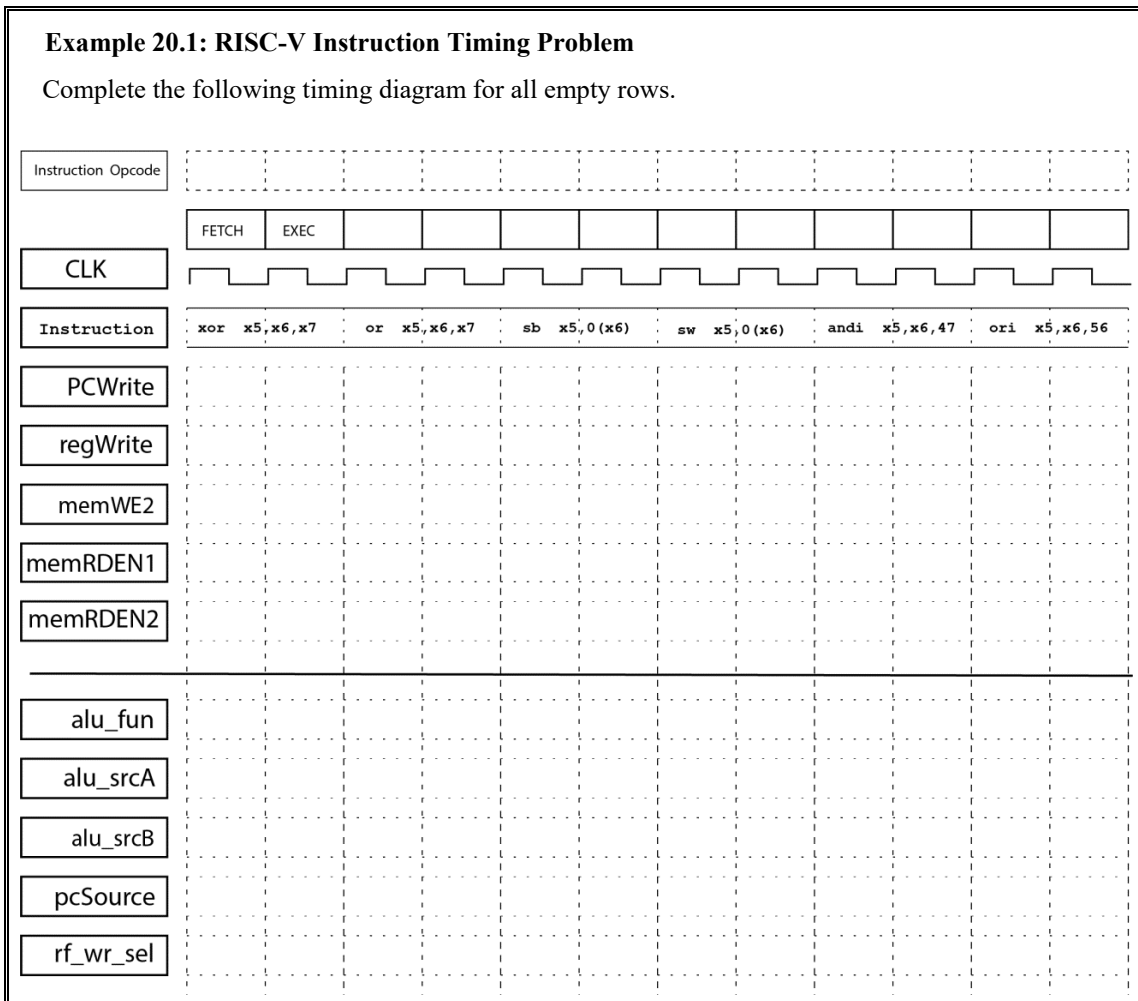
You can't fully understand the lower-level operations of the MCU unless you have a firm grasp on the underlying timing issues associated with the MCU instruction set. This section outlines the underlying timing issues by solving a few key timing problems associated with MCU instructions. The idea behind this section is to convince you that the operation of the MCU is fully deterministic and relatively simple once you fully understand all aspects of the MCU hardware and how the MCU instructions interact with that hardware.

We covered some the basic RISC-V MCU timing issues in previous chapters. In chapter, we take a more detailed look at some of the more important operations such as basic instruction execution timing, and subroutine calls and returns. The RISC-V MCU has many internal signals, so many that it would be near impossible to complete

a timing diagram with all the internal signals. Because of this, we limit our with timing diagrams in this chapter to a small set of meaningful data and control signals associated with instruction execution.

20.2.1 Modeling Instructions Using Timing Diagrams

These problems are rather unique compared to other problems we've worked with thus far. With programming-type problems, we knew there were many approaches to find functionally equivalent solutions. The ultimate goal was to write code that solved the problem, and there were essentially an infinite number of ways to do that. There were many ways to arrive at the solution and there were many solutions (although the solutions were necessarily functionally equivalent). With timing diagram problems, there are typically many approaches to arrive at the solution, but there is only one solution. Another way to state this is that operation of the RISC-V MCU is *deterministic*, which roughly means everything is 100% predictable and nothing is left to chance. We base this determinism on the RISC-V instructions and the underlying hardware that implements the instructions.



Solution: Figure 20.1 shows the solution to Example 20.1. These timing diagram solutions are hard to describe after the fact, but we'll do our best here. The best way to understand these problems is to watch them being done in class or in a video. With any luck, you'll have one of those available. Here's the skinny:

- Because these instructions do not include program flow control instructions, we are not constrained to completing one instruction at a time and instead can complete one row at a time in many instances.

- The **Instructional Opcode** line shows the opcode associated with each instruction below it. Each instruction includes a fetch and execute cycle as the line below the Instruction Opcode indicates.
- The CLK signal delineates the fetch and execute cycles, which the underlying FSM controls.
- The Instruction line shows the instruction that the MCU is executing at any given time.
- The **PCWrite** line is responsible to loading new values into the PC. It asserted soon after the start of each execute cycle; the actual loading of the new PC occurs at the next rising clock edge. The FSM asserts the PCWrite control signal after entry to the execute cycle; the loading of the PC occurs at the next active clock edge.
- The **regWrite** line controls writing to the register file. The first two and last two instructions write the result to the register file, which is why the **regWrite** signal asserted by the FSM soon after the start of the execute cycle for those four instructions. The two store-type instructions write values to memory and not the register file, which is why **regWrite** is not asserted in the execute state for the two store-type instructions.
- The **memWE2** line asserts in the execute cycle for the two store-type instructions. Asserting the **memWE2** signal allows the hardware to write the data in the register file to main memory.
- The **memRDEN1** signal asserts as part of the fetch cycle for every instruction. This allows the address that was loaded into the PC at the start of the fetch to serve as an index into main memory. The memory has synchronous reads so the output from the memory becomes available on the **DOUT2** after the rising clock edge that transitions the FSM from the fetch to execute state.
- We are not executing any load-type instructions, so the **memRDEN2** input remains unasserted throughout the timing diagram.
- The **alu_fun** chooses what operation the ALU performs. It performs XOR, OR, AND, and OR operations for the first two and last two instructions; the four bits shown in the timing diagram correspond to the operation chart on the RISC-V MCU schematic. The store-type instructions both require the ALU to do an addition operation as part of the address calculation for the instructions. This means that the **alu_fun** must choose an addition operation (0000) for all instructions requiring address calculations, which include all load-type and store-type instructions.
- The **alu_srcA** signal remains unasserted for these instructions, which means the instructions are always choosing **rs1** to as the **srcA** input of the ALU.
- The **alu_srcB** choose the **rs2** input for the logic instructions (00), and choose the S-type input (10) for the store-type instructions.
- The **pcSource** always chooses (00) for this example, which is the $PC \leftarrow PC + 4$ operation. This is normal operation, meaning that none of instructions are program flow control-type instructions.
- The **rf_wr_sel** chooses the output of the ALU for the four logic-type operations, which is the (00) input on the register file MUX. The two store-type instructions do not use the **rf_wr_sel** signal, which is why we list the **rf_wr_sel** value as don't cares for those instructions. There actually a value on the data output the register file MUX, but since we're not loading values into the register file, the actual values do not matter.

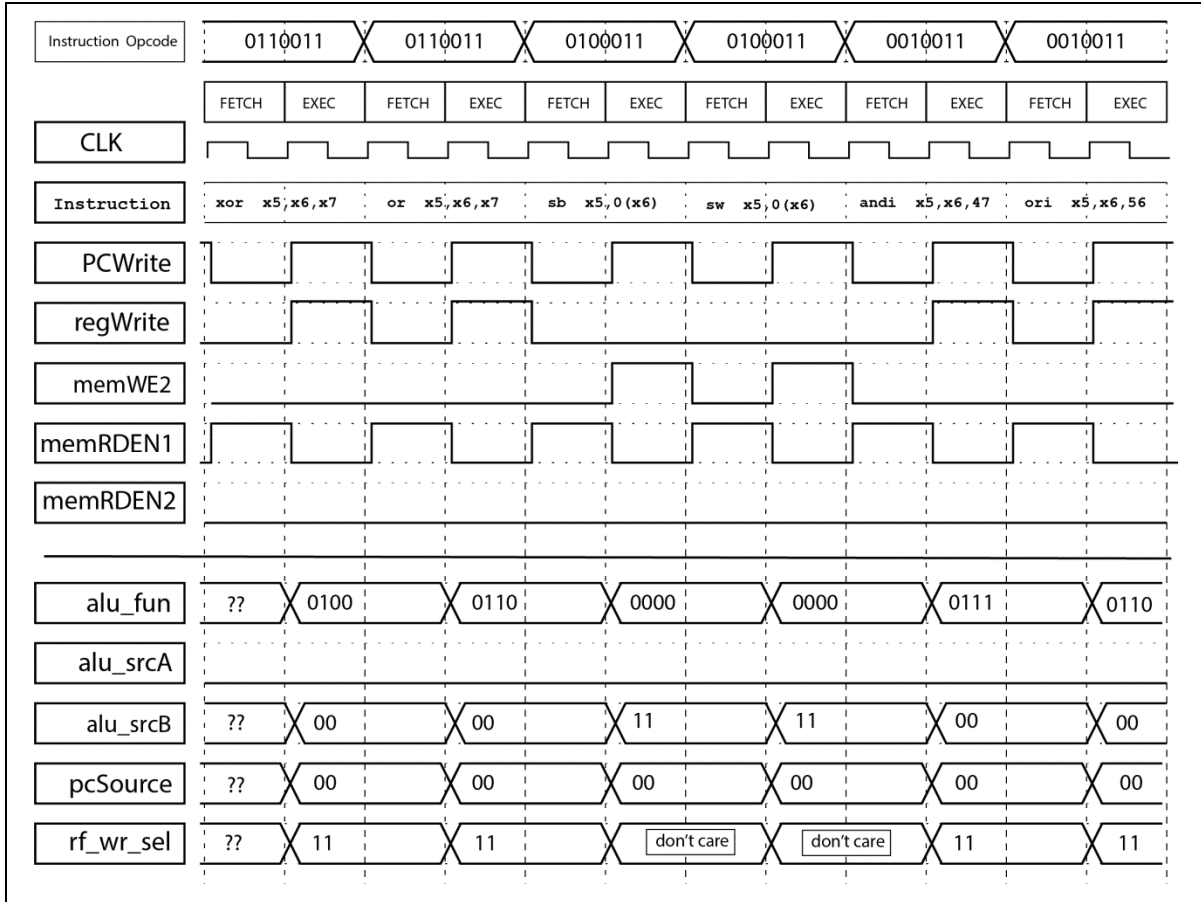
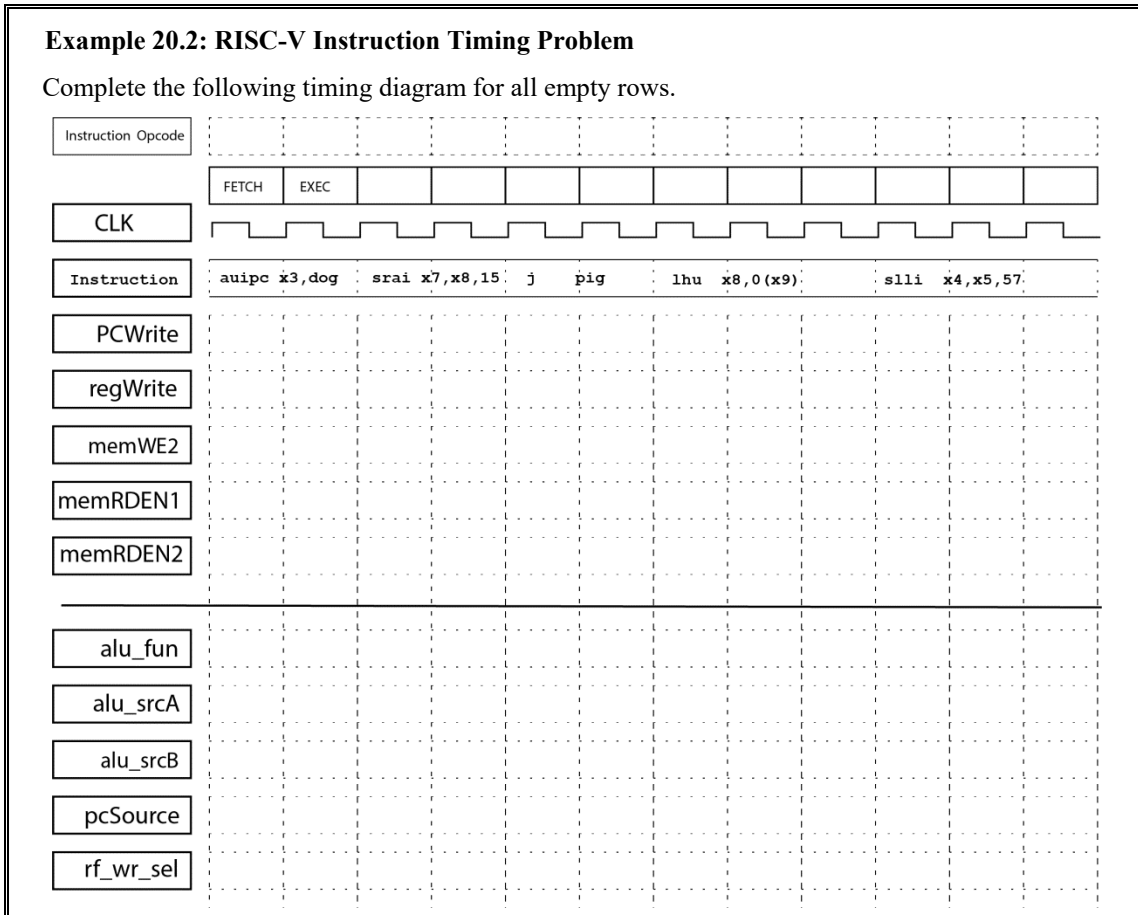


Figure 20.1: The solution to Example 20.1.



Solution: Figure 20.2 shows the solution to this example. Once again, and for the last time, these are much better explained in an incremental manner, rather than “here’s the complete diagram, let me try to explain it”. If you’re reading this (and I know you are), there is a video where I go through this one part at a time. Keep in mind that there are many approaches you can take to complete this problem. Here is the important stuff to know about this timing diagram:

- Most signals operate “normally”, though there is a load-type instruction in there that is different because it requires three clock cycles, this means that some of the control signal outputs are persistent through both the execute and writeback cycles.
- The **PCWrite** signal always asserts in the state before the fetch state of the next instruction. For the **lhu** instruction, this means **PCWrite** asserts as part of the writeback state. The **memRDEN1** signal shows a similar behavior in that it always asserts as part of the execute cycle; this execution is delayed with the three-cycle **lhu** instruction.
- The **lhu** instruction reads a value from memory and sticks it in a register, which requires the **memRDEN2** signal to assert during the execute cycle of the **lhu** instruction. At this point, the ALU has calculated the effective memory address and the hardware can read the memory data.
- None of the instructions write to data memory, so the **memWE2** signal remains unasserted.
- The **auipc** instruction and the **lhu** instruction both use the ALU to formulate values, which is why the **alu_fun** signal is configured for addition for both of these instructions.
- Because **j** is a pseudoinstruction, we know that it translates to a **jal** base instruction. Part of the **jal** instruction is to write the new PC value to a register, which is why the **regWrite** signal is

asserted during the execute cycle. Accordingly, the `rf_wr_sel` signal chooses the “PC+4” input to be loaded into the register file.

- Because the `j` instruction does not use the ALU, so the three ALU-controlling signals (`alu_fun`, `alu_srcA`, & `alu_srcB`) are “don’t cares” for this instruction.
- The `rf_wr_sel` signal selects the memory output for the `lhu` instruction. Even though the `rf_wr_sel` signal is asserted to “10” for three clock cycles associated with the `lhu` instruction, the only time `rf_wr_sel` matters is at the end of the writeback cycle, which is when the `regWrite` signal asserts.
- The `alu_srcB` signal typically follows the instruction type. For reg-reg instructions, the `alu_srcB` signal chooses the `rs2` register output, but for other instructions, it inputs either the PC (`auipc`), or the I-Type or S-Type immediate values. The `lhu` instruction is an I-Type instruction, which is why the `alu_srcB` signal is “10” for this instruction.

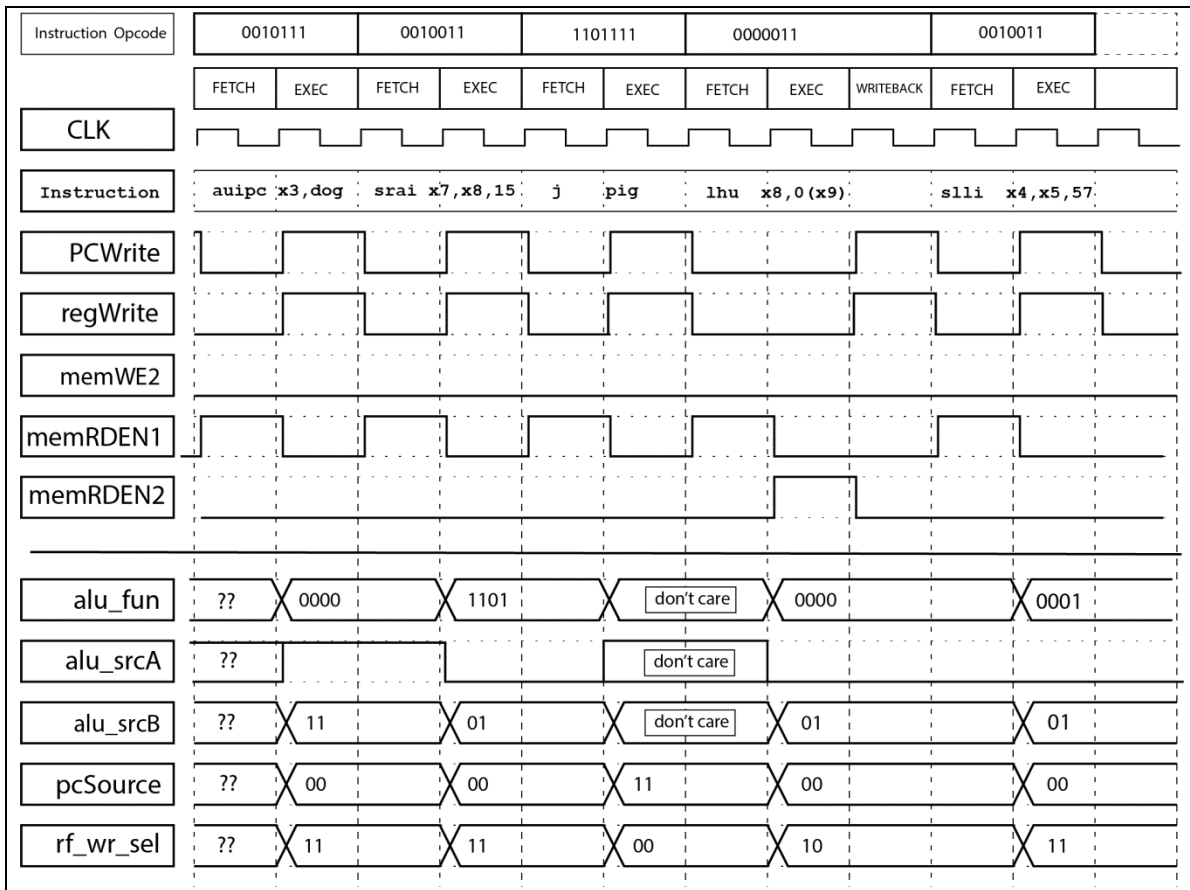
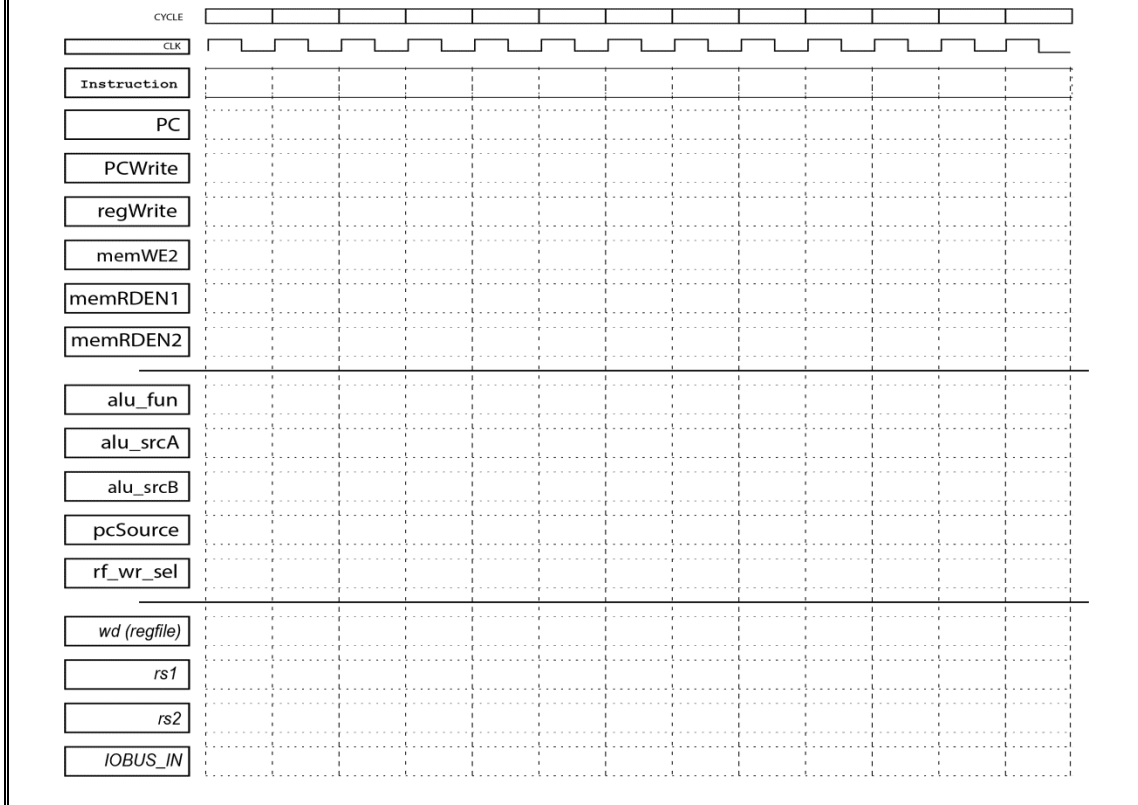


Figure 20.2: The solution to this example.

Example 20.3: RISC-V Instruction Timing Problem

Use the following program and information to complete the empty rows in the timing diagram.

Cat:	slt	x20, x10, x13	x10 = 0xAA
	jal	dog	x13 = 0xCC
	andi	x10, x8, 0xF	x11 = 0x11001100
dog:	lw	x8, 0(x11)	cat = 0x50
	bgt	x8, x13, done	IOBUS = 0x000000FF
	xor	x8, x10, x13	
done:	jalr	x1	
	andi	x8, x8, 0x0F	



Solution: Figure 20.3 shows the solution to this example. This solution for this timing diagram is slightly more complicated in that we don't know in advance if conditional branches are taken or not. This means we need to work through the solution on instruction at a time. We only are interested in the interesting signals below. Here is the cool stuff to know about this timing diagram:

- No instructions write memory, so the **WE2** is always unasserted.
- The first instruction is an **slt**. Because it writes a '1' or '0' to the destination register, **regWrite** asserts during the execute cycle. The **alu_fun** selects the "0010" option supporting the **slt** instruction as the value that the hardware assigns to the destination register from the ALU output. The **wd** signal indicates that the instruction writes a '1' to the destination register. The **rs1** & **rs2** lines are the source operands, which are the values in the x10 & x13 registers, respectively.
- The **jal** instruction causes a jump to the instruction associated with the "dog" label. The **pcSource** must choose the jal option to the PC MUX, which the **pcSource** signal indicates on the execute cycle with a value of "11". The **jal** instruction also writes a PC+4 value to the register file, which is why **regWrite** asserts and **rf_wr_sel** is set to "00" to route the value in the PC to the

register file. The value that writes to the register file is four greater than the current PC, which is 0x58, as indicated on the **wd** signal.

- The instruction at the “**dog**” label is an **lw**, the infamous three-cycle instruction. This causes a **memRDEN2** assertion during the execute cycle and **regWrite** assertion on the writeback cycle. The **rf_wr_sel** is set to “10” to allow the memory output to route to the register file. The data that writes to the register file is the **IOBUS_IN** data because the address in x11 is greater than 0x0000FFFF, making this an input operation and not a memory access operations (read). The **rs1** & **rs2** values are the base register and offset values for the load instruction, respectively. And finally, the actual input data writes to the register file, as the **wd** signal indicates.
- The **bgt** instruction takes the branch because the value in x8 is greater than the value in x13. Note that the x8 & x13 values are output from the register file (**rs1** & **rs2**, respectively). The ALU is not involved in comparisons, so ALU support signals are in “don’t care” space. The **pcSource** chooses the *branch* input to the PC MUX to be the next address loaded to the PC. The **pcSource** is a control output from the CU_DCDR, which acts on the comparison done in the BRANCH_COND_GEN module. The code takes the branch, which causes the value of the “done” label to load into the PC.
- The **jalr** instruction is similar to the **jal** instruction; it causes a jump by using the **pcSource** to load the **jalr** input to the PC MUX into the PC. Also like the **jal** instruction, the **jalr** instruction causes the PC+4 value (0x6C in this case) to load into the register file, which requires the **rf_wr_sel** to be “00” and the **regWrite** to assert. The **jalr** instruction uses a source register to calculate the absolute address, which is the value put in the **ra** as a result of the **jal** instruction; this value is 0x58, as the value of the **rs1** signal indicates.
- The final instruction performs an AND operation on the data in x8, which is trying to be some sort of masking operation. The data in x8 is 0xFF; ANDing it with 0x0F results in 0x0F, which is indicated on the **wd** signal for this instruction. The **alu_fun** chooses AND operation with a “0111”; **alu_srcB** chooses the lower ALU operand to be the I-Type immediate value from the IMMED_GEN module. The **rs2** line is an unknown because the immediate operand is output from the IMMED_GEN module and not the register file.

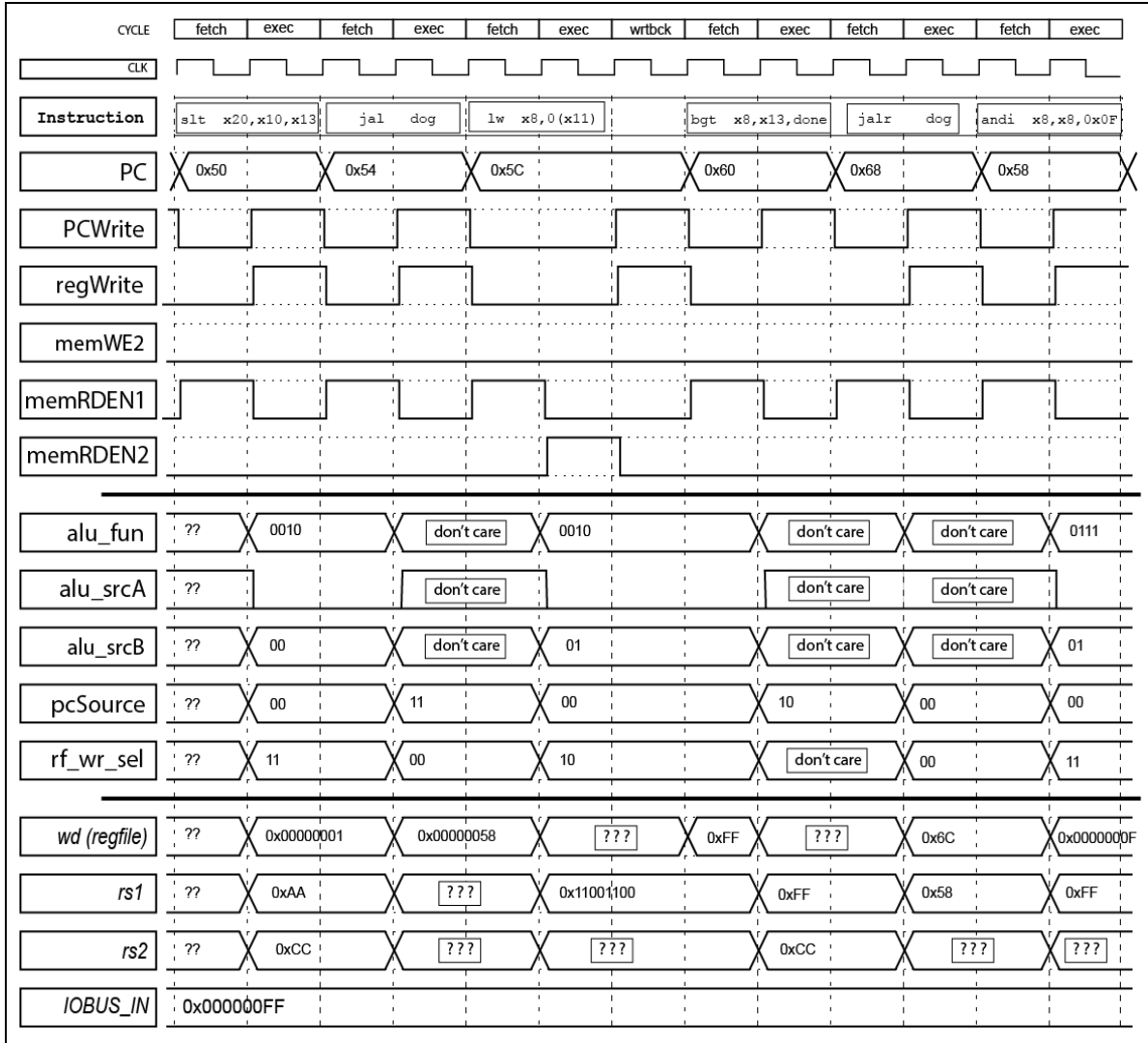


Figure 20.3: The solution to this example.

Example 20.4: RISC-V Instruction Timing Problem

Use the following program and information to complete the empty rows in the timing diagram. Use the empty timing diagram from the previous example problem.

cat:	addi	x10,x10,5	x30 = 0xC0
	beq	x10,x11,dog	x10 = x40
	add	x10,x0,x0	x11 = 0x45
dog:	sb	x11,4(x30)	cat = 0x80
	slli	x11,x11,1	
	jal	cat	

Solution: Figure 20.4 shows the solution to this example. This is another solution where we can't write out the instructions because the program includes program flow control instructions in the form of conditional branches;

and we don't know in advance if the program takes the branches. We only are interested in the signals that provide us with enlightenment in the discussion below. Here is the good stuff to know about this timing diagram:

- The program fragment has no input instructions so the **IOBUS_IN** has no use in this example.
- There are no instructions reading from data memory, so the **memRDEN2** signal never asserts.
- We know in advance what **memRDEN1** looks like because only one of the instructions is a load-type instruction.
- The **addi** instruction is at 0x80 (**cat** label), which the problem description provides. This instruction adds the value in x10 (**rs1**=0x40) to the I-Type immediate value (5) and stores the result in x10. Note that we don't know the **rs2** value as the immediate value is an output the **IMMED_GEN** module. The requires the **regWrite** asserted and the **alu_fun** to select an add operation. The result of the addition operation is 0x45 appears on the **wd** line. The **alu_srcA** signal chooses the register value and the **alu_srcB** signal chooses the immediate input to the MUX to be the input to the ALU. The ALU result writes to the register file thus requiring the **regWrite** to assert.
- The **beq** instruction compares two register value **rs1** & **rs2** as they appear on the **rs1** & **rs2** lines; both values are 0x45, which causes the instruction to take the branch. The instruction does not use the ALU so all ALU-based signals are don't cares. The **pcSource** chooses the branch option ("10"), which is an output from the **CU_DCDR**. The program jumps a whopping one instruction to 0x8C as the value on the **PC** line indicates.
- The **sb** instruction copies data from a register to memory, which requires the assertion of **memWE2** on the execute cycle. The ALU does an add operation to calculate the effective address with an offset of "4" (**rs2**) and a base register value of 0xC0 (x30 @ **rs1**). The **sb** is an S-Type instruction so the **alu_srcB** is set to "10" to have the appropriate immediate value input to the ALU. The **rs2** line is an unknown because the immediate value is output from the **IMMED_GEN** module. The instruction does not involve the register file other than reading the base register (**rs1**) so the register file related control signals are don't cares.
- The **slli** instruction is an I-Type instruction. The ALU inputs are "1" for the I-Type input and 0x45 for the other input (**rs1**). The shifted-left value appears on **wd** as 0x8A, which is the value that writes to the register file. The **alu_fun** chooses the "sll" option.
- The **jal** instruction causes program control to unconditionally transfer to the instruction associated with the **cat** label, which is at address 0x80. This instruction causes the **pcSource** to be "11". This instruction also writes the "PC+4" (0x98) address to x1, which requires the **regWrite** signal to assert on the execute cycle. Recall that the "PC+4" is a potential return address as programmers can use the **jal** instruction to call subroutines. The ALU is not used so associated signals are don't cares.
- The **addi** instruction is a repeat of the first instruction, so nothing new happens. The value in x10 is now 0x45 because the first **addi** instruction advanced it by five. The **wd** line shows the result to that writes back to the register file.

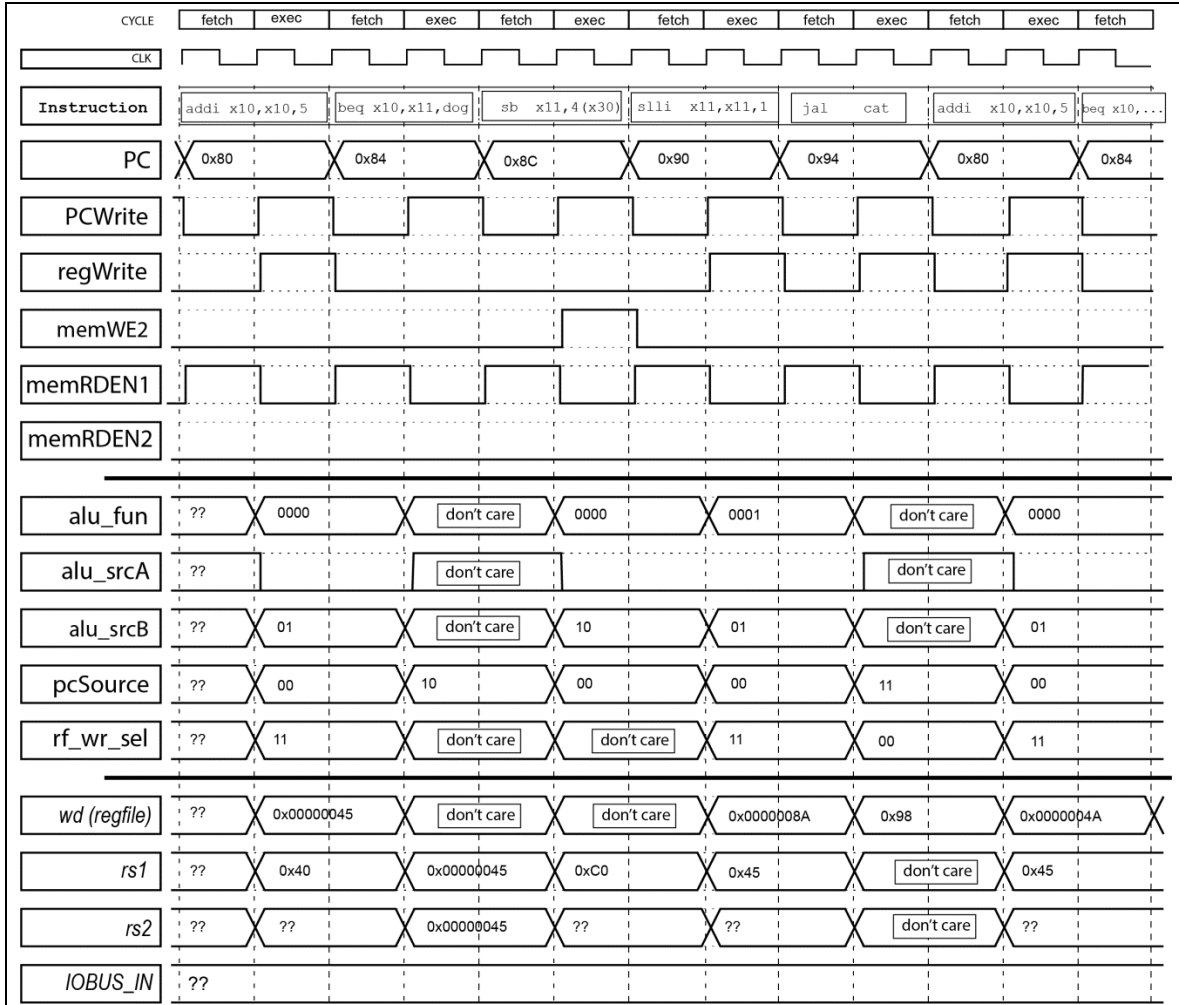


Figure 20.4: The solution to this example.

Example 20.5: RISC-V Instruction Timing Problem

Use the following program and information to complete the empty rows in the timing diagram. Use the empty timing diagram from the previous example problem.

<pre> Cat: addi sp, sp, -4 sw x8, 0(sp) xor x8, x8, x9 hen: beq x8, x0, dog slti x7, x8, 0x34 xori x8, x10, 0x40 andi x8, x8, 0x01 dog: lw x8, 0(sp) addi sp, sp, 4 </pre>	<pre> # cat = 0xB0 # x2 = 0xF08 # x8 = 0xAA # x9 = 0x55 </pre>
--	--

Solution: Figure 20.5 shows the solution to this example. This is another solution where we can't write out the instructions as a first step in the solution because the program fragment includes program flow control

instructions in the form of conditional branches. We only are interested in the interesting signals and happenings in the solution, which we liberally describe below:

- We don't use the **IOBUS_IN** signal because the program fragment has no input instructions.
- This example basically shows the pushing and popping of a single register on the stack. Both operations require a memory access and an adjustment of the stack pointer.
- The problem description provides the **addi** instruction address, which is 0xB0 (**cat** label). This instruction adjusts the stack pointer, which it needs to do to make room for the register that the next instruction stores. The stack pointer is at address 0xF08, which the problem description provides. The **rs1** value is one operand; the other operand is the output from the **IMMED_GEN** module, which is -4 (not shown in diagram). The result is loaded into the register file, which the 0xF04 on the **wd** line shows.
- The **sw** instruction asserts the **memWE2** signal on the execute cycle to write x8 to memory. The instruction uses the ALU to create an effective address by adding the offset (0) to the base register (**sp**); note the address value on rs1 is 0xF04, which was the value written to **sp** by the previous instruction. This instruction does not use the register file, which is why the **wd** line shows a "???".
- The **xor** instruction performs an exclusive OR on the data in the x8 & x9 registers and stores the result in x8. We see the result on the **wd** line, which we conveniently designed to be zero when we created this problem.
- The branch instruction compares the value in x8 & x0, which are equal, so the code takes branch. The code takes the branch by setting the **pcSource** to "10", which chooses the branch address to be loaded into the **PC**. This branch causes a jump to the instruction with the "**dog**" label, which is at address 0xCC, as the **PC** value associated with the **lw** instruction indicates. The **regWrite** signal does not assert, as branch instructions don't involve the register file.
- This **lw** is the first instruction in the pop operation. The **lw** is a three-cycle instruction that formulates the memory address during the execute cycle (the **memRDEN2** signal) and write that value to the register during the writeback cycle (the **regWrite** signal). The **rf_wr_sel** signal is set to "10" to select the memory output (DOUT2) to be input the register file. The effective address is the offset (0, not shown) plus the value in **sp** (x1=0xF04). The **rs2** value is not known as the offset is an output of the **IMMED_GEN** module, which **alu_srcB** chooses ("01") to input to the ALU. The **alu_fun** selects "0000" to perform the addition that generates the effective address. The value written to the register file is on the **wd** signal; 0xAA is the original value of x8 before the **sw** instruction operated on it. Note that for the **lw** instruction, the data that writes to the register file is not available until the effect address calculation completes and the **memRDEN2** signal enables the reading of data. This timing is important, which is why we littered the diagram with the arrow.
- The **addi** instruction adjusts the stack pointer, and is thus the second half of the pop operation. The first instruction in the program fragment decreased the **sp**; this **addi** instruction returns it to its original value. The **rs1** signal shows the current **sp** value of 0xF04; the **wd** signal shows the new value for **sp**, which is 0xF08.

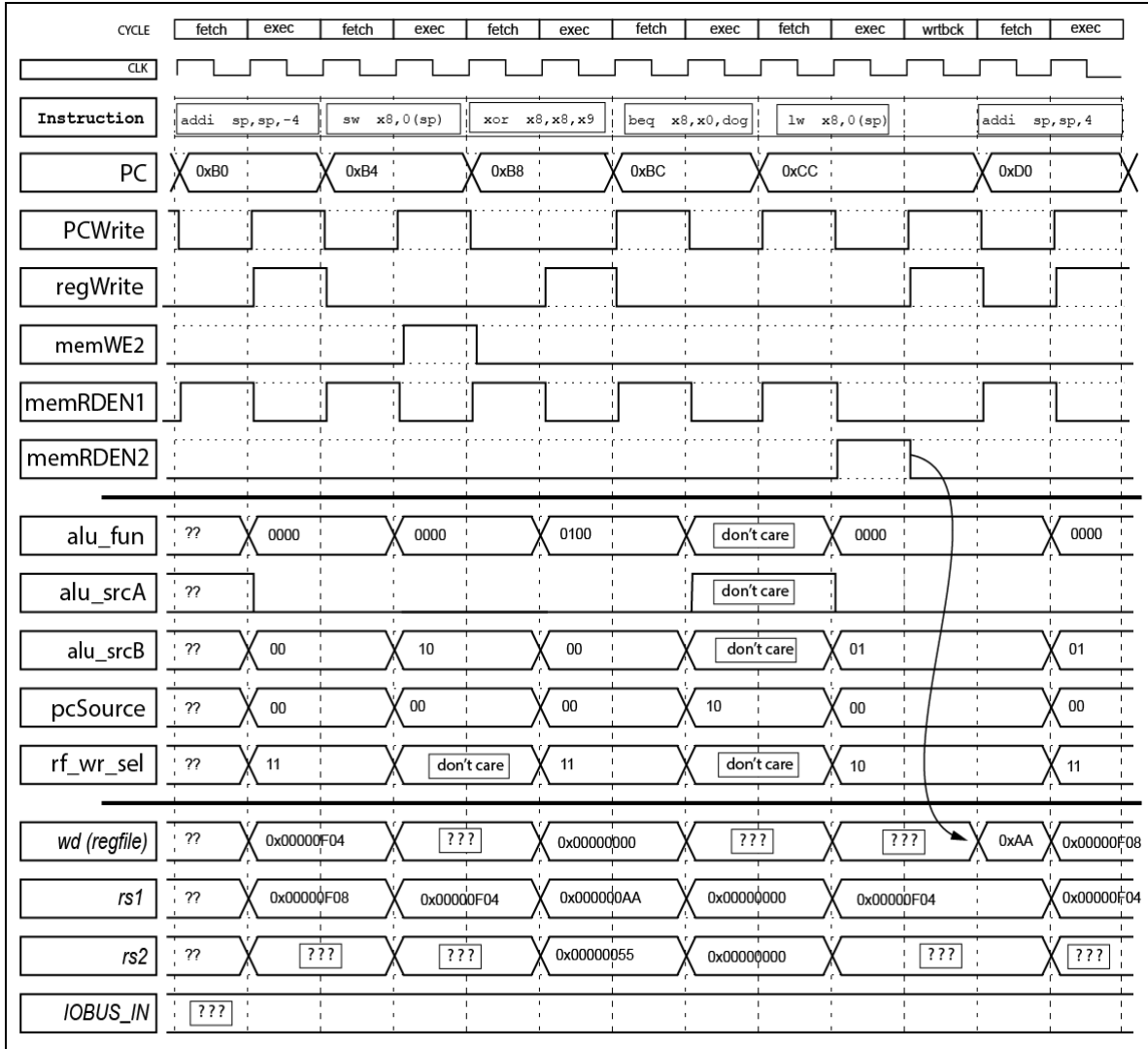


Figure 20.5: The solution to this example.

20.3 Chapter Summary

- Computers do what they do over a given period of time, which allows us to model program execution units, such as instructions, as a function of time. The standard approach to viewing digital signals as a function of time is the timing diagram.
 - All RISC-V instructions require two clock cycles to execute, except for load-type instructions, which required three clock cycles to execute.
 - Most program flow is conditional, meaning that it is controlled by some type of conditional branch instruction. This means that we must complete timing diagrams using a vertical analysis going from left to right and completing all signals. We can complete timing diagrams associated with unconditional branches or no branches one signal at a time going from left to right. Calling subroutines is an example of unconditional program flow control.
-

20.4 Chapter Exercises

- 1) Briefly describe why the timing diagrams in the problems examined in this chapter only contained an abbreviated set of control signals.
 - 2) Briefly describe what the term “timing diagram” refers to.
 - 3) Briefly describe why there is only one unique solution to timing diagram problems associated with the RISC-V MCU.
 - 4) Briefly describe what condition in a timing diagram problem allows us to complete entire signals at one time rather than one instruction at time.
 - 5) Briefly describe the main difference in analyzing problems containing conditional branch instructions compared to problems not containing conditional branch instructions.
 - 6) Briefly describe why we typically slightly delay the switching of control signals in timing diagrams. This delay manifests itself as a slight delay past the active clock edge in the timing diagram.
-

21 RISC-V Architectural Modifications

21.1 Introduction

As you extend your knowledge regarding computer architecture and assembly language programming, you'll no doubt start questioning some of the design decisions that went into design the RISC-V MCU ISA and associated hardware. In truth, a countless number of design decisions went into the design. The main thought here is that these design decisions, though well thought out, were somewhat arbitrary all the same. As with any digital design, if you stare at it long enough, you'll surely figure out a better approach to the design. The same goes for the RISC-V OTTER MCU instruction set. If you haven't found yourself complaining that there is a useful/important instruction missing from the instruction set, you're probably missing something.

This chapter allows you to apply your knowledge and skills by asking you to describe various changes to the RISC-V MCU hardware and/or instruction set. The idea here is that if you can't describe viable changes to the RISC-V MCU architecture and instruction set, you may not have a strong understanding of the MCU in general. In other words, to make meaningful changes to the RISC-V, you must understand all aspects of the RISC-V MCU, particularly how the hardware implements instructions in the RISC-V MCU. If you're not quite at that point yet, the examples in this chapter quickly move you along in the direction of complete understanding of the RISC-V MCU. Recall that this course provided you with a schematic of a working RISC-V MCU, so there was no hardware design involved. But this course in general is about hardware and assembly language programming, so it makes more sense to divide the time between hardware and firmware rather than moving to strictly firmware after you've implemented the RISC-V MCU.

Computer design continues to be an open book. While there is a significant amount of commonality between various computer architectures, they are still completely arbitrary. If you're a computer user, you need to understand the hardware provided for you. If you're a computer designer, you have the ability to design any computer you want.

Main Chapter Topics

- **RISC-V HARDWARE ARCHITECTURE MODIFICATIONS:** This chapter outlines hardware modifications in order to achieve various stated design goals.
- **RISC-V ASSEMBLER MODIFICATIONS:** This chapter outlines changes to the RISC-V assembler in the context of desired hardware architecture modifications.
- **RISC-V INSTRUCTION SET ARCHITECTURE MODIFICATIONS:** This chapter outlines changes in the instruction sets in response to proposed RISC-V hardware architecture changes.

Why This Chapter is Important

This chapter is important because it advances your knowledge of the RISC-V MCU by outlining hardware architecture changes in response to stated design goals.

21.2 RISC-V Architectural Modifications and Extensions

The chapter comprises of suggested modifications and changes to the RISC-V hardware architecture and/or instruction set. These problems represent my best take on these proposed changes; I have no doubt that I have mistakenly omitted important information in these examples and/or you can think of a better solution than the

ones I'll present here. That's good. These problems are quite open-ended, meaning there are many correct solutions. You can argue that one solution is better than another solution, but that's not primarily what we're after here. If you discover something that I did not see, then you definitely know the RISC-V MCU architecture and instruction set, which is the underlying goal of this text and associated course.

Once again, these problems are open-ended. The only general guideline to follow is that you should always try to find the most "doable" solution. In the context of these problems, doable means you can describe what you're going to do without having to massively increase the complexity of the RISC-V MCU hardware. One other guideline is the notion of "adding" or "removing" functionality from the RISC-V hardware or instruction set. Whenever you "add" something, you're most likely increasing the complexity of the hardware include the amount of memory required to support that addition. Conversely, if you remove something, you're probably reducing complexity and possibly removing memory requirements.

In reality, you must consider each addition or removal individually in order to ascertain its full ramifications. For these types of problems, know that you're going to need to pull out the assembler manual and architectural diagrams in order to help you generate a viable solution. In the real world, you must consult various sources when you're working on problems; it's strange why academia needs to be different.

When writing your solutions, be as descriptively complete as possible. Humans grading these problems need to know what you know. If something requires modification, then be specific about the required modifications; you certainly can't be too descriptive in this area. If you say something like "there are no changes required", be specific about what is not changing and why that thing does not need to change. Note that problems such as these that appear on an exam or quiz is going to require you to do an organized brain dump so the wacky instructor can ascertain what you know or don't know. The instructor won't be giving you the benefit of the doubt on these types of problems.

Example 21.1: Shift-Set Instructions

Add a two new instructions to the RISC-V MCU:

```
sll_1 rd,rs1 # x[rd] ← { x[1'b1,rs1[30:0]] }
srl_1 rd,rs1 # x[rd] ← { x[rs1[31:1]],1'b1 }
```

For this problem, describe the following

- Required changes to the RISC-V MCU hardware
- Required changes the RISC-V assembler
- Required changes in RISC-V memory requirements
- Describe why these instructions are potentially useful
- Describe why you don't need a `sll_0` & `srl_0`

Solution: There is nothing magic about this solution; it is somewhat arbitrary. I feel it's about as simple as I can think of. The first thing to notice about this problem is that it requests that you add something to the RISC-V MCU, which means we want to stay on the lookout for "things increasing", such as memory and/or the width of MUX select lines, width of data lines, etc. Note that unlike the current shift instructions, these two new instructions are single shifts and not barrel shifts.

a) Required changes to the RISC-V MCU hardware:

- You would need to modify the ALU to recognize these new instructions. The `alu_fun` signal would not need to change width as there is room for five more instructions; these two instructions require only two more ALU choices. This also means that you can implement this instruction with only the ALU, and there is no need to modify the lower ALU MUX to include a '1' input.

- You would need to modify the CU_FSM to include a new instruction type. This instruction would be similar to an I-type instruction as far as operands go. There are currently six I-type instructions that use the “0010011” opcode, so you can reuse this opcode for these instructions; in this case you would need to change the **funct3** opcode value for these instructions to the two **funct3** opcodes that are not currently included in the six ALU-oriented immediate instructions, which are “001” & “101” (check the spec).
 - You would need to modify the CU_DCDR to account for these instructions so it can send out the correct **alu_fun** signals in the case of ALU-based immediate instructions. The CU_FSM requires no changes because we were able to make this a true I-type instruction by reusing the immediate-type opcode and only tweaking the **funct3** opcode.
- b) Required changes the RISC-V assembler:
- You would need to make the assembler aware of the new instruction including the three new **funct3** opcodes from the previous step.
- c) Required changes in RISC-V memory requirements
- You did not add states, change main memory, change the CSR, or change the PC, so there would be no changes in memory sizes.
- d) Describe why these instructions are potentially useful
- If your application needs to shift 1’s into register rather than 0’s, this instruction would save an instruction or a register (depending on whether you would implement the shift as an immediate of reg-type instruction).
- e) Describe why you don’t need a **sll_0** & **sr1_0**
- The current shift left and right instructions insert 0’s already, which means a shift-type instruction that shifts one bit location does the same thing.
-

Example 21.2: Reg-Reg Load Instructions

You must modify the RISC-V OTTER to include three new instructions:

```

lb    rd,rs2(rs1)    # load rd with data at mem addr M[rs2+rs1]
lh    rd,rs2(rs1)    # load rd with data at mem addr M[rs2+rs1]
lw    rd,rs2(rs1)    # load rd with data at mem addr M[rs2+rs1]

```

The **lb** and **lh** instruction should zero-extend data such that it fills the 32-bit destination register.

For this problem, describe the following:

- changes you need to make to the RISC-V MCU hardware
- changes you need to make to the RISC-V MCU assembler
- changes in RISC-V MCU memory requirements
- why this modification would or could be useful

Solution: This problem asks you to add something to the MCU, which means there is a possibility that the width of some items such data widths or memory may increase. We don't know anything for sure yet, but we'll be on the lookout for stuff "growing".

a) Required changes to the RISC-V MCU hardware:

- The first thing to notice about these instructions is that they have the same mnemonic as existing RISC-V MCU instructions. This is possible and programmers do it quite often. The trick here is even though the instructions are the same, the form of the operands is different, which thus allows the assembler to differentiate between the instruction types. So no need to worry there. We do then hope to make these I-type instructions similar to the other load-type instructions. There are currently five load-type instructions that share the "0000011" opcode; these instructions are differentiated by the **funct3** opcode field. Since there are only five instructions, and the 3-bit **funct3** opcode has three unused bit combinations, we can easily fit this instruction with the other instructions having the "0000011" opcode flavor.
- We next need to examine the control units since we're adding new instructions. The instructions as similar to other load-type instructions in that they provide an absolute address to the memory module; these instructions only differ in the way the instruction calculates the absolute address. This instruction chooses **rs2** for the second operand rather than an immediate value, which means the only difference between a "**lw rd,imm(rs1)**" and "**lw rd,rs2(rs1)**" is the data selection on the ALU's **srcB** MUX. We thus need to modify the **CU_DCDR** to recognize these instructions and send out the correct **alu_srcB** signal. We don't need to change the **CU_FSM** because the present support for the load-type signals works fine for our new instruction.

b) Required changes the RISC-V assembler:

- We would need to make the assembler aware of the three new instructions including whatever new opcodes you decided from the previous step.

c) Required changes in RISC-V memory requirements:

- We did not add states, change main memory, change the CSR, or change the PC, so there would be no changes in memory sizes.

d) Why this modification would or could be useful:

- This instruction provides another memory addressing mode. Now this may not excite you the programmer and hardware person, but it make compiler writers slather at the mouth as they now have more options as to implementing higher-level language code. In reality, the previous address calculation used one register and one immediate value, while these two new instructions use two register values. In this context, registers are effectively variable, which renders these load-type instructions potentially more useful than the current RISC-V load-type instructions.

Example 21.3: Branch Based on Memory Data

Add the following instructions to the RISC-V MCU. These are conditional branches based on a comparison of a bit set in a register and a memory address.

```

bm_eq  rs1,rs2,label  # branch to label if rs1 = mem[rs2]
bm_ge  rs1,rs2,label  # branch to label if rs1 ≥ mem[rs2]
bm_geu rs1,rs2,label  # branch to label if rs1 ≥u mem[rs2]
bm_lt  rs1,rs2,label  # branch to label if rs1 < mem[rs2]
bm_ltu rs1,rs2,label  # branch to label if rs1 <u mem[rs2]
bm_ne  rs1,rs2,label  # branch to label if rs1 ≠ mem[rs2]

```

```

example:    bm_eq    x10,x11,My_label

```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful
- why you would or would not include these instructions in the computer you're designing

Solution: This is yet another branch-type instruction. Here's what I see:

a) Changes you need to make to the RISC-V hardware:

- There are six instructions here, which seem like quite a few. What we want to do is use as much as the existing hardware as possible to implement these instructions. These looked like the regular branch instructions, so that is a good starting point for these problems. The `BRANCH_COND_GEN` module currently has two inputs, which are the two outputs from the register file. We can use one of those outputs, but the other output needs to come from the memory module. But since we need to support the regular branch instructions as well, the `rs2` input to the `BRANCH_COND_GEN` needs a MUX in front of the `rs2` input. This means the `CU_DCDR` needs another select output. The other input to the MUX connects to the `DOUT2` output of memory.
- We need to be able to use the register file's `rs2` output as an address to memory, which means we need a MUX to choose between the current input (the ALU output) and `rs2`. This requires that we add another select signal to the output of the `CU_DCDR`.
- Our next concern involves timing. These new instructions are somewhat like load-type instructions in that the first needs to get a memory address, and then need to do something with that address. The load-type instructions require three states because of the synchronous read nature of the memory. Therefore, these new types of instruction are going to require three states.

- We don't need to change the BRANCH_COND_GEN module in any way, which simplifies the changes we need to make to the CU_DCDR.
 - We can make these new instructions have the B-type format, since it already contains the required fields. We need to have it a new 7-bit opcode and differentiate the instructions using the **funct3** opcode field.
 - We need to modify the CU_DCDR to recognize these six instructions. Additionally, we need to add a control signals for the two MUXes we added to the MCU. One MUX chooses between the **rs2** output or the memory output for the lower input of the BRANCH_COND_GEN; the other MUX chooses between the **rs2** output of the register file or the ALU result output to act as the address to data memory. We'll need to modify all load and store-type instructions to use the memory address MUX select; we'll need to do the same thing with the regular branch-type instructions, but with the other new MUX.
 - We need to change the CU_FSM to recognize these instructions. Additionally, we can get either add a new state or make the current third state (the writeback) state more complex by supporting these new instruction. We'll shoot for the more complex FSM to make the problem more interesting. This means we'll add a special state for these new branch-type instructions, which is similar to the writeback state.
- b) Changes you need to make to the RISC-V Assembler:
- The assembler needs to recognize the new instructions. We created a new opcode for a B-type instruction, and added a supporting set of **funct3** opcodes.
- c) Required changes in RISC-V memory requirements:
- Because we added a new "writeback" state especially for these new instructions, we increased the number of FSM states from four to five. Assuming we use the minimum number of bits to encode the FSM states, we then need to add another bit to the state registers, which means the state registers grow from two to three bits. We did not change main memory, the PC, the register file, or the CSR register, so their memory requirements do not change.
- d) Why this modification would be useful:
- This modification would be useful to save instructions by accessing memory directly for branch instructions. If we did not have this instruction, we would need to do get the data from memory before executing the branch instruction.
- e) Why you would or would not include this instruction in the computer you're designing:
- If your application required that you do many branches based on the values in memory, these instructions would certainly save clock cycles. Once again, there are tradeoffs involved. These instructions require three clock cycles, so the saving are not overly significant. On other hand, the comparisons can be made without bring memory data to registers, so it does reduce register usage.
-

Example 21.4: Bit Set and Bit Clear Instructions

Add the following instructions to the RISC-V MCU. These are conditional branches based on a bit set in a register.

```
bbset  rs1,rs2,label # branch to label if rs1[rs2] = 1
bbclr  rs1,rs2,label # branch to label if rs1[rs2] = 0
```

```
example:  bbclr    x10,x11,My_label
```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful
- why you would or would not include this instruction in the computer you're designing

Solution: The following is my take on this problem; my solution is has no magic associated with it. Please let me know if you have better ideas. This is another example where we are adding something, so we'll stay attentive to parts of the architecture that require expanding.

a) Changes you need to make to the RISC-V hardware:

- These new instructions look just like branch instructions, meaning we can use the B-type format for these new instructions, which is good because we won't need to define a new instruction format. There are currently six B-type instructions using the "1100011" format and differentiated by the 3-bit **funct3** opcode, so there are two unused **funct3** opcodes we can use for these two instructions.
- We first need to modify the BRANCH_COND_GEN module to include more functionality. We don't need more inputs as we have both registers already input. We need to add one output, which would indicate whether a specific bit in a specific register was set or not, such as **br_bb**, similar to the other outputs from the modules. The hardware that we add to the BRANCH_COND_GEN module would drive the output to complete the given functionality.
- We then need to modify the CU_DCDR to include these two new branch instructions, which would entail adding another input bit to the module to handle the new output from the BRANCH_COND_GEN module.
- We can use the CU_FSM as is because we did not add a new OPCODE. This module will not need to change.

b) Changes you need to make to the RISC-V Assembler:

- The assembler needs to recognize the new instructions. We didn't change the B-type OPCODE, but we did add two **funct3** opcodes; the assembler needs to know about these.

c) Required changes in RISC-V memory requirements:

- The memory size does not change. We did not change main memory, the PC, the register file, or the CSR register. We also did not change the CU_FSM, which is where the last bits of memory reside.

d) Why this modification would be useful:

- This modification would be useful to save instructions. If we did not have this instruction, we would need to do masking and shift and other voodoo that makes adds extra instructions.

e) Why you would or would not include this instruction in the computer you're designing:

- The RISC-V designers probably didn't add this instruction because it is somewhat specialized, and programmers would not use it a relative large amount of time in normal coding. However, if the application you're designing your computer for could or would find this instruction useful, you would add it as it would have several instructions if you did not have it available. The again, implementing this instruction does take up much chip real estate, so you would only add it if it truly provided you with some meaningful benefit.

Example 21.5: Reg-Reg Load Instructions

Add the following instructions to the RISC-V MCU. Make as few modifications to hardware as possible. These are rotate left and right instructions based on words, two halfwords, or four bytes. For halfwords, the instructions perform rotates on the two individual halfwords in a register; for bytes the instructions perform rotates on each of the bytes in the register. These are similar to the shifts in that they are barrel rotates based on the lower bits of the immed value.

```
rolw rd,rs1,imm # rotate word left by imm val; store result in rd
rorw rd,rs1,imm # rotate word right by imm val; store result in rd
rolh rd,rs1,imm # rotate 2 halfword left by imm val; store result in rd
rorh rd,rs1,imm # rotate 2 halfword right by imm val; store result in rd
rolb rd,rs1,imm # rotate 4 bytes left by imm val; store result in rd
rorb rd,rs1,imm # rotate 4 bytes right by imm val; store result in rd
# example:    rolw x4,x5,4    # barrel left rotate v positions
```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful
- why you would or would not include this instruction in the computer you're designing

Solution: This problem asks you to implement six instructions, all of which represent some type of left or right rotates. We are adding something to the current RISC-V MCU, so we want to be aware of possible changes in memory requirements.

a) changes you need to make to the RISC-V hardware:

- These six instructions look like just like immediate instructions so we'll encode these as I-types. We don't have enough code space to encode these instructions using the current I-type instruction opcodes, so we'll give these instructions a new 7-bit opcode (it does not matter what exactly that is so long as it is unique). We'll then differentiate these instructions by assigning the each of the instructions a different **funct3** opcode, which works because there is space for eight instructions in the 3-bit **funct3** opcode space.
- The ALU does all the work for these new instructions, so we need to modify the ALU to support these instructions. There is currently only enough room for five more instructions in the ALU based on the width of the **alu_fun** signal, so we need to increase the bit-width of the **alu_fun** signal from four to five bits.

- We need to modify the CU_DCDR so that it recognizes these instructions with their sporty new opcodes and associated **funct3** codes. We also need to extend the width of the alu_fun output from four to five bits. This changes all the other 11 **alu_fun** values to account for the six extra instructions.
 - We need to make the CU_FSM recognize our new opcode for these instructions. They are I-type instructions, and the same as other I-type instructions, but with a different opcode. Recall that the CU_FSM does not use the **funct3** opcode.
- b) changes you need to make to the RISC-V assembler
- We need to make the assembler aware of the new instructions and send out the correct opcodes and **funct3** codes.
- c) changes in RISC-V MCU memory requirements
- We did not change main memory, the PC, the CSR, the reg file, or add new states to the FSM; so these changes do not cause the memory size to change.
- d) why this modification would be useful
- Rotates are always nice to have in the instruction set. You don't always have a use for them, but they really save time and effort when you do. Wimpy answer indeed.
- e) why you would or would not include this instruction in the computer you're designing:
- These are six instructions, which are significant, but also the hardware implementation requires a non-trivial amount of hardware. Therefore, you must really have a specific use for these instructions or it may be a better choice to use the instructions currently available in the RISC-V and take a small hit on execution time and program space.

Example 21.6: Push Instruction

You want to add the following instruction to the RISC-V MCU. Make as few modifications to hardware as possible. Implement this instruction in two clock cycles. Note that this instruction assumes the stack pointer is always x2.

```
push    rs2    # push rs2 on stack; assume stack pointer is x2
```

examples:

```
push    x25    # push x25 on the stack
```

For this problem, describe the following:

- a) changes you need to make to the RISC-V hardware
- b) changes you need to make to the RISC-V assembler
- c) changes in RISC-V MCU memory requirements
- d) why this modification would be useful

Solution: This problem is a bit challenging, but we included it because it shows you the possibilities and accompanying thought process you need to take on when you do these types of problems. The first thing to be aware of is that we're adding something to the RISC-V MCU, which means we may increase memory requirements.

- a) Changes you need to make to the RISC-V hardware:

- Push operations are generally two things: write something to memory at the correct address and change the address of the stack pointer. Write it out:

$x2 \leftarrow x2 - 4; \quad \text{mem}[x2 - 4] \leftarrow rs2 \quad \# \text{ these operations happen simultaneously}$

- Writing it out shows that we need to access two registers at the same time, which is the data in **rs2**, and the SP address in **x2**. There is only one type of instruction outputs values from two registers, which is the R-type instructions, so we can thus model our push instruction as an R-type instruction. There are many R-type instructions using the “0110011” opcode, which leaves no space for any more instructions using this opcode (based on the **funct3** opcode), so we give this instruction a new opcode and differentiate it with the 3-bit **funct3** opcode.
 - We then need to get the right data to the places, which means adjusting addresses. We first need to write the data to the reg file into register **x2**. We can do this by hardcoding the value “00010” into the **rd** field in the push instruction (recall we’re using an R-type format). Therefore, we’ve taken care of the address that we need to write to the reg file.
 - The data we need to write to the register file is the data in **x2** minus 4 (**x2-4**). We do this getting the **x2** data out of the register file by hardcoding the **adr1** input to “00010”. Once on the output, we subtract 4 from that value and feed it into the register file select MUX, which means we have to grow the MUX select (**rf_wr_sel**) by one select variable. The act of subtracting four from the value indicates that we also need to include some type of adder. Note that this approach opted to not use the ALU. We could have achieved the same result by adding a “-4” input to the **srcB** MUX, which also would have required we add an extra select bit to **srcB** MUX.
 - **rs2** currently connects to the data input of the memory, so that does not need to change. The data address input connects to the ALU output, which is not going to help us. We need to place a 2:1 MUX on the memory **ADDR2** input to allow either the ALU output or the **x2-4** data for use as the data address.
 - We need to change the **CU_DCDR** because we added two new control signals and a new instruction. Recall that we need to expand the register file MUX to account for the adjusted SP, and the new MUX in front of the memory address. The two new control signals for the **CU_DCDR** are for extending the register file MUX and by one bit and adding a 2:1 MUX for the data address.
 - We need to change the **CU_FSM** to recognize the new 7-bit **OPCODE**, though it has the same control outputs as other R-type instructions.
- b) Changes you need to make to the RISC-V assembler:
- The assembler needs to recognize the new instruction and assign the correct opcodes; recall that we hardcoded two of the R-type instruction register address fields.
- c) Changes in RISC-V memory requirements:
- The memory size does not need to change to support this instruction. We did not change main memory, the PC, the register file, or the CSR register. Although we changed the **CU_FSM**, we did not add new states, which is the only way we could have increased memory size in that unit.
- d) Why this modification would be useful:
- This mod would be useful because it would change a push operation from two instructions into one, thus providing a glimmer of hope for the free world.
-

Example 21.7: Conditional Jump Instructions

Add the following instructions to the RISC-V MCU. Make as few modifications to hardware as possible. Implement these instructions in two clock cycles. These instructions are essentially conditional jump instructions. When we issue a **jal** instruction as a jump (not a subroutine call), we don't utilize the destination register (**rd**). For the following instructions, we use the **rd** register as a source address register rather than a register to write to as in the normal **jal** instruction.

```
jal_s  rd,imm      # jump if reg_file[rd] != 0
jal_c  rd,imm      # jump if reg_file[rd] == 0
```

examples:

```
jal_s  x25,my_label # jump if value in x25 is non-zero
jal_c  x4,my_label  # jump if value in x4 is zero
```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful
- why you couldn't use these instructions for subroutine calls
- how these instructions differ from the other branch-type instructions

Solution: Here is a possible solution. Keep in mind that we are adding something to the MCU, so we'll stay attentive of the possibility of things growing.

a) Changes you need to make to the RISC-V hardware:

- This instruction is some type of jump so we can use the **jal** instruction format, which is a J-type instruction. What we need to do is to get a register value to the **BRANCH_COND_GEN** module; the registers already both connect to the module but the J-type instruction format has only one field for registers, which is the destination register. We'll use the **rd** field in the **jal** instruction as a source register and not a destination register. We can use either source register for this as both the **rs1** and **rs2** outputs from the reg file connect to the **BRANCH_COND_GEN**. To make this work, we need to place a 2:1 MUX in front of the **adr2** input on the reg file (this choice is arbitrary; you could use either source address for this) so that we can use the destination address (**wa**) field in the J-type instruction as a source address. The inputs to this MUX would be the current **adr2** input (**ir[24:20]**); the other input would be the current **wa** input value (**ir[11:7]**). In this way, when we issue this instruction, we can use the **rd** value as a source address.
- Due to the previous bullet, the **rs2** has the address of **rd**. The **rs2** output already connects to the **BRANCH_COND_GEN**, but we need to slightly modify the module. We need to add a single-bit output to the **BRANCH_COND_GEN** that indicates when the **rs2** input is zero or non-zero; the new hardware would drive this output and connect to the **CU_DCDR** as an input.
- We need to modify the **CU_DCDR** to recognize the new output from the **BRANCH_COND_GEN** and to recognize the new instructions. Since we are using a J-type format for our new instructions, there is no **funct3** field, so each of these instructions would require its own 7-bit opcode to differentiate these instructions from the current **jal** instruction. The **CU_DCDR** would also now have a new control output, which would be the select input to the MUX we added in front of the reg file's **adr2** input. The **CU_DCDR** would decide to take the jump or advance the PC to the next instruction.
- The **CU_FSM** would need to change to recognize the new 7-bit opcodes for these instructions and output the correct control signals, which would be different from the **jal** instructions.

- b) Changes you need to make to the RISC-V assembler:
- The assembler would need to be modified to recognize the two new instructions so it could generate the correct machine code for the instructions.
- c) Changes in RISC-V memory requirements:
- We did not change main memory, the reg file, the CSR module, or the PC. We also did not add new states to the CU_FSM, so this change would require no memory changes.
- d) Why this modification would be useful:
- This instruction would save a few instructions as it creates a conditional branch. The jal and jalr instructions are currently unconditional branches.
- e) Why you could not use these instructions for calling subroutines:
- You could not use this instruction for subroutines calls because a subroutine call would use the destination register to store the return address, which is typically x1. The way we implemented this instruction took away to the rd write addresses and used it as a source address.
- f) How these instructions differ from the other branch-type instructions
- These new instruction allow for farther jumps than branch-type instruction based on the width of the immediate field in the jal instruction as opposed to the branch-type instructions.

Example 21.8: Reg-Reg Swap Instruction

Add the following instruction to the RISC-V MCU. Make as few modifications to hardware as possible. This instruction swaps the data in two registers.

```
reg_swp  rs1,rs2    # swaps the values in source registers
```

example:

```
reg_swp  x10,x11
```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful
- describe an algorithm where this instruction could particularly useful

Solution: Here is a possible solution. Note that this problem requires you to add something to the RISC-V MCU, so there is a possibility that memory requirements will change.

- a) Changes you need to make to the RISC-V hardware:
- The most obvious thing we see from this problem is that it has two reg file writes; that means there is no way to do this in one execute cycle. We're going to have to add extra cycles for this instruction. We can do this problem in many ways, but they are all complicated. The problem is storing intermediate data while we do the swap. The solution is to use the XOR trick and swap the registers "in place". This is what we want: `xor rs1,rs2,rs1` followed by `xor rs2,rs1,rs2` followed by `xor rs1,rs2,rs2` where the left-most operation is the destination register.

- First, let's decide to use a B-type instruction as it has two source operands and no destination operand. We need both source operands as the destination operand (not at the same time), so we need to put a MUX in front of the **wa** input to the MUX. It needs to be a 4:1 MUX because we'll need to choose between three different values as the **wa** input (the three current reg file addresses). This creates the need for two select inputs.
 - We don't need to change the CU_DCDR other than to recognize this instruction in order send out the correct control signals. We need to make the CU_DCDR aware of the opcode for this instruction.
 - We need to add the two select signal for the new MUX to the CU_FSM. We need to add them here because we need to change the values for each execute cycle of the instruction. Since the CU_DCDR does not know of the cycles, we can't add them to that module. We need to do three task reg file writes for this instruction, so we need to add two more states to the CU_FSM. We can get one write done with the current execute cycle, but we then need do add two more cycles for the other two required XOR operations for this instruction. On an exam, you would for sure want to draw the new state diagram for clarity.
- b) Changes you need to make to the RISC-V assembler:
- The assembler would need to be modified to recognize the two new instructions so it could generate the correct machine code for the instructions.
- c) Changes in RISC-V MCU memory requirements:
- We did not change main memory, the reg file, the CSR module, or the PC. We added two new states to the FSM, which currently has four states. We maxed out the code space for the two bits of state variable, so we need to add another bit, which gives us the option of coding eight states. This was a 50% increase in memory for the CU_FSM.
- d) Why this modification would be useful:
- This instruction would do two things: save a few instructions, and because it does an "in-place" swap, we used less registers.
- e) Describe an algorithm where this instruction would be useful:
- This instruction would be ideal for sort algorithms.
-

Example 21.9: Pop Instruction

Add the following instructions to the RISC-V OTTER MCU. Make as few modifications to hardware as possible. This instruction assumes x2 is used as the stack pointer.

```
pop    rs1    # pop data from stack into rs1;
```

example:

```
pop    x10
```

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful

Solution: This is similar to the push instruction we described in a previous solution we conquered. We're adding something to the MCU, so we may cause an increase in memory requirements. The first thing you want to do in a problem like this is to understand the underlying RTL. Here it is for a pop operation:

$$rs1 \leftarrow mem[sp]; \quad sp \leftarrow sp+4$$

a) Changes you need to make to the RISC-V hardware:

- The first thing to notice about this problem is that we're doing two writes to the register file. That means there is no way we can implement this instruction with one execute cycle. This gets ugly; hang on. We need two cycles: the first cycle copies data from memory to the register file; the second cycle advances sp and stores the result in sp.
- Make this an I-type instruction. Give it a new 7-bit opcode to differentiate it from other I-type instructions. Hardcode the immediate field in the I-type instruction to zero. Hardcode the rs1 in the instruction to 2, which is the address of the sp. The output of the ALU then has the memory address of the sp, which is the data we need to load into the register in the register file. The output of the memory already connects to the register file. This would be the first cycle. The second cycle would be to add a "+4" box to the rs1 reg file output, and connect that output to the register file MUX. This means we need to change the register file MUX to be an 8:1 MUX and add an extra bit to rf_wr_sel. We also need to place a MUX in front of the wa input to the reg file so we can route the adr1 reg file address there to be the write address for the second cycle. But wait, it gets worse.
- We need to change the CU_FSM in a few ways. First, we need to add an extra state to the state machine for this instruction. Second, we need to override the entire register file MUX for this instruction. We need to do this because rf_wr_sel needs to be different for each execute cycle for this instruction, and the CU_DCDR controls this signal. We need to add a MUX to the external hardware to have the CU_FSM take over the rf_wr_sel signal for this instruction. This would require an extra output from the CU_FSM to control this MUX.
- We need to modify the CU_DCDR so that it recognizes this instruction with its new opcode. We need to add a select signal output from the CU_DCDR to control the MUX for the reg file's wa input.

b) changes you need to make to the RISC-V assembler

- We need to make the assembler aware of this new instruction and send out the correct associated machine code.

c) Changes in RISC-V memory requirements:

- We did not change main memory, the PC, the CSR, the reg file. We did add a new state to the CU_FSM, which put us at five states instead of four states. This requires an extra bit in FSM's state registers.
- d) Why this modification would be useful:
- This would save an instruction when you need to do a pop, and make the code more readable as opposed to tweaking the immediate values on the associated **lw** instruction.

Example 21.10: Adding a HALT Instruction

Add the ability to pause program execution. Many CPUs contain a “HALT” instruction, so I want the RISC-V to have one also. Describe the changes you need to make to the associated hardware and the assembler in order to implement this instruction and any other instruction I would need as a result of executing a “HALT” instruction. For this problem, do the following:

For this problem, describe the following:

- changes you need to make to the RISC-V hardware
- changes you need to make to the RISC-V assembler
- changes in RISC-V MCU memory requirements
- why this modification would be useful

Solution: The MCU is always doing something; the designers planned it that way. Recall that the never stopping feature is a characteristic of embedded systems. HALT-type instructions are useful in many ways, the ways are not worth going into here. The most obvious advantage would be to reduce power consumption by stopping the MCU from doing anything. There is nothing special about this solution, so you can definitely come up with a better one yourself.

- a) Changes you need to make to the RISC-V hardware:
- Implementing this instruction would require the Control Unit to be modified in order to recognize this instruction and send out the appropriate control signals. Adding an instruction does not change the number of states in the control unit thus does not change memory requirements. We would implement this instruction by causing the MCU to go into a “HALT” state if the MCU executed this instruction. This could possibly cause an increase in the memory associated with the control unit as we’re officially adding a state.
- The main issue with this instruction is how you would restart the MCU once you executed a HALT instruction. The question is rather misleading on this issue as it suggests that you must add another instruction to “START” the CPU. But, if the MCU is HALTed, you won’t be able to execute an instruction. The only solution is to have some external signal “unhalt” the MCU. In this case, the HALT instruction would cause the Control Unit to go into a “HALT” state; in this case, some external signal, strangely similar to an interrupt, would be required to get the MCU doing something meaningful again by leaving the HALT state. This signal could be anything, such as a user button-press or something similar. Keep in mind in real life, CPUs do all they can to turn themselves off if they are not being used; these are referred to as power-saving modes. But, they need to quickly wake up when something important needs the CPUs computational abilities.
- b) changes you need to make to the RISC-V assembler
- We need to make the assembler aware of this new instruction and send out the correct associated machine code.
- c) Changes in RISC-V memory requirements:

- We did not change main memory, the PC, the CSR, the reg file. We did add a new state to the CU_FSM, which put us at five states instead of four states. This requires an extra bit in FSM's state registers.
- d) Why this modification would be useful:
- This would essentially provide a power-saving mode to the MCU. This would also be a useful instruction if you were using the RISC-V in a multi-processor environment.
-

21.3 Chapter Summary

- Because the RISC-V architecture was provided to you, it leaves little room for actual hardware design. In cause you may have not noticed, hardware design is a significant aspect of this course. If you truly understand the current RISC-V MCU hardware and how it interfaces with the RISC-V ISA, you should be able to make modifications to the RISC-V to extend the current functionality.
 - This chapter presented a bunch of possible modifications to the RISC-V MCU and/or RISC-V instruction set. The learning element here is comes by understanding the solutions.
 - Any changes to “additions” to the current RISC-V MCU may affect the complexity and storage requirements of the hardware; and “reductions” made to the RISC-V MCU may have the opposite affect.
 - You must reference the RISC-V assembly language manual and RISC-V MCU architectural diagram in order to complete these problems. No one should expect you to memorize reference data such as that stuff. A complete understanding of the instruction formats is vital when working with these problems.
-

21.4 Chapter Exercises

- 1) The text describes these hardware modification problems as “open ended”. Briefly describe what that means.
 - 2) Briefly describe how it is possible to reuse instruction mnemonics for new instructions and not freak out the assembler.
-

21.5 xxxxChapter Design Problems

- 1) Add the following instruction to the RISC-V OTTER MCU. Make as few modifications to hardware as possible. Implement this instruction in two clock cycles. This instruction stores the word in the rd register to memory and leaves the rd value cleared.

```
mr_clr      rd,imm(rs)      : xd ← 0; mem[X(rs)+sext(imm)] ← xd
```

example: mr_clr x25,8(x10) # write x25 to memory; clear x25

- a) Describe changes you need to make to the RISC-V hardware
 - b) Describe changes you need to make to the RISC-V assembler
 - c) Describe changes in RISC-V MCU memory requirements
 - d) Describe why this modification could be useful
-
- 2) xxxxImplement the following instruction: “INOUT r0,0x23” where the source operand is considered a port_id. This instruction will simultaneously input a value and write it to r0 (the destination operand) and also output the value to the port_id (the source operand).
 - 3) Implement the following instruction: “IN 0x37,0x23” where the source operand (right operand) is considered a port_id and the destination operand is an address used to index RAM. This instruction inputs the value from the input port to the given RAM address.
 - 4) Implement the following instruction: “ROLST r1,(r2)” where the destination operand (r1) is rotated left and the result is stored in the RAM location referenced by the value in the source operand (r2).
 - 5) Implement an instruction that automatically increments the index register for indirect LD instructions. For example, when I write “LD r0, (r1+)”, the value from address value in r1 is loaded from memory into r0 and the value in r1 is automatically incremented as part of this operation. For this problem, do the following:
 - 6) I want to implement the following instructions: “SRSWP” which stands for “RAM swap”. This instruction has the form “SRSWP r1,(r2)”, which means it swaps the value in r1 with the contents of the memory location indicated by the value in r2.
-

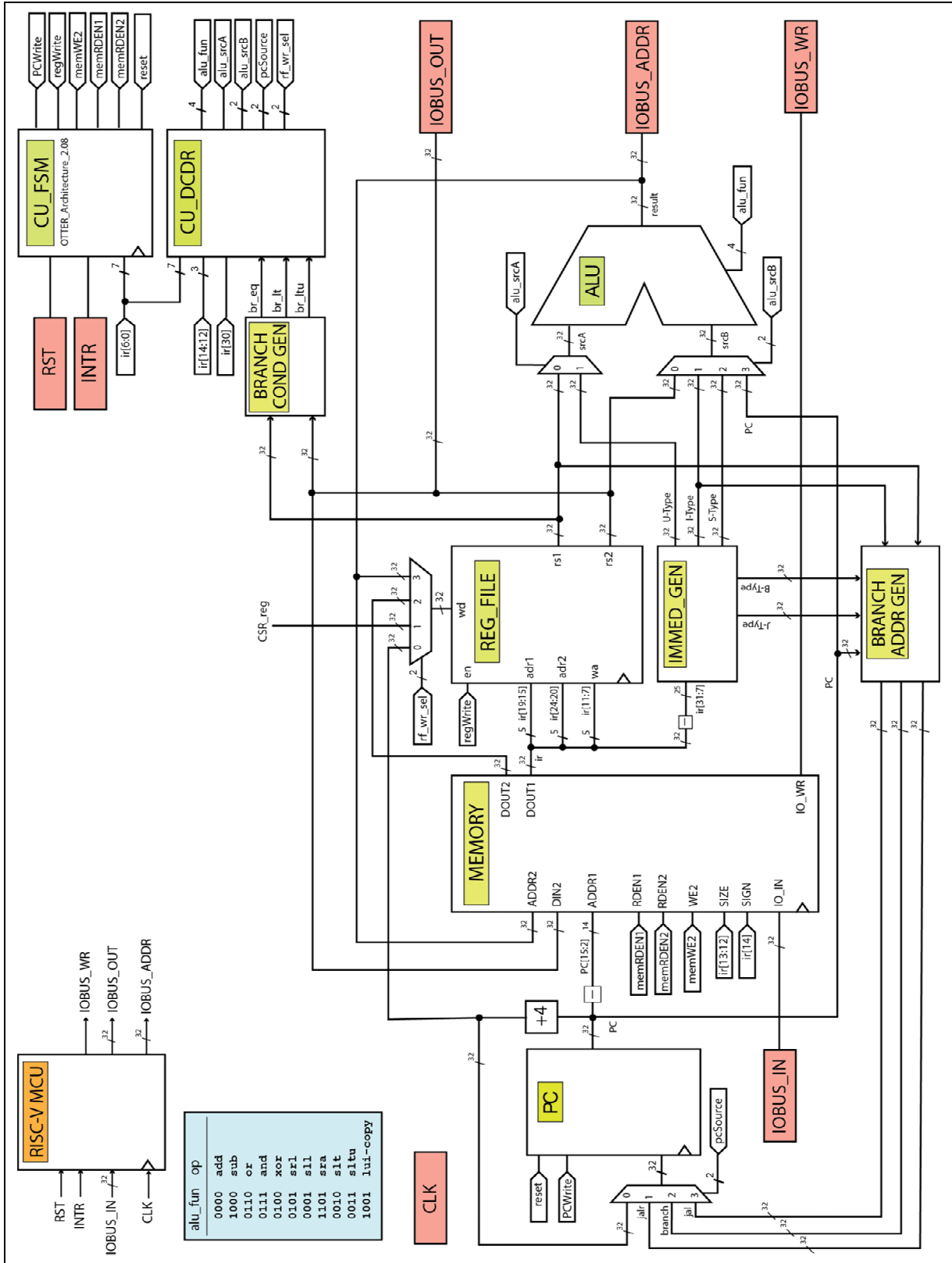
Appendix

Foundation Modeling Cheatsheet

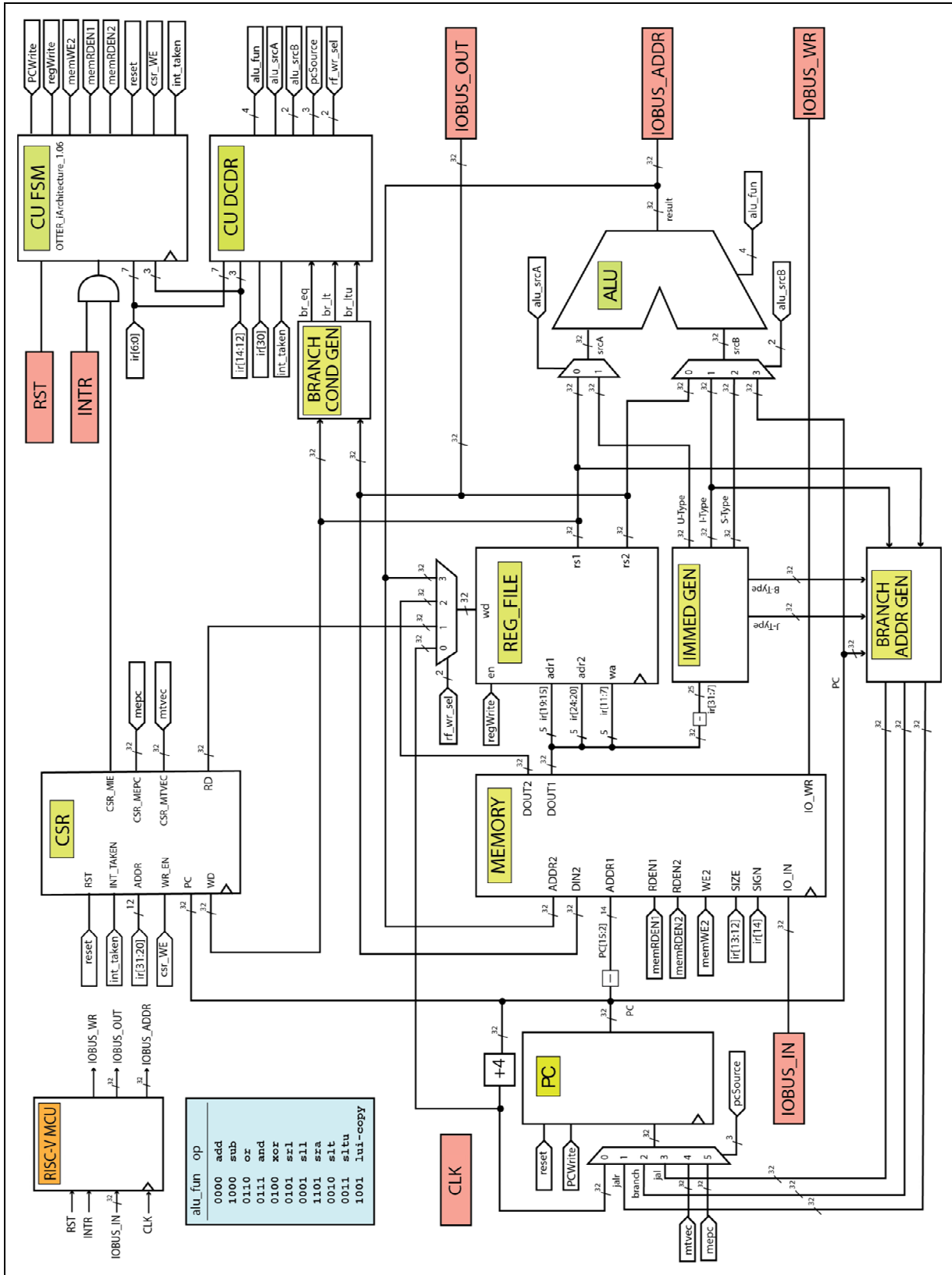
		Circuit Diagram	Data IN	Control IN	Data OUT	Status OUT
Combinatorial	RCA		A B Cin	-	SUM	Co
	MUX		Multiple DATA	SEL	Single DATA	-
	Generic Decoder (LUT)		IN_DATA	-	OUT_DATA	-
	Standard Decoder		IN_DATA	SEL	OUT_DATA	-
	Comparator		A B	-	-	EQ GT LT
	Parity Generator		DATA	-	-	PARITY
Sequential	Register		IN_DATA	CLK LD CLR	OUT_DATA	-
	Counter		IN_DATA	CLK LD UP/DOWN ENABLE	COUNT	RCO
	Shift Register		IN_DATA	CLK LD SH LEFT/RIGHT data ENABLE	OUT_DATA	-
	RAM		IN_DATA -	CLK WE ADDR	OUT_DATA -	-
					Inputs	
	FSM		-	CLK status	-	control
FSM Models				Moore		
				Mealy		

DATA = multiple bits data = single bit

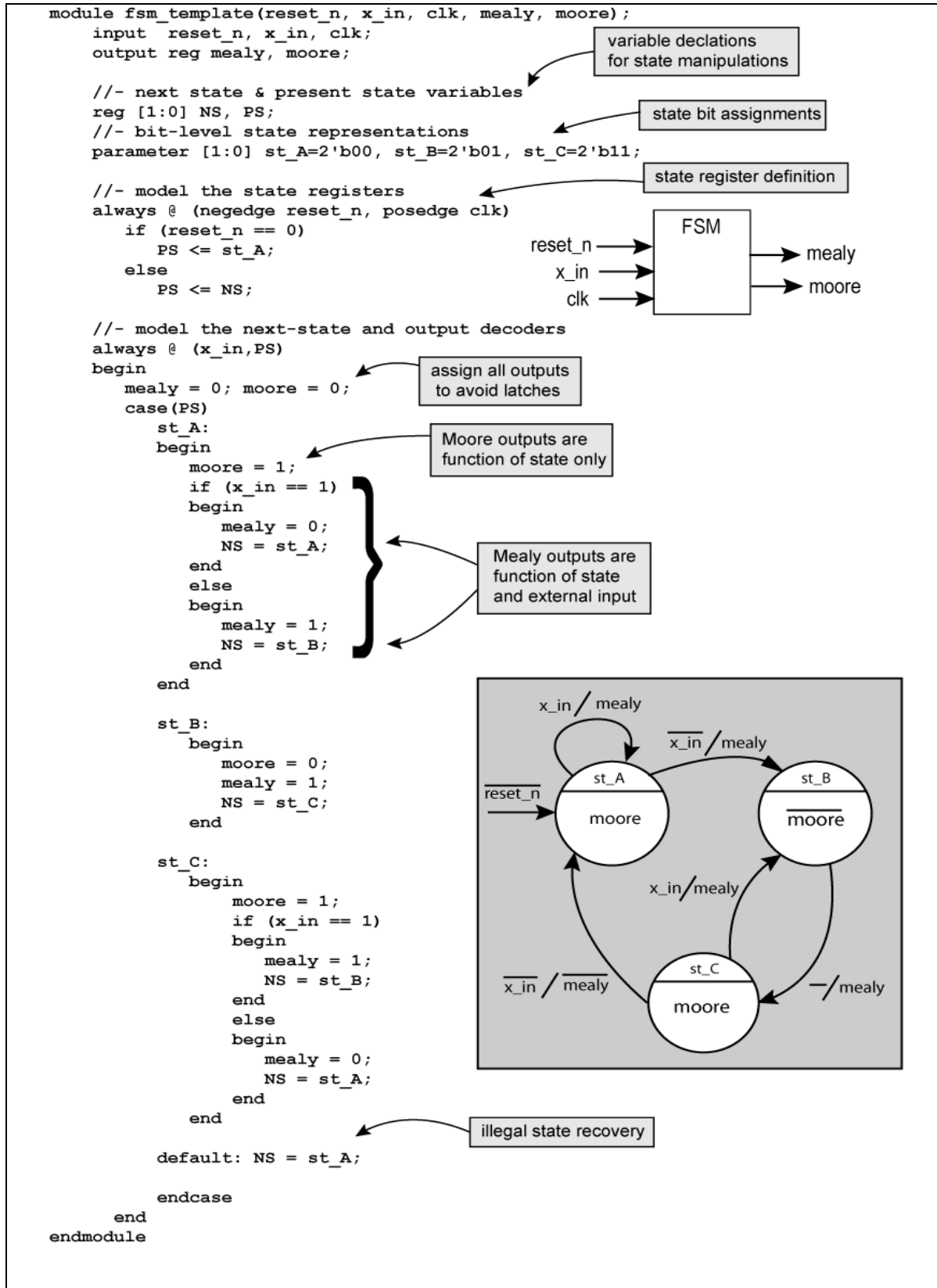
RISC-V OTTER MCU Architecture Diagram (no Interrupts)



RISC-V OTTER MCU Architecture Diagram (with Interrupts)



Finite State Machine Modeling using Verilog Behavioral Models



RISC-V MCU Wrapper Source Code

```

// Engineer: James Ratner, Joseph, Paul Hummel, Celina Lazaro
// Revision 1.04 - (02-08-2020) removed typo for anodes

module OTTER_Wrapper(
    input clk,
    input [4:0] buttons,
    input [15:0] switches,
    output logic [15:0] leds,
    output logic [7:0] segs,
    output logic [3:0] an );

    //- INPUT PORT IDS -----
    localparam SWITCHES_PORT_ADDR = 32'h11008000; // 0x1100_8000
    localparam BUTTONS_PORT_ADDR = 32'h11008004; // 0x1100_8004

    //- OUTPUT PORT IDS -----
    localparam LEDES_PORT_ADDR = 32'h1100C000; // 0x1100_C000
    localparam SEGS_PORT_ADDR = 32'h1100C004; // 0x1100_C004
    localparam ANODES_PORT_ADDR = 32'h1100C008; // 0x1100_C008

    //- Signals for connecting OTTER_MCU to OTTER_wrapper
    logic s_interrupt;
    logic s_reset;
    logic s_clk = 0;

    logic [31:0] IOBUS_out;
    logic [31:0] IOBUS_in;
    logic [31:0] IOBUS_addr;
    logic IOBUS_wr;

    //- registers for dev board output devices -----
    logic [7:0] r_segs; // register for segments (cathodes)
    logic [15:0] r_leds; // register for LEDs
    logic [3:0] r_an; // register for display enables (anodes)

    assign s_interrupt = buttons[4];
    assign s_reset = buttons[3];

    //- Instantiate RISC-V OTTER MCU
    OTTER_MCU my_otter(
        .RST (s_reset),
        .intr (1'b0),
        .clk (s_clk),
        .iobus_in (IOBUS_in),
        .iobus_out (IOBUS_out),
        .iobus_addr (IOBUS_addr),
        .iobus_wr (IOBUS_wr) );

    //- Divide clk by 2
    always_ff @ (posedge clk)
        s_clk <= ~s_clk;

    //- Drive dev board output devices with registers
    always_ff @ (posedge s_clk)
    begin
        if (IOBUS_wr == 1)
            begin
                case (IOBUS_addr)
                    LEDES_PORT_ADDR: r_leds <= IOBUS_out[15:0];
                    SEGS_PORT_ADDR: r_segs <= IOBUS_out[7:0];
                    ANODES_PORT_ADDR: r_an <= IOBUS_out[3:0];
                    default: r_leds <= 0;
                endcase
            end
        end
    end
end

```

```
    //- MUX to route input devices to I/O Bus
    //- IOBUS_addr is the select signal to the MUX
    always_comb
    begin
        IOBUS_in=32'b0;
        case(IOBUS_addr)
            SWITCHES_PORT_ADDR : IOBUS_in[15:0] = switches;
            BUTTONS_PORT_ADDR  : IOBUS_in[4:0]  = buttons;
            default: IOBUS_in=32'b0;
        endcase
    end

    //- assign registered outputs to actual outputs
    assign leds = r_leds;
    assign segs = r_segs;
    assign an = r_an;

endmodule
```


Verilog Style File

James Mealy v1.00

The main goal of your Verilog source code is to model a digital circuit, which means that your only required mission is to satisfy the Verilog synthesizer. Good Verilog models must both work properly and be readable by humans. Poorly written Verilog models can be work properly but are not maintainable or reusable if humans can't easily read and understand the code. Digital designers can generate good Verilog models by following a few relatively simple guidelines. The following code describes how digital designers can create superbly formatted Verilog models, but does not cover aspects of how to properly use the various Verilog constructs. Use this document in conjunction with the Verilog Coding Guidelines to help you create most excellent Verilog models. The overriding factor with your Verilog source code is to make it neat, organized, and readable; any specific items not listed in this style file should adhere to these principles.

```
// The file contains a header describing the important features of the file.

////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer:  James Ratner & Myron Bucketts
//
// Create Date: 07/07/2018 08:05:03 AM
// Design Name:
// Module Name: prime_gen_fsm
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Model contains the names of the model creator, name of
// the module, and a description at the very least. You should also track
// revisions of the model also. This is the standard Xilinx header which
// contains other items we choose to remain blank.
//
// Dependencies:
//
// Revisions:
// Revision 1.00 - (07-07-2018) File Created
//
// Additional Comments:
//
////////////////////////////////////

// The module name describes the module's purpose, separates inputs and outputs,
// and only places one item per line. We opted to include types (wire or reg),
// but this is not necessary.
module prime_gen_fsm(
    input wire PRIME,
    input wire DONE,
    input wire RCO,
    input wire btn,
    input wire clk,

    output reg START,
    output reg WE,
    output reg UP1,
    output reg UP2,
    output reg CLR,
    output reg SEL  );

    /* Longer comments are more easily modified if you delineate them using
    block comments.
    */

    /* Note how we separate each "item" with whitespace (blank lines). We
    Do this in the entire file. While this approach makes the code model
    Longer, it does not affect the size of the synthesized hardware.
    */

    /* Note that we place all declarations at the top of the model and do
    Not inter-mingle declarations throughout the code.
    */
```

```
*/

// next state & present state variables
reg [1:0] NS, PS;

wire s_clk; // divided (slowed) clock signal

// bit-level state representations
parameter [1:0] st_wait=2'b00, st_start=2'b01, st_work=2'b11;

/* We use the vertical "dot" form for instantiations. There is one
   mapping per line, everything is nicely aligned, and we use self-
   commenting names.
*/

// divide the FSM clock down
clk_2n_div_test #(.n(25)) fsm_clk_divider (
    .clockin   (clk),
    .fclk_only (1'b0),
    .clockout  (s_clk) );

/* this is a sequential block based on the "posedge" argument in the
   Sensitivity list. Note that we use non-blocking assignment statements
   Because this block models a sequential circuit (state registers)
*/

// the state registers
always @ (posedge s_clk)
    PS <= NS;

// the next-state and output decoders
always @ (*)
begin

    // we place many assignments on the same line because they serve
    // a similar purpose.
    START=0; WE=0; UP1=0; UP2=0; CLR=0; SEL=0; // assign all outputs

    /* All the cases in the case statement are nicely delineated.
       The if clauses in the final cases are also separated using
       whitespace (blank lines).

       All cases are represented so the case statement does not rely
       on the default clause to work properly.

       Longer blocks use comments for "ends" to indicate what they
       are ending.

       If statements with compound requirements are delineated using
       parenthesis.

       This is a combinatorial block (a decoder) so all if statements
       contain else statements, all case statements contain default
       statements, and we use block assignment statements.
    */

    case(PS)

        st_wait: // waiting for button press
        begin
            if (btn == 0)
            begin
                CLR = 0;
                NS = st_wait;
            end
            else
            begin
                CLR = 1;
                NS = st_start;
            end
        end
    endcase
end
```

```
    end
  end

  st_start: // prepare FSM to start calculation
  begin
    START = 1; SEL = 1;
    NS = st_work;
  end

  st_work: // state doing the main work
  begin
    START = 0; SEL = 1;
    if ( (RCO==1) && (PRIME==1) && (DONE==1) )
    begin
      WE = 1;
      NS = st_wait;
    end

    else if ( (DONE==1) && (PRIME==0) )
    begin
      UP1=1; UP2=0; WE=0;
      NS = st_start;
    end

    else if ( (RCO==0) && (DONE==1) && (PRIME==1) )
    begin
      UP1=1; UP2=1; WE=1;
      NS = st_start;
    end

    else if (DONE==0)
      NS = st_work;

    else
      NS = st_work;
  end // ends current case

  default: NS = st_wait;

endcase
end // ends always block
endmodule
```

RISC-V MCU Assembly Language Style File

The following file shows some of the more important issues regarding generating neat and readable RISC-V MCU assembly source code. No style file can show you everything and they rarely make such an attempt. The underlying factor in writing any source code is to be neat and consistent. Using proper indentation, white space and commenting helps you attain the goals of being neat and consistent. The code is some example problem that we edited for clarity and to make it shorter. The code does not assemble, but that is not the point; the program is presented primarily for appearance purposes. Also, you can't see it in the following code, but never use the tabs of align items; use spaces instead.

```

#-----
# Program Description:
#
# The entire program contains a header (or banner) describing the purpose of
# the program. The more detail you can provide here the better, as you or anyone
# else reading the program will want to know the details. Note that this banner
# clearly delineated from the remainder of the program.
#-----

# the data segment is listed before code; all data in the data
# segment is aligned and divided up between lines for clarity
#---- data segment -----
.data                                # data segment
junk:      .word  0x3, 0x7           # describe purpose of data
bugs:      .word  0x32, 0xDD        # don't try to fit data on one line
trash:     .byte  0x3, 0x7         # more description

# Always declare a text segment even if you have no data segment
# The first part of any program or subroutine should have some
# type of "init" label.
.text                                  # text segment
init:      la     x10,junk           # load address of junk
           li     x20,2             # load count of data

# Use white space (blank lines) between sections of program
# that perform distinctively different tasks.

# Align all labels, instructions, left-most operands and comments
# One comment per line is a good assembly language programming approach.

init1:     mv     x25,x0             # designated large value
           addi  sp,sp,-4           # make space for ra
           sw    ra,0(sp)           # store return address

loop1:     beq   x20,x0,done1        # quit if count is zero
           lw    x30,0(x10)         # get value
           call  Calc_unary         # find unary equivalent

admin:     addi  x10,x10,4           # advance address
           addi  x20,x20,-1         # decrement count
           j     loop1              # repeat

done1:     lw    ra,0(sp)            # pop return address
           addi  sp,sp,4            # adjust sp
           ret    # going home, all the time

#-----
# Subroutine: My_sub:
#
# All subroutines have banners describing that the subroutine does, the values
# passed to and returned from the subroutine, and the register that the
# subroutine permanently changes.
#
# Passed values: x30
# Returned values: x29
#
# Tweaked register: x25, x31, x29
#-----
My_sub:
init2:     mv     x31,x0             # init count
loop2:     beq   x30,x0,done2        # see if no more ones
           andi  x29,x30,1          # mask LSB
           add  x31,x31,x29         # accumulate count
           srli x30,x30,1           # shift value 1 to right
           j     loop2              # do it again
done2:     ret    # bring it home

```

Glossary of Computer Design Terms

-A-

Abstract Data Types: A data type that is described at a high-level, such as how the object should behave, rather than describing the type using low-level implementation details.

Academic: The rallying cry for those who dare to expose the endemic corruption in academia.

Active Edge: The portion of a logic signal used to synchronize digital circuit operation; can either be rising edge ('0' to '1' transition) or falling edge ('1' to '0') transition.

Address: (or Memory Address), the way semiconductor memory devices (structured memory) specify memory locations. The address is analogous to an array "index" in higher-level programming languages.

Address Space: The maximum amount of memory a given processor can access (or address). This does not refer to the actual amount of memory (physical memory) in any given system.

Anode: The positive end of a diode (the end that accepts electrons). See *cathode*;

Arithmetic Logic Unit (ALU): The ALU is generally a datapath submodule, which in turn is a submodule of CPU. The ALU is responsible for standard bit operations such as arithmetic and logical operations (and shifts and any other way you can think of to tweak bits). The ALU is responsible for generating status of various operations (zero, negative, overflow, carry, pointlessness, parity, etc.) which are typically individual bits that are latched outside of the ALU.

Arithmetic shifts: Shift operations that protect the sign of data residing in a shift register when performing shift operations.

Assemble Time: The notion of what values are known at the time a program is assembled. Generally speaking, the assembler knows constants values at assemble time but does not know constant values until run time.

Assembler Directives: One of the three main parts of an assembly language program. Assembler directives provide a method for the programmer to send messages to the assembler.

Assembler: An assembler is a computer program that translates assembly code (instruction mnemonics) into machine code.

Assembly Language Program Parts: There are generally three types of information found in assembly language programs: 1) comments, 2) assembler directives, and, 3) assembly language instructions.

Assembly Language: A computer language that uses mnemonics to represent the instructions available to the programmer (the instruction set) for a given computer architecture. The mnemonics roughly spell out what the instruction does in terms of the underlying hardware. Assembly language programs are translated to machine code by use of a software program referred to as an assembler. Assembly language is generally non-portable in that the assembly instructions are specific to a given computer architecture.

Astable Multivibrator: A term referring to a signal with no stable states, typically an oscillator (clock signal).

Asynchronous: A term that refers to digital circuits whose operations are not synchronized to any signal such as a clock signal. This term often is a synonym for combinatorial circuits.

-B-

Background Task: A term used to describe the program code associated with main code and not associated with interrupt service routines. The foreground task is generally all the code that is not initialization code or interrupt service routine code.

Barrel shifts: A special type of shift register shift that shifts any number of bits (other than one bit) on a single clock cycle.

Base Address: A value that is one of the values to calculate a physical address; another value, such as an *offset*, is used to modify the base address to create a physical address.

Big Endian: One of two ways to represent a multibyte value in a byte-oriented memory that places the most significant byte of the data at the lower address value. See *little endian*.

Bit: A term commonly used to describe a *binary digit*.

Bitwise: A really smart bit, or a term meaning that an operation on a set of bits is done at the bit level (on individual bit-pairs) and not on the entire set of bits. You typically hear this term associated with MCU's logic-based instructions.

Bistable Multivibrator: A device with two stable states (on and off); another term for a flip-flop.

Bi-Directional Signals: A term that refers to the notion that data can flow through a line in two directions (though not at the same time) rather than only one direction. Bi-directions signals are generally associated with tri-state outputs because a given device cannot generally simultaneously drive a signal and read from that signal.

Bit Masks: The term bit-mask describes a value that "selects" certain bit locations of a given word while disregarding other bit locations. The disregarded bits are generally cleared by the bit-masking operation. Bit masking is generally required because most operations in microcontrollers occur on the byte-level.

Bit-Banging: The process of using microcontroller outputs on a bit-level to control external peripherals. In this way, the general purpose outputs of a microcontroller are used to generate the control signals required to control and/or exchange information with external devices.

Bit-Wise Operations: This term generally refers to operations that on done on individual sets of bits in registers, such as logic operations.

Block Diagram: A modeling approach used in hardware to quickly transfer high-level knowledge regarding the operations of a given circuit to the human reader. Block diagrams can and should be hierarchical in nature when appropriate to expedite their understanding to the human reader.

Branch: A computer instruction that can cause program flow to transfer to an instruction other than the instruction following the current instruction. Branch instructions are by definition conditional, meaning program control transfers based on the state of the hardware or a condition encoded into the instruction.

Bus Contention: Bus contention occurs when two different busses attempt to simultaneously drive the same bus. In this context, the bus is a *shared resource*. Contention can also occur on individual signals as well as busses.

Bus: A set of electrical signals that are grouped together because they share a common purpose. The term "bus" also refers to various standard data transmission protocols, and as a result, a bus, as defined here is often referred to as a *bundle*.

Byte: A set of eight binary digits.

-C-

Carry: The bit that overflows or underflows from a mathematical or shift operation.

Cathode: The negative end of a diode (or the end that sources electrons). See *anode*.

Central Processing Unit (CPU): The CPU is generally considered the part of the computer that executes the instructions. Typical submodules of the CPU include the control unit, datapath, program counter, instruction memory, register files, accumulators, ALUs, secondary memory, roach motels, etc.

CISC: An acronym for *complex instruction set computer*. See complex instruction set computer.

Clear: The act of making a bit value into a '0'. Often used to refer to making a set of bit's all zeros, such as "clear the register".

Clock: A signal that sequences or synchronizes all operations in a sequential digital circuit. Clock signals are typically periodic outputs from oscillators circuits such as astable multivibrators.

Code Space: The part of a computer system's memory dedicated to the program memory.

Complex Instruction Set Computer: This acronym officially stands for “Complex Instruction Set Architecture” and is generally used to describe computer architectures. CISC computers generally have the following characteristics:

The architecture contains relatively few general purpose registers

- The instruction word formats are of different lengths
- Instructions require a different number of clock cycles to complete execution
- Some instructions in the instruction set are complex (meaning they can generate a significant amount of processing internal to the architecture)

System clock rates are generally slower than their RISC counter-parts.

Complexicated: Something that is both complex and complicated.

Constant: A value that never changes; the opposite of a variable.

Combinatorial vs. Sequential Circuits: The outputs of a combinatorial circuit are a function of the current inputs while the outputs of a sequential circuit are a function of the combination of past inputs. Stated differently, combinatorial circuits do not have the ability to “remember” bits while sequential circuits are able to store values and are this considered to have memory.

Compiler: A computer program that translates higher-level language code into machine code. Compilers generally also produce assembly language code listings, which are specific to the target computer. Compiling is generally a two-step process where the code is first translated to a generic intermediate form, then translated to a form specific to a given computer architecture.

Computational Complexity: A term that provides a way to classify and/or describe the amount of resources required to execute a program, section of code or algorithm.

Computer I/O: One of the three main subsections of a computer that allows the computer to interact with the outside world.

Context: A term that refers to the state of the processor at any given time, where state is defined by the data the given processor is storing at a given time. This term is synonymous with *operating context*.

Context Restoration: A term describing what a CPU does upon completion of servicing an interrupt. In this case, context restoration refers to the notion that the CPU must return to the state it was in (flags, registers, etc.) before the CPU executed the interrupt service routine.

Context Saving: A term that describes what a CPU must do when an interrupt is acted upon. The general notion is that interrupts are asynchronous and can occur while the CPU is executing some important piece of code. In this case, the CPU saves the current state of the CPU (flags, registers, etc.) before processing executing the interrupt service routine.

Counter: A hardware sequential device that generates a known sequence of values on the circuit’s outputs. The device is typically synchronous. Typical counter functions include increment, decrement, clear, and hold.

-D-

Data: A set of 1’s and 0’s.

Data Segment: The part of an assembly language program used to declare and/or define data; no instructions can be listed in the data segment.

Datapath: The hardware module that is generally considered to do the number crunching associated with instructions. Submodules of the datapath generally include the ALU, register file, accumulator, various selection logic, etc.

Debouncer: An entity that “debounces” a switch, which means the entity provides a noiseless state transition. Switch debouncing can be done in hardware or firmware.

Debug: The act or removing errors from an entity such as a computer program or hardware schematic/circuit.

Decoder: A standard combinatorial hardware device the implements Boolean functions characterized by tables. Two flavors of decoders include generic and standard decoders.

Decrement: The act of increasing a value by one; it's also one of the typical operations of a standard digital counter.

Delay: A given span of time in a circuit where state of the circuit does not change in a meaningful way.

Destination Operand: A given span of time in a circuit where state of the circuit does not change in a meaningful way.

Deterministic: An event that is known to happen the same way each time it occurs and can thus be described in advance.

Dev Board: A nickname for a development board.

Development Board: A populated PC board containing hardware that allows you to prototype various electronic circuit projects.

Diode: A two-terminal semiconductor device that passes current in only one direction (from anode to cathode). The diode causes a voltage drop across the device terminals when conducting.

Direct Memory Access: One of three main type of computer I/O, characterized by the MCU initiating data transfers with external peripherals but not expending significant amount of clock cycles controlling that I/O operation. The *programmed I/O* and *interrupt I/O* are the other two types.

Disassembly: The act of generating the assembly code that generated the machine code from the machine code.

Display Multiplexing: An approach typically used by LED-based 7-segment displays that allows the driving device to control many digits without dedicating a signal to each LED in each segment. The general approach is to connect each type of segments with one signal and give each individual display an on/off control. Using this configuration, display multiplexing only actuates one display at a time, but does so at a rate that makes it appear as if all displays are on at the same time. Multiplexing works for humans because of the notion of retinal persistence.

DMA: An acronym standing for *direct memory access*; see direct memory access.

Do-While Loop: An iterative loop characterized the fact that the loop body is executed at least one time, which it does by checking the loop ending condition only after it executes one loop body.

-E-

Edge-Triggered: A term referring to a sequential digital circuit whose state can only be changed synchronized with an active edge of a given signal, typically, a clock signal. The edge in question can either be a "rising edge" (0→1 transition) or a "falling edge" (1→0 transition).

Elementary Operation: A basic operation performed by a sequential circuit. Elementary operations are most often spoken of in terms of registers. Typical operations performed by registers include loading (generally a parallel load), setting (sets all bits in register), clearing (clears all bits in register), shifting/rotating (specifically for shift registers), and incrementing/decrementing (generally for counters).

Embedded System: The hardware and software of a computer system that typically performs a dedicated task. Embedded systems are well known to be hard to test and debug based on a limited number of input and output features. The software that runs embedded systems is typically firmware because it is specialized to run only one particular set of hardware.

Endianness: A term that describes how multibyte values are stored in byte-oriented memories; the choices are *little endian* and *big endian*.

Ethics: A quality that you either have or don't have. If you tell the world you have it, or you're an academic administrator, then you don't have any ethics.

Event: An occurrence of something that has meaning and/or significance. In the context of computers, an event is typically some occurrence the computer is expected to react to.

-F-

Feature Creep: A condition where device or program specifications grow over time while typically bypassing the appropriate channels for such changes.

Feedback: A portion of an entities output is returned (fed back) to the input to allow the input to be modified accordingly. Feedback can either be positive or negative feedback.

Fetch Cycle: The part of instruction execution that typically includes reading an instruction from program memory.

Field Code: A term referring to an underlying bit field in an assembly language instruction. Field codes are variables for a given instruction (meaning they have no set value); opcodes are constants for a given instruction.

Field Programmable Gate Array (FPGA): A programmable logic device (PLD) is an integrated circuit that contains internal devices that can be configured (or programmed) to implement a given digital circuit. The internal devices include logic, memory, routing, and input/output resources.

Finite State Machine (FSM): An abstract machine that defines a finite set of states, actions performed in those states, and a set of rules defining how the machine transitions from state to state. FSM are generally classified as either *Mealy* or *Moore* machines. FSMs are one of two major hardware devices that are typically used to control other hardware entities. In these cases, FSM inputs are considered status inputs while FSM outputs are considered control outputs.

Firmware: Firmware is a computer program that is written to run on a specific piece of hardware and is thus often associated with embedded systems. Firmware does not refer to the language-level in which the program is written thus can be written in machine code, assembly code, or a higher-level language.

First Five Things for a New CPU: When you first examine a new CPU, the five things you should initially examine are 1) the programmer's model, 2) the instruction set, 3) the interrupt architecture, 4) the memory model, and 5) the I/O architecture.

Flag: A value used a Boolean variable to that indicates a two-stateness of something (on-off, true-false, yes-no, etc.).

Flag Register: A register used to represent a flag value.

Flicker: An issue associated with display multiplexing where the multiplexing rate is slow enough for humans to note that displays are not "always on".

Flip-Flop: A synchronous single-bit store device (aka, bistable multivibrator). Typical flavors include D (data), T (toggle), and JK (unknown).

Flowchart: A diagram that uses a few distinctive symbols to model the program flow associated with an algorithm. Computer programmers use flowcharts as an aid to program design and/or documentation support. Flowcharts can and should be hierarchical in nature when appropriate. The hardware analogy to a flowchart is the black-box diagram.

Foreground Task: A term used to describe the program code associated with interrupt service routines.

Fragile: A label attached to code that is unmaintainable. Fragile code breaks if you attempt to modify it, hence the name fragile. The roots of fragile code are a complete lack of planning of the code as well as modifications made by people who don't know what the f**k they're doing.

-G-

General Purpose Computer: A computer with an instruction set that is designed to be flexible to give it the ability to solve a wide range of problems.

Generic Decoder: A generic decoder is a hardware implementation of a look-up-table (LUT). LUTs generally establish a functional relationship between inputs and outputs by assigning an output for every unique input.

Ghosting: An issue associated with display multiplexing where an LED is on when it should be off resulting in dimly lit LED showing incorrect information.

-H-

Hardware: The part of an embedded system that is not software and/or firmware; the stuff you can hold in your hand.

Harvard Architecture: A computer architecture that has separate memory space for both data and instructions.

Hex: A shorthand name for “hexadecimal”; see hexadecimal. Don’t look too hard on this one.

Hexadecimal: A number represented with a radix of 16 (base 16).

HDLs (Hardware Description Languages): Text-based languages used to model digital circuits. The main flavors of HDLs include VHDL and Verilog.

Higher-Level Computer Language: A computer language that uses opened-ended expressions and functions to generate desired results. Higher-level languages are generally input to computer programs such as compilers, which translate the languages to both assembly code and machine code associated with the target machine. Higher-level languages are generally portable (processor independent) and thus various higher-level language code can be compiled to run on different target machines.

High-Impedance: A term that refers to a device that effectively removes itself from a circuit by turning off its drive current. A device that cannot drive a circuit can no longer affect the circuit and is thus effectively not in the circuit.

Hold Time: The amount of time a synchronous circuit’s non-clock inputs must remain stable after the active clock edge; violation of hold times cause the hardware to go “metastable” and makes your circuit act like an academic administrator.

-I-

I/O: An acronym for *input/output*.

Immediate Operand: One of the operands of an assembly language instruction that is a constant value, but typically not an offset value.

Increment: The act of increasing a value by one; it’s also one of the typical operations of a standard digital counter.

Indentation: The act of using white space to align various parts of source code to make the code more readable to humans. In most meaningful computer languages, indentation is optional, but highly recommended. Chimpanzees can generate source code; intelligent people can generate museum quality source code.

Infinite Loop: An iterative structure that never terminates; the main code in embedded systems applications are typically encoded as infinite loops.

Information: A set of data that has a known meaning.

Input/Output: One of the three basic computer subsystems; it’s the subsystem that allows a computer to interact with the external world.

Initialization: A section of code that when executed, places a system in a known state. Typically, embedded applications and subroutines include a section of code dedicated to initialization.

Instruction format: The bit-level description of instructions associated with assembly language instructions. Each instruction is comprised of op-codes in every case, but also can include field codes in most cases.

Instruction Register: A common register in computer architectures that holds the machine code of the current instruction being executed by the computer; often referred to as simply *ir*.

Instruction Set: The instruction set describes the operations that the computer hardware can perform under program control (either software or firmware).

Instruction Set Architecture (ISA): A term that is typically used to refer to the instruction set organization and purpose for a given architecture. This is a more global and inclusive term for *instruction set*.

Iteration: The act of repeating something, such as the body of a loop in a computer program.

Iteration Count: The value that governs how many times something, such as a loop in a computer program, will iterate.

Interrupt Architecture: A common term used to describe all the characteristics (both hardware and software considerations) of interrupts for a given processor. Every computer device generally has a different interrupt architecture and is thus one of the three important aspects of any computer device (the programmer’s model and instruction set are the two other important aspects).

Interrupt Cycle: The steps a CPU goes through in order to handle an interrupt. The interrupt cycle is generally different from “normal” processing cycles.

Interrupt Driven I/O: Another form of I/O characterized by an external device having the ability to change the normal flow of a program by executing a special set of code referred to as the interrupt service routine.

Interrupt Handlers: An alternate name for interrupt service routines (ISRs).

Interrupt Masking: This refers to the notion that most interrupt architectures allow for the prevention of response to interrupts based on software control. Processor support for interrupts generally includes instructions that allow for processor response to interrupt signals (unmasking) or prevent system response to interrupt signals (masking).

Interrupt Service Routine (ISR): When a processor responds to an interrupt, the given interrupt architecture responds by executing a set of instructions known as an interrupt service routine. The ISR is nothing more than a subroutine that is executed after being “called” by some device. ISRs are often referred to as “interrupt handlers”.

Interrupt Service Routine: A section of code that the CPU executes automatically as a result of acting on an interrupt.

Interrupt Vector Address: The address the CPU places into the program counter when the CPU acts on an interrupt. Thus, when an interrupt is processed, the first instruction executed is the one residing at the vector address. The instruction at the vector address is generally a branch to the interrupt service routine.

Interrupts: An asynchronous signal from an external device to the processor. Exactly how the processor reacts when an interrupt is received is based on the interrupt architecture for a given processor. In simple terms, an interrupt can be considered a method for internal hardware or external devices to call a special subroutine (ISR). Interrupt signals are generally considered asynchronous in nature, which makes them vital to real-time embedded systems.

ir: Common acronym standing for *instruction register* (see *instruction register*).

Iterative: Something that repeats, such as a section of code in a loop.

-J-

JK Flip-Flop: A standard but antiquated type of flip-flop that has characteristics of both D and T flip-flops.

Jump: A computer instruction that causes program flow to be unconditionally transferred to an instruction other than the instruction following the current instruction.

-K-

Karnaugh Map: A sophomoric approach to reducing Boolean functions, only taught by those who fear modern digital design.

Kernel: Another word for operating system, typical a minimal operating system.

-L-

Label: A placeholder or alias that allows humans to understand computer programs. Labels are used in place of memory addresses. Some program statements reference labels, but labels can also be used for commenting purposes.

Latches vs. Flip-flops: Both latches and flip-flops are 1-bit storage devices. Latches are considered level-sensitive devices and its outputs can change anytime a change in its inputs occurs. Flip-flops are considered edge-sensitive devices and changes in outputs are synchronized to an edge-sensitive input, which is often assigned as a clock signal.

Latency: The span of time between the occurrence of an event and the beginning of the response to that event.

Lead Zero Blanking: A term associated with seven-segment display devices, where the left-most digit(s) in a given number are not displayed if they don't change the value of the number (meaning left-most zeros are not displayed). If the given number to display is zero, one zero is displayed in the right-most position.

LIFO: An acronym standing for “last-in, first out”, which describes the general operating characteristic of a stack data object.

Little Endian: One of two ways to represent a multibyte value in a byte-oriented memory that places the most significant byte of the data at the higher address value. See *big endian*.

Load: The act of latching data into a register. In terms of computer instructions, it's the act of loading data from program memory into a register in the register file.

Load-Store Architecture: A computer architecture that tweaks memory in registers rather than directly from memory. Data must first be loaded to registers for it to be tweaked.

Logic Analyzer: A device that debugs actual hardware by interpreting all signals as digital signals and outputting timing diagrams and/or state charts.

Look-Up Table (LUT): A LUT is a programming or hardware construct that translates an input value to a specific output value. Hardware LUTs are typically implemented with *generic decoders* while software LUTs are generally organized as a list of entries in successive memory locations. Hardware LUTs generally save on logic generation and can be used to speed-up hardware operations. Software/Firmware LUTs are typically used to avoid costly calculations at the cost of dedicating memory resources to the LUT.

Loop: A portion of computer code that is iterative, which means it can repeat based on the structure of the code.

Loop Count: A variable that controls the number of times the body of a loop is executed.

Low Power Mode: Many MCUs have the ability to adjust to ambient circumstances and operate using less power. These special modes accomplish low-power using means such as turning off unused portions of the circuit or lowering system clock speeds.

-M-

Machine Code: A computer program in its lowest-level form. Machine code is comprised of the 1's and 0's that the computer hardware interprets to perform the given operations specified by the program. Machine code is the only level of programming that hardware can actually understand.

Main Computer Components: The three main components of a computer include: 1) CPU, 2) the I/O, and 3) memory. The CPU is the brains/number crunching portion of the computer, the I/O allows the computer to interact with the outside world, and the memory is generally used for program and intermediate data storage.

Main Memory: The term used described the large structured memory device in a computer system. This term does not include items such as register files.

Main Task: The set of code that a program executes when it has no other tasks to attend to. The main task is often referred to as the *background task*.

Main Types of Code in Assembly Language

Programs: There are generally three main part of an assembly language program: 1) initialization code, 2) main task code, and 3) interrupt service routine code.

MCU: A common abbreviation for a microcontroller.

Mealy vs. Moore FSM Models: There are two classes of finite state machine model which are referred to as Mealy and Moore "machines", or "models". The external outputs of a Moore machine are a function of state only and output changes are thus considered to be synchronized to state changes in the FSM. The external outputs of a Mealy machine are a function of both FSM state and the internal inputs. Changes in external outputs of a Mealy machine are not necessarily synchronized to the changes in FSM state since they are also a function of external inputs.

Mealy's First Law of Digital Design: If in doubt, draw some black box diagrams.

Mealy's Second Law of Digital Design: If your digital design is running into weird obstacles that require kludgy solutions, toss out the design and start over from square one.

Mealy's Third Law of Digital Design: Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.

Mealy's Fourth Law of Digital Design: The digital design process is circular, not linear; you rarely generate the correct solution with one pass. The digital design process is circular; going back a few a few steps to fix unforeseen issues is part of the design process. Don't try to make your design perfect from the get-go, make it simple to understand so that you can fix issues as they arise.

Mealy's First and Only Law of Computer Programming: If you understand the hardware of the computer your program will run on, then you are able to write better programs than someone who does not understand the hardware.

Memory Access Time: A term referring to the amount of time require to either read data from or write data to a memory object.

Memory Bandwidth: Memory bandwidth refers to the amount of data that can be transferred to and from memory. The speed of memory reads and writes are constrained by physical attributes of the device as well as the system in which the device operates in which thusly allow for a maximum amount of information to be transferred to and from the device.

Memory Capacity: The amount of storage a given memory contains. Memory capacity is stated in various forms such as total number of bits, total number of bytes, or total number of words.

Memory Configurations: This term refers to the notion that multiple memories can be configured in ways to obtain different memory capacities (number of accessible storage elements) and different storage characteristics (the width or word-length) of each storage element.

Memory Levels: A term that encompasses the various types of memory in a given system. Generally speaking, the lower-level memories are faster but more expensive than higher-level memories. Computer system deal with a trade-off between program execution speed and expense.

Memory Mapped I/O (MMIO): One of the two forms of *programmed I/O*, characterized the instruction set using memory access instructions, such as *load & store*, to perform input and output operations, respectively.

Memory Model: A term that describes the general way a given CPU utilizes the memory resources it has at its disposal.

Memory Performance Measures: Because systems rely heavily on memory, items such as read access times, write cycle times, and memory bandwidth are used to measure the specific performance of memory devices within the system.

Memory Reading: An operation that accesses the contents of memory without changing those contents.

Memory Writing: An operation that changes the contents of memory.

Metastability: Digital circuits can become metastable when a set-up and/or hold time is not met. Metastability is a loose definition and means the circuit's output is neither high nor low and may remain in that state there for an unstated amount of time.

Microcontroller: An integrated computer system that contains a CPU, memory (program and data), and can also contain on-board peripherals.

Microoperations: A microoperation is an elementary operation performed on data stored in a register. Microoperations can also include interactions with other registers such as storing the result of microoperations associated with other circuit elements. Microoperations are commonly used in higher-level descriptions of digital circuitry such as computers.

Microprocessor: A hardware device containing a general-purpose central processing unit (CPU).

MMIO: An acronym standing for *memory mapped I/O*.

Mnemonic: A set of letters that represents a given operation. Generally speaking, mnemonics loosely describe, in an abbreviated manner, the operation they represent.

Model: A model is a representation of something. A more (definitive) descriptive description of a model is a description of something in terms that highlights the relevant information in that thing while hiding the less useful information. The purpose of a model is to quickly transfer important information to the entity reading the model (whether human, or computer, or member of the EE Faculty). Generally speaking, the quality of any model is determined by its ability to transfer information to the user.

Mono-Stable Multivibrator: A device that has one stable state; the stable state can either be the '0' or '1' state. The device's output is only in the non-stable state momentarily before transitioning to the stable state. This term is a fancy name for a device commonly referred to as a "one-shot"

Museum Quality: A clever label attached to source code that is highly pleasing to the eye of intelligent humans. Such source code is easily readable, understandable, maintainable, and modifiable.

-N-

NAFT Engineer: An acronym describing a certain type of engineering generally associated with the defense industry (speaking from personal experience); the acronym stands for *Not A F*cking Thing*. Usage: "what type of engineering do you do?" Ans: "I'm a NAFT engineer".

Narcissistic Personality Disorder (NPD): A disorder inflicting most faculty members in academia. Individuals must have this disorder in order to become a successful academic administrator.

Nesting: A term that common refers to two different items in programming of MCU. First, it refers to a subroutine that calls another subroutine (not including calling itself, which is recursion). Second, it refers to the nesting of interrupts, which is similar to the nesting of subroutines.

NOP: An acronym for "no operation", which is a common executable instruction in assembly languages used exclusively to create delays in program throughput.

Nibble: A 4-bit set of data.

Non-Maskable Interrupt: An interrupt that can't be disabled under program control.

Non-Volatile: A term that refers to an electronic device that retains its memory when power is removed and reapplied. ROMs are considered non-volatile memory devices.

-O-

Off-by-One Error: A common error where a loop iterates one too many or one too few times, typically based on the many non-intuitive ways instructions control loops and access memory. The C programming language is famous for this type of errors.

Offset: A value that modifies another value, such as a base value, to calculate a final value. For example, a base address plus an offset is used to calculate the physical address.

One-off: A solution that is specific to a given problem and won't generally apply to other problems. This includes programs, subroutine, and digital circuits.

One-Shot: The common name for a mono-stable multivibrator. One-shots are used to synthesize fixed-length pulse signals in response to signal events such as clock edges.

Ones Complement: A mathematical term referring to complementing or toggling all bits in a set of bits.

Opcod: A term that is shorthand for "operational code". Opcodes are the bits of an instruction that are used by the control unit to decode which instruction is being executed. Opcodes are constant in any given instruction whereas *field codes* are variable.

Operand: The computer code that acts as an interface between the hardware and the executing program.

Operating Context: A term that refers to the state of the processor at any given time, where state is defined by the data the given processor is storing at that given time. This term is often referred to as simply *context*.

Operating System: The computer code that acts as an interface between the hardware and the executing program.

-P-

Parity: A term that describes a characteristic of a group of bits. If an odd number of bits in the group are set, the group of bits exhibits odd parity; otherwise, the set of bits exhibits even parity (even number or zero bits set).

Parity Bit: A bit that describes the parity of a given set of bit, where typically '1' represents odd parity and '0' represents even parity.

Passed Value: A value that is provided to a subroutine.

Pig: A term that completely describes academic administrators, though the term can be insulting to our actual porcine friends.

Persistence of Vision: An alternative term for *retinal persistence*, which is a characteristic of the human vision system utilized by some electronic devices.

Physical Address: An value that appears on the address lines of a memory device such as a RAM or ROM.

Physical Memory: The actual amount of memory present in a given system.

Polling: Processors use polling to interface with external devices where the process constantly evaluates the status of the external device in order to determine if the device is in need of services from the processor. Polling is considered to be used in "programmed I/O" and is one of three major types of computer related I/O. Polling is generally associated with inefficient embedded system design in that the system is considered to have low overall throughput when executing a polling loop

Pop: An operation associated with stacks where an item is removed from a stack; the stack pointer is appropriately adjusted.

Port: A generic location in a computer system's address space.

Port Address: The numeric value associated with an external input or output device used by I/O instructions. The I/O instructions use these values to differentiate and communicate with various external peripheral devices.

Port Mapped I/O (PMIO): One of the two forms of *programmed I/O*, characterized by dedicated instructions in the instruction set for performing data input and output.

Princeton Architecture: A computer architecture where data and instructions share the same memory space. This architecture is also known as a Von Neumann architecture.

Processor: A generic term that generally refers to some type of device that does processing such as a microcontroller or microprocessor.

Program: Noun: a complete set of software that can be in different forms such as a listing or machine code. Verb: the act of writing text that can be used to control a computer.

Program Memory: The part of a computer system memory that stores the machine code for programs the computer is running.

Program Counter (PC): The program counter is a simple counter generally found in a computer's control unit and whose output is generally used as an address that points to the next instruction in program memory to be executed by the program. The PC is typically expected to do standard counter microoperations such as parallel load and increment.

Program Flow Control Instructions: Instructions that cause or potentially cause the CPU to execute an instruction other than the instruction following the current instruction. Examples of program flow control instructions are conditional/unconditional branches, and subroutine calls/returns.

Program Flow Control: For computer programs to do useful things, they must appropriately respond accordingly to important "events". This response at a low level includes executing different portions of the given computer program. Computer instructions that facilitate any computer operation other than simple incremental execution of instructions from the program memory are generally referred to as program flow control instructions. Program flow control is generally handled by clever manipulations of the program counter.

Programmable Logic Device (PLD): Any integrated circuit used to create circuits in which the functionality of the internal circuit is not defined until the device is programmed (in this context, the term “program” does not typically refer to a computer programming language). One common type of PLD is the FPGA.

Programmed I/O: One of three main forms of computer I/O. The two subtypes of programmed I/O include *port mapped I/O (PMIO)* and *memory mapped I/O (MMIO)*. The other forms of I/O include Interrupt I/O and Direct Memory Access.

Programming Language Levels: Computer programs can be written on one of three general levels (listed from low to high): machine code level, assembly code level, or higher-level. Higher-level languages include C, C++, C#, Java, Wanker, etc.

Programming Model: The programming model, or *programmer’s model*, describes the hardware resources available on a programmable computer-type device that the programmer is able to control via the program control. Program control is provided by the operations described by the device’s instruction set and can either be categorized as software or firmware.

Pseudocode: An approach to modeling programs that looks somewhat similar to the actual programming code; used often in the design of computer programs. .

Pseudoinstruction: An instruction that you can use with the RISC-V MCU, but that instruction is not actually implemented in hardware; pseudoinstructions are instead implemented by the assembler with one or more base instructions.

Pure Programmer: A programmer who knows how to program using a particular language, but knows nothing about digital hardware, or particularly, the digital hardware the program they write will execute on.

Push: An operation associated with stacks where data is placed onto a stack; the stack pointer is appropriately adjusted.

-Q -

Q: The letter commonly used to represent state variables when working with FSMs and flip-flops.

Q⁺: The symbology commonly used to represent the next state when working with FSMs.

Quiz-rama: A word I made up describing having many quizzes instead of few mid-term exams.

Quizster: A person who loses sleep looking forward to and/or preparing for quizzes.

-R-

ra: A term the RISC-V MCU documentation uses to refer to the *return address*.

Radix: A term describing the number of symbols in the symbol set associated with a given number system.

Radix Point: The symbol, typically a small dot low in the symbol field, that separates the integral and fractional portions of a number.

RAM: The acronym officially stands for Random Access Memory; a solid definition for RAM is fleeting due to advances in technology. RAMs are most often characterized as volatile, random access storage devices.

Random Access: A memory device is considered random access if it can access any of its contents in a constant amount of time. Devices such as flash drives are considered random access while devices such as tape drives and hard drives are not random access.

rd: A term that RISC-V documentation uses to refer to a generic destination register.

Read: A term referring to copying data from a memory device to another location without changing the value in memory.

Read Access Time: The amount of time required for memory output data to become available after an address and the correct control signals have been provided to the device.

Read Only Memory: A memory device roughly meaning that you can only read from it and not write to it. The accepted definition is a memory that is readable, but not writeable, and is non-volatile.

Real Time System: A computer system that has deadlines (response time to events) that must be met in order for the system to work properly. Interrupt-driven systems are after referred to as real-time systems because programmers can leverage the interrupt architecture to reduce response time.

Recursion: The notion of a subroutine calling itself. The “depth of recursion” refers to the number of times a subroutine calls itself before commencing returning from the subroutine calls.

Reduced Instruction Set Computer: This acronym officially stands for “Reduced Instruction Set Architecture” and is generally used to describe computer architectures. In actuality, the term has little or nothing to do with the size of the instruction set. RISC architectures generally have the following characteristics:

- The architecture contains a register file with many general purpose registers
- The instructions word formats all contain the same number of bits (no extended opcodes)
- The instructions are executed in the same number of clock cycles
- The instructions generally are not overly complicated (meaning they don’t generate great amounts of processing within the architecture)

They have higher system clock frequencies than non-RISC architectures

Register File: An abstract device that is used to model a given number of general purpose registers that are directly accessible by the given computers instruction set. Register files are typically modeled as multiport RAMs that can read and/or write multiple registers, roughly speaking, in a simultaneous manner.

Register Transfer Language (RTL): A syntactically loose approach to specifying a digital circuit that can be modeled as the synchronous transfer of data between sequential circuits such as registers. A RTL statement generally describes a microoperation (or set of micro-operations) generally associated with a digital circuit. The two parts of an RTL statement are 1) the register transfer specification, and 2) the specific conditions that are necessary for that transfer to occur. Generally speaking, only signals necessary for the stated transfer to occur are listed in the RTL statement while non-listed signals are assumed to be “properly handled” elsewhere. Each RTL statement is assumed to occur in one clock cycle. RTL is also known as *register transfer notation* (RTN).

Register: An n-bit wide sequential circuit that is primarily known for its ability to store bits. Registers are generally modeled as “n” D flip-flops, which share a common clock. Register generally have synchronous parallel load inputs and sometimes other features (elementary operations) such as asynchronous or synchronous presets and clears. Specialized registers include shift registers and counters.

Retinal Persistence: The notion associated with the human visual system that does not allow humans to perceive an off state of an LED at the exact time the LED is turned off. The notion of retinal persistence is what allows display multiplexing to work for humans.

Return Address: The address of an instruction that is the next instruction to execute after a returning from a subroutine or interrupt service routine.

Return Value: A value that is *returned*, or passed from a subroutine back to the part of the program that called the subroutine.

RISC vs. CISC: The age-old computer argument of which is better that has never been solved. Generally speaking, RISC architectures require more instructions to complete a given operation than a CISC architecture would for that same operation, but those instructions are executed “more quickly” than a CISC architecture.

RISC: This acronym for *reduced instruction set computer*”; see *reduced instruction set computer*.

ROM: The acronym officially stands for Read Only Memory

rsx: An abbreviation used by the RISC-V literature to indicate a numbered source register, such as *rs1* or *rs2*.

Round-Up: The act of adjusting a value to better reflect the parts of a value that was *truncated*. Numbers are typically rounded up if the truncated value is 0.5 or greater, which is digital is the weighting of the first bit to the right of the radix point.

Run Time: A term that refers to the active of running a program, as opposed to compile time, which is an important term but necessarily occurs before a program is actually run.

Run Time Complexity: A term that refers to the amount of time required to run a section of code, a program, or an algorithm; this term is closely related to *computational complexity* in that more complex coder requires more time to execute.

Running Time: A term that refers to the amount of time required for a program, section of code, or algorithm to execute in hardware. Running time can be measured in time units or other time-related metrics such as rate of instruction execution.

-S-

Self-serving: The defining characteristic of all academic administrators and most engineering faculty.

Set: The act of making a bit into a '1'.

Sequential Circuit: A circuit whose output is a function of the sequence of the circuit's inputs. Another common definition is a circuit that has *state*, meaning it can store data.

Set-up & Hold Times: Digital devices that are edge sensitive (circuit changes state on a rising or falling clock edge) must hold inputs stable (the inputs must not change state) for a certain amount of time before the active clock edge arrives; this time is referred to as the *set-up* time. Digital devices must also hold the inputs stable for a certain amount of time after the active clock edge which is referred to as the *hold time*. Failing to meet set-up and/or hold times leads to the circuit going *metastable*.

Shift Register: A special flavor of register designed to perform contiguous bit-level transfers (or serial transfers) of data between the bit storage elements of the register. Shift registers generally shift all the storage elements to a contiguous storage element once per clock cycle.

Simulator: A piece of software that outputs the expected output from another piece of software or a model of a digital circuit.

Signed Extension: A term associated with expanding the bit-width of *signed data* by adding extra bits to the left side of the data and setting all the added bits to the value of the original sign bit.

Softcore MCU: An MCU that has been or will be synthesized on a programmable logic device.

Signed Value: A set of bits or other numbers that is to be interpreted as either negative, zero, or positive.

Software: In the specific case, software is a computer program that is written in a generic way so that it can run on a more than one type computer. Software does not refer to the language-level in which the program is written and thus can be written in machine code, assembly code, or a higher-level language. In the less specific case, the term software is often means any code written to run on a computer.

Source Operand: The term used in assembly languages to describe where an instruction gathers data from, such as from main memory or a register.

sp: A shorthand notion for *stack pointer*.

Space Efficient: Refers to the storage requirements for a given program. If two programs are functionally equivalent, the program that uses less program memory is the more space efficient program. The more space efficient program may not be more *time efficient* than that other program.

Spaghetti code: Programming code that does not follow standard structured programming concepts. Spaghetti code is by definition fragile; it is hard to understand, maintain, modify, and reuse.

Stack pointer: A term that refers to an entity that contains information that describes the "top of the stack".

Stack: An abstract data type that implement a last-in/first-out (LIFO) queue (or list of things). Stacks can be implemented in hardware or software with hardware implementation of stacks employing the use of a stack pointer to increase efficiency of the device. Stacks are typically used in computer architectures to keep track of hierarchically nested processes such as subroutines and interrupts.

Stack Segment: The part of the main memory dedicated to the system's stack operations.

Stack Overflow/Underflow: Stacks require a given amount of memory space. Stacks operations (pushes and pops) can cause the stack beyond that memory space which results in overflow or underflow conditions. Overflow and underflow can overwrite important information such as program memory and data memory.

Standard Decoder: A standard decoder is a hardware device that implements a *one-hot* or *one-cold* output based on a given set of inputs. There is typically a binary relationship between the number of select inputs and the number of outputs and come in such flavors as 1:2, 2:4, 3:8, etc.

Start-up code: The code that is inserted automatically by the assembler as a result of declaring data in the program that requires initialization. The start-up code is typically comprised of instructions that initialize data memory.

State: The currently value(s) being stored by a sequential digital circuit.

Store: A common term that refers to the act of writing a value to memory. This term is most often synonymous with *write*.

Structured Code: Code that can be decomposed into three basic structures: 1) sequence, 2) if-then-else, and, 3) iterative. Structured code is easily understood, maintained, modified, and reused.

Stupathetic: A term used to describe people who are both stupid and apathetic; we all know who they are.

Structured Memory: A term used to describe relatively large semiconductor memories, such as a RAM or ROM. Structured memory does not include distributed registers in a circuit. The notion of "structure" is derived from regular structures, which have a repeatable pattern on the silicon die.

Subroutine: A set of instructions that a computer explicitly transfers to and returns from. In terms of program flow, the program transfers program execution to a set of instructions referred to as the subroutine. When the instructions in the subroutine have completed executing, control is returned to the instruction after the instruction, which caused the program to initially transfer to the subroutine.

SWAG System: An acronym standing for *scientific wild ass guess*, which can be the first step in getting something done when you know nothing. However, it sure sounds good when you use the term because most people are afraid to ask what it means.

Switch Bounce: A condition associated with all mechanical switches where upon actuation, the switch contacts make and break connections several times before the "settling" to the connected state. Switch bounce can last up to 20ms, depending on what source you consult.

Synchronous: The process of converting a circuit described using an HDL model into a gate-level representation of that circuit.

Synthesis: The process of converting a circuit described using an HDL model into a gate-level representation of that circuit.

System Verilog: System Verilog is one of several modeling systems referred to as "hardware description languages", or HDLs. System Verilog is a superset of Verilog and contains a rich set of programming-like functionality to support the use of System Verilog as a circuit verification tool. Most of the added functionality is not synthesizable.

-T-

Tab: A shorthand approach to indentation in source code that astute programmers and designers never use because different editors and printer interpret them differently and can make your code look like garbage.

Task: Typically a set of operations that “need doing”. This term has a richer definition in the context of real-time operating systems.

Test Vector: A set of numbers used by a simulator to verify a given hardware design; can be machine generated.

Three State: This is an alternative term for *tri-state*, where the issue here is that someone had a trademark on one of the terms.

Throughput: The throughput of a system is the total amount of useful information processed or communicated during a specified time period. Note that this definition is really general. Systems with high throughput are generally desired over systems with low throughput with the exception of administrative systems on university campuses.

Time Efficient: Refers to the running time, or the time to generate a meaningful result, for a given program. If two programs are functionally equivalent, the program that generates the result in a shorter amount of time is the more time efficient program. The more time efficient program may not be more *space efficient* program.

Timing Diagram: An illustration that provides digital signal values as a function of time, where the vertical axis is represents the digital value and the horizontal axis represents time.

Tool Chain: The set of software programs that allow humans to go from a concept to a working system, typically an embedded system. These tools typically include assemblers and/or compilers, linkers, and a mechanism to insert programs onto hardware.

Top-of-stack: A term that generally refers to the more recent item placed onto a stack. The *stack pointer* typically points to the top of the stack.

Tri-State: A term that refers to a devices ability to effectively remove itself from a circuit. Thus, a tri-state device in a digital circuit can either be high, low, or high-impedance. The notion of tri-stating is used to share routing resources in a circuit; the only possible drawback of tri-stating is that only one device can drive the resource at a given time, otherwise the condition of contention occurs, which is ungood.

Truth Table: A matrix that displays all possible input and output values for a given Boolean equation or digital circuit.

Truncation: The act of removing part of something. In digital design, we often chop off lower bits of value such as in shift operations. The chopped bits are lost, with no possibility of *round-up*.

Two’s Complement: A common representation for signed binary numbers. Additionally, taking the “two’s complement” of a number is equivalent to taking the *one’s complement* of the number and adding one to the result.

-U-

Universal Shift Register: A special flavor of shift register that performs actions other than simple one-directional shifts including some or all of the following operations: shift left, shift right, barrel shifts, arithmetic shift, and rotates.

Unsigned Value: A set of bits or other numbers that is to be interpreted as a zero or positive value.

-V-

Verification: The act of testing HDL models to discern the correctness before the models are synthesized. Verification is thus one form of testing.

Verilog: Verilog is one of several modeling systems referred to as “hardware description languages”, or HDLs. Verilog is typically used to model digital circuits; the resultant models can be used to simulate circuits, or synthesize circuit implementations on PLDs or silicone. Verilog uses a “C-like” syntax, which makes it popular with software-type people.

VHDL (Very High Speed Circuit Hardware Description Language): VHDL is one of several modeling systems referred to as “hardware description languages”, or HDLs. VHDL is typically used to model digital circuits; the resultant models can be used to simulate circuits, or synthesize circuit implementations on PLDs or silicone.

Volatile/Non-Volatile: A device is considered volatile if its contents are lost when power is removed from the device while non-volatile devices retain their memory when power is removed and subsequently returned. The term volatile is most often associated with memory devices and PLDs such as FPGAs.

Von Neumann Architecture: A computer architecture where data and instructions share the same memory space. The term Von Neumann machine is often used to mean Von Neumann architecture. Von Neumann architecture is sometimes referred to as a “Princeton” architecture.

-W-

While Loop: An iterative programming construct characterized by the condition to continue the iteration is checked before performing the first iteration.

White Space: Empty space in source code files including indentation (not tabs), blank lines, and extra spaces (such as to align parts of the listing). White space is used by good programmers and hardware designers.

Wrapper: A term used to provide a higher-level interface to a circuit. The wrapper circuit is thus a superset of the circuit it *wraps*.

Write Cycle Timing: The amount of time required for data to be written to memory after a valid address, valid input data, and the appropriate control signals have been provided to the device.

-X-

X: The symbol typically used to represent input variables in finite state machines.

XOR: A common shorthand notion for *exclusive OR*.

-Y-

Y: The symbol typically used to represent state variables in finite state machines.

-Z-

Z: The symbol typically used to represent high impedance. This symbol is also used to represent output variables state machines.

Zero Extension: A term associated with expanding the bit-width of *unsigned data* by adding extra bits to the left side of the data and clearing those bits.

INDEX

6

68000 · - 21 -

A

ABI · *See* Application Binary Interface
 absolute address · - 461 -
 absolute addresses · - 473 -
abstract data types · - 292 -
 academic administrator · - 180 -
 accumulator · - 46 -
 active clock edge · - 47 -
 address of the ISR · - 517 -
 addressing modes · - 281 -
ADTs · *See* abstract data types
algorithm · - 176 -
 alignment · - 190 -
 alternative register names · - 196 -, - 202 -
 ALU · - 161 -
 antiquated · - 161 -
 Application Binary Interface · - 196 -
 architecture · - 159 -
 Arithmetic Logic Unit · - 161 -
 arithmetic shift · - 74 -
 arithmetic shifts · - 73 -
 array · - 366 -
 arrow · - 55 -
 assemble time · - 253 -, - 280 -
 assembler · - 160 -, - 170 -, - 171 -
 assembler directive · - 222 -, - 428 -
 assembler directives · - 189 -
assembly language · - 170 -

B

background task · - 331 -
 bad press · - 171 -
 bag of tricks · - 229 -
 barrel shift · - 72 -
 barrel shifts · - 478 -
 base address value · - 280 -
 base instruction · - 203 -
 base instructions · - 194 -
 Basys3 · - 483 -
BFD · *See* Brute Force Design
 bi-directional · - 68 -
 bit banging · - 265 -
 bit crunching · - 229 -
 bit manipulations · - 194 -
 bit masking · - 174 -

bit-addressable · - 115 -
bit-masks · - 265 -
 bit-tweaking · - 265 -
 bit-twiddling · - 163 -
 bit-wise · - 230 -
 black block diagram · - 176 -
 bottleneck · - 119 -
 bounce · - 534 -
 brain dump · - 560 -
 brains · - 159 -
 branch · - 194 -, - 221 -
 branches · - 245 -
BRUTE FORCE DESIGN · - 29 -
 bulletproof · - 305 -
 bulletproof code · - 182 -
 bus contention". · - 67 -
 butthead friends · - 181 -

C

C programming language · - 248 -, - 505 -
call · - 500 -
 cascadeability · - 53 -
 Cascadeable · - 47 -
 case structure · - 179 -
 Central Processing Unit · - 161 -
CIRCUIT CONTROL · - 29 -
 Circular · - 47 -
 CISC · - 531 -
 clearing · - 265 -
 clever · - 265 -
 code segment · - 349 -
 co-design · - 158 -
 code-word · - 47 -
 Combinatorial · - 32 -
 Combinatorial circuits · - 36 -
 Comments · - 188 -
 common cathode · - 539 -
 comparator · - 43 -
 compiler · - 160 -, - 171 -
 Complex Instruction Set Computer · - 531 -
 complex programs · - 175 -
 computationally expensive · - 72 -
 computer · - 21 -, - 158 -
 computer *architecture* · - 159 -
computer architectures · - 170 -
 computer language · - 160 -
 computer peripherals. · - 30 -
 computer program · - 160 -
 computer programmer · - 159 -
 computer user · - 159 -
 computerland · - 230 -
 computersaureses · - 171 -
 conditional branch · - 248 -
 conditional branches · - 245 -

conditionally · - 221 -
 constant · - 223 -, - 253 -
 control signals · - 57 -
 control unit · - 161 -
 Control Unit · - 161 -
 cost effective · - 176 -
 Count Enable · - 47 -
 counter · - 46 -
 Counter Overflow · - 47 -
 Counter Underflow · - 47 -
 CPU · - 161 -
 cross-coupled NAND cell · - 44 -
 cross-coupled NOR · - 44 -
 crunch data · - 159 -
 CU_DCDR · - 445 -
 CU_FSM · - 446 -
 cursory glance · - 181 -

D

data · - 112 -
 data memory · - 461 -
 datapath · - 533 -
 dead gate · - 515 -
 debouncer · - 535 -
 debugger · - 351 -
 debugging · - 180 -
 Decision · - 178 -
 decision point · - 179 -
 decode/execute cycle · - 448 -
 decoder · - 38 -
 Decrement · - 47 -
 design libraries · - 34 -
destination register · - 85 -, - 197 -, - 202 -, - 498 -
 deterministic · - 545 -
 development board · - 483 -, - 484 -
 digital bag of tricks · - 27 -
 Digital tricks · - 174 -
 direct memory access · - 214 -
Direct Memory Access · - 214 -
 DMA · *See* direct memory access, *See* direct memory access
 don't care · - 58 -
 do-while loop · - 179 -
 do-while loops · - 253 -
 Down Counter · - 47 -
 drive the bus · - 67 -
 driving the bus · - 67 -
 dumb loop · - 327 -
 dumb loops · - 538 -

E

eloquence and beauty · - 181 -
 embedded system · - 190 -, - 197 -, - 328 -
 enable interrupts · - 333 -

endless loop · - 190 -
 entry point · - 179 -
 ESX MCU · - 22 -
 execute cycle · - 446 -, - 448 -
 EXTERNAL CONTROL · - 29 -
 external interrupts · - 327 -

F

FA · *See* full adder
 fast multiplication · - 74 -
 feature creep · - 182 -
 feature set · - 53 -
 fetch cycle · - 446 -, - 448 -
 fetching · - 446 -
 field codes · - 498 -
 file banner · - 190 -, - 222 -
 file header · - 190 -
 filtering · - 536 -
 finite state machine · - 30 -, - 446 -
 Finite State Machine · - 53 -
 firmware · - 176 -, - 197 -
 flag register · - 336 -
 Flip-flops · - 33 -
 floating point numbers · - 72 -
 flow arrows · - 178 -
 flowchart · - 175 -
 foreground task · - 331 -
 forward slash · - 57 -
 FPGA · - 483 -
 fragile · - 173 -
 Full Adder · - 37 -
function · - 297 -
Functionally Complete · - 32 -

G

general purpose · - 197 -
 general purpose registers · - 195 -
 General-Purpose Computer · - 187 -
 general-purpose register · - 194 -
 generic decoder · - 38 -
 Generic Decoder · - 39 -
 Good-looking code · - 181 -

H

HA · - 36 -, *See* half adder
 Half Adder · - 36 -
 handles · - 329 -
 hard drives · - 113 -
 hardcoded to 0 · - 196 -
 Harvard architecture · - 532 -
 header · - 222 -
 hierarchical · - 165 -

higher-level language · - 171 -
 high-impedance · - 64 -
 hi-Z · - 66 -
 HLL · See higher-level language
 holding · - 265 -
 Hold-times · - 32 -
 human visual system · - 538 -

I

I/O · - 159 -
 if/else constructs · - 174 -
 if-then-else construct · - 175 -
 if-then-else structure · - 179 -
IMD · See Iterative Modular Design
 immediate instruction · - 206 -
 in-case-of · - 179 -
 incidental memory · - 112 -
 Increment · - 47 -
 information · - 112 -
 information content · - 112 -
 Input/Output · - 159 -
 Input/Output architecture · - 174 -
 instruction memory · - 161 -
 instruction set · - 162 -
Instruction Set · - 172 -
 instruction set architecture · - 186 -
 instructions cycles · - 448 -
 integer-based math · - 72 -
INTERNAL CONTROL · - 29 -
 internal interrupts · - 327 -
 interrupt architecture · - 327 -, - 329 -
Interrupt architecture · - 174 -
 interrupt cycle · - 513 -
 interrupt enable · - 332 -, - 515 -
 interrupt mask bit · - 333 -
 interrupt nesting · - 334 -
 interrupt service routine · - 329 -, - 512 -
 interrupt vector · - 517 -
 interrupt vector address · - 517 -
 ISA · - 186 -, See instruction set architecture
 ISR · See interrupt service routine
 iterative construct · - 175 -
 iterative design · - 37 -
 iterative modular design · - 43 -
ITERATIVE MODULAR DESIGN · - 29 -
 iterative structure · - 179 -

J

job security · - 173 -, - 181 -
 jump · - 221 -
 jump and link register · - 454 -
 jumps · - 245 -

K

kludgy · - 28 -, - 597 -
 knarly · - 208 -

L

Labels · - 189 -
Last In, First Out · - 293 -
 Latches · - 33 -
 lazy professors · - 181 -
 lead-zero blanking · - 539 -
 level sensitive · - 44 -
 levels of memory · - 533 -
LIFO · - 293 -
 link · - 454 -
 look-up-table · - 505 -
 loop overhead · - 359 -
 loops · - 253 -
 low power · - 330 -
 low-power mode · - 328 -
 LUT · See look up table
 LUTs · - 182 -
 LZB · See lead zero blanking

M

machine code · - 162 -, - 170 -
machine language · - 162 -, - 170 -
 main memory · - 348 -
 masked · - 333 -
 maybe · - 250 -
 MCU · See microcontroller
 Mealy · - 32 -
 memory · - 159 -
 Memory · - 193 -
 memory address space · - 349 -
 memory elements · - 54 -
 memory map · - 214 -
 memory mapped · - 194 -
 memory mapped I/O · - 213 -
 messages from the programmer to the assembler · - 189 -
 -
 messages to assembler · - 189 -
 messages to humans · - 188 -
method · - 297 -
 Microcomputer · - 22 -
 microcontroller · - 30 -
 Microcontroller · - 22 -
mnemonics · - 170 -
 model · - 159 -
 modular code · - 182 -
MODULAR DESIGN · - 29 -
 monstable multivibrator · - 536 -
 Moore · - 32 -

mtvec · - 333 -
 multiplexor · - 42 -
 MUX · - 42 -, *See* multiplexor

N

n-bit Counter · - 47 -
next state · - 54 -
 Next State · - 55 -
next state decoder · - 54 -
 Next State Decoder · - 53 -
next state forming logic · - 54 -
 next state logic · - 54 -
No CONTROL · - 29 -
 non-normal operation · - 243 -
 non-volatile · - 113 -
 normal operation · - 243 -

O

off-page connection · - 178 -
 offset value · - 280 -
 off-the-shelf · - 30 -
 Ohm's Law · - 64 -
 old guys · - 21 -
 one-off · - 158 -
 one-shot · - 536 -
 opcodes · - 498 -
 operands · - 194 -
 ort addresses · - 484 -
 oscilloscope · - 534 -
 Output decoder · - 53 -
Output Decoder · - 54 -
 overhead · - 361 -

P

Parallel Load · - 47 -
 peripherals · - 327 -
 physical memory · - 461 -
 PicoBlaze2 · - 22 -
 PicoBlaze3 · - 22 -
pipeline · - 533 -
 PLD · - 483 -, *See* Programmable Logic Device
 pointers · - 390 -
 polling · - 213 -, - 327 -
 polling loop · - 327 -, - 328 -
portable · - 171 -
 Predefined Process · - 178 -
 preprocessor directive · - 428 -
 Present State · - 55 -
 Process · - 178 -
processor · - 159 -
 Processor · - 161 -
 profiler · - 172 -

program · - 160 -
 program control · - 245 -
 program counter · - 193 -, - 195 -
 program flow control · - 221 -, - 243 -, - 454 -
 program memory · - 349 -
 programmable logic device · - 174 -
 Programmed I/O · - 212 -
 programmers model · - 162 -
Programmers Model · - 172 -
 programming efficiency · - 358 -
 Programming Model · - 192 -
 programming style · - 176 -
 Pseudo code · - 175 -
pseudoinstruction · - 202 -, - 203 -
 pseudoinstructions · - 194 -
 psychic · - 182 -

R

ra · - 454 -
 RAM · - 113 -
 random access memory · - 113 -
 RAT · - 22 -
 RCA · *See* ripple carry adder
 read only memory · - 113 -
 real-time clock · - 327 -
 real-time programming · - 328 -
recursion · - 311 -
recursive subroutine call · - 311 -
 Reduced Instruction Set Computer · - 531 -
reg file · - 201 -
register · - 54 -
 register addressing · - 206 -
 register file · - 474 -
 relative addresses · - 473 -
 replacement operator · - 86 -
 restoring context · - 304 -
 retinal persistence · - 538 -
 retirement · - 21 -
return · - 298 -
 return address · - 306 -, - 454 -, - 519 -
 ripple carry adder · - 37 -
 Ripple Carry Out · - 47 -
 RISC · - 531 -
 RISC vs. CISC · - 531 -
 robot · - 180 -
 robot grader · - 181 -
 ROM · - 113 -
 rotates · - 73 -
 run time · - 253 -
 running · - 160 -
 runtime · - 280 -

S

saving context · - 304 -

segments · - 348 -
 self-loop · - 55 -
 sequence construct · - 175 -
 sequence of code words · - 46 -
 sequence structure · - 178 -
 Sequential circuits · - 44 -
Sequential Circuits · - 32 -
 sequential execution · - 194 -
 setting · - 265 -
 Set-up · - 32 -
 seven-segment display · - 537 -
 shift register · - 49 -
 shift register cell · - 49 -
 sign extended · - 479 -
 sign extension · - 462 -
 signed offset · - 457 -
 signedness · - 74 -
 Simple code · - 181 -
 simple registers · - 44 -
 single purpose · - 158 -, - 197 -, - 359 -
 societal norms · - 327 -
 softcore MCUs · - 483 -
 software · - 176 -, - 197 -
 software-based interrupts · - 327 -
 software-land · - 292 -
 solid-state drives · - 534 -
source register · - 85 -, - 202 -, - 498 -
source registers · - 197 -
 space efficient · - 204 -, - 233 -, - 270 -, - 295 -
 spaghetti code · - 175 -
 Specific Purpose Computer · - 187 -
 spiritually enlightening · - 251 -
 stack pointer · - 293 -
 standard decoder · - 38 -
 Standard Decoder · - 40 -
 state · - 53 -
 state bubble · - 57 -
 state diagram · - 53 -
state registers · - 54 -
 State Registers · - 53 -
 state transition · - 55 -
 state transition arrow · - 55 -
state variables · - 54 -
 status signals · - 54 -
 stop running · - 190 -
 structured code · - 175 -
 structured memories · - 533 -
 structured memory · - 112 -
 structured programming · - 178 -, - 249 -
 Structured programs · - 181 -
 style file · - 182 -
subroutine · - 296 -
 subroutine call · - 194 -
 switch bounce · - 534 -
 symbology · - 59 -
 synthesizing · - 536 -
 system clock · - 55 -

T

T cycles · - 446 -
 tape drives · - 113 -
task code · - 331 -
 terminal · - 178 -
 testing · - 180 -
 text editor · - 160 -
 three-state · - 64 -
throughput · - 330 -
 time delay · - 538 -
 time efficient · - 204 -, - 295 -
 time slots · - 54 -
 toggling · - 265 -
 toolchain · - 174 -
 top of the stack · - 293 -
 transition · - 55 -
 tricks · - 229 -
 tricky code · - 182 -
 tri-state · - 64 -
 tri-state register · - 66 -
 twiddles · - 158 -

U

Unconditional branch · - 194 -
 unconditional branches · - 245 -
 unconditionally · - 221 -
 Understandable code · - 181 -
 universal shift register · - 70 -
 unmasked · - 333 -
 Up Counter · - 47 -
 Up/Down Counter · - 47 -

V

variable · - 253 -
 vector address · - 332 -
 vectors · - 332 -
 vernacular · - 195 -
 volatile · - 113 -
 Von Neumann architecture · - 532 -
 voodoo · - 565 -

W

wacky instructor · - 560 -
 while loop · - 179 -
 while loops · - 253 -
 white space · - 182 -
 whitespace · - 190 -
 word · - 115 -
 working register · - 281 -
 wrapper · - 483 -

write enable · - 486 -
write pulse · - 486 -
writeback cycle · - 446 -, - 448 -

Z

zero extension · - 462 -