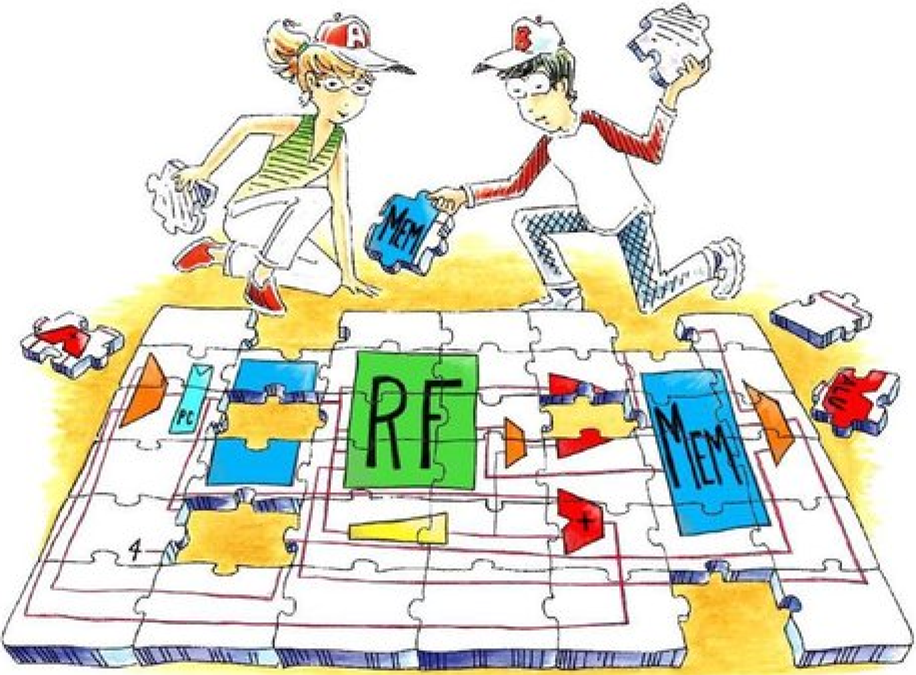
Digital Design and Computer Architecture

RISC-V Edition

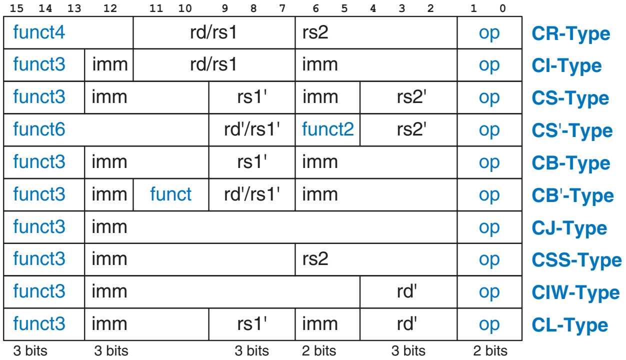


MORGAN KAUFMANN

Sarah L Harris
David Money Harris

Table B.4 **Register names and numbers**

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers



15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	CR-Type
funct3	imm	rd/rs1		imm		op	CI-Type
funct3	imm		rs1'	imm	rs2'	op	CS-Type
funct6			rd'/rs1'	funct2	rs2'	op	CS'-Type
funct3	imm		rs1'	imm		op	CB-Type
funct3	imm	funct	rd'/rs1'	imm		op	CB'-Type
funct3	imm					op	CJ-Type
funct3	imm			rs2		op	CSS-Type
funct3	imm				rd'	op	CIW-Type
funct3	imm		rs1'	imm	rd'	op	CL-Type
3 bits	3 bits		3 bits	2 bits	3 bits	2 bits	

Figure B.2 **RISC-V compressed (16-bit) instruction formats**

Table B.5 **RVM: RISC-V multiply and divide instructions**

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply	`rd = (rs1 * rs2)`$_{31:0}$
0110011 (51)	001	0000001	R	`mulh rd, rs1, rs2`	multiply high signed signed	`rd = (rs1 * rs2)`$_{63:32}$
0110011 (51)	010	0000001	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned	`rd = (rs1 * rs2)`$_{63:32}$
0110011 (51)	011	0000001	R	`mulhu rd, rs1, rs2`	multiply high unsigned unsigned	`rd = (rs1 * rs2)`$_{63:32}$
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed)	`rd = rs1 / rs2`
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned	`rd = rs1 / rs2`
0110011 (51)	110	0000001	R	`rem rd, rs1, rs2`	remainder (signed)	`rd = rs1 % rs2`
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned	`rd = rs1 % rs2`

Table B.6 **RVC: RISC-V compressed (16-bit) instructions**

op	instr$_{15:10}$	funct2	Type	RVC Instruction	32-Bit Equivalent
00 (0)	000–––	–	CIW	`c.addi4spn rd', sp, imm`	`addi rd', sp, ZeroExt(imm)*4`
00 (0)	001–––	–	CL	`c.fld fd', imm(rs1')`	`fld fd', (ZeroExt(imm)*8)(rs1')`
00 (0)	010–––	–	CL	`c.lw rd', imm(rs1')`	`lw rd', (ZeroExt(imm)*4)(rs1')`
00 (0)	011–––	–	CL	`c.flw fd', imm(rs1')`	`flw fd', (ZeroExt(imm)*4)(rs1')`
00 (0)	101–––	–	CS	`c.fsd fs2', imm(rs1')`	`fsd fs2', (ZeroExt(imm)*8)(rs1')`
00 (0)	110–––	–	CS	`c.sw rs2', imm(rs1')`	`sw rs2', (ZeroExt(imm)*4)(rs1')`
00 (0)	111–––	–	CS	`c.fsw fs2', imm(rs1')`	`fsw fs2', (ZeroExt(imm)*4)(rs1')`
01 (1)	000000	–	CI	`c.nop (rs1=0,imm=0)`	`nop`
01 (1)	000–––	–	CI	`c.addi rd, imm`	`addi rd, rd, SignExt(imm)`
01 (1)	001–––	–	CJ	`c.jal label`	`jal ra, label`
01 (1)	010–––	–	CI	`c.li rd, imm`	`addi rd, x0, SignExt(imm)`
01 (1)	011–––	–	CI	`c.lui rd, imm`	`lui rd, {14{imm`$_5$`}, imm}`
01 (1)	011–––	–	CI	`c.addi16sp sp, imm`	`addi sp, sp, SignExt(imm)*16`
01 (1)	100–00	–	CB'	`c.srli rd', imm`	`srli rd', rd', imm`
01 (1)	100–01	–	CB'	`c.srai rd', imm`	`srai rd', rd', imm`
01 (1)	100–10	–	CB'	`c.andi rd', imm`	`andi rd', rd', SignExt(imm)`
01 (1)	100011	00	CS'	`c.sub rd', rs2'`	`sub rd', rd', rs2'`
01 (1)	100011	01	CS'	`c.xor rd', rs2'`	`xor rd', rd', rs2'`
01 (1)	100011	10	CS'	`c.or rd', rs2'`	`or rd', rd', rs2'`
01 (1)	100011	11	CS'	`c.and rd', rs2'`	`and rd', rd', rs2'`
01 (1)	101–––	–	CJ	`c.j label`	`jal x0, label`
01 (1)	110–––	–	CB	`c.beqz rs1', label`	`beq rs1', x0, label`
01 (1)	111–––	–	CB	`c.bnez rs1', label`	`bne rs1', x0, label`
10 (2)	000–––	–	CI	`c.slli rd, imm`	`slli rd, rd, imm`
10 (2)	001–––	–	CI	`c.fldsp fd, imm`	`fld fd, (ZeroExt(imm)*8)(sp)`
10 (2)	010–––	–	CI	`c.lwsp rd, imm`	`lw rd, (ZeroExt(imm)*4)(sp)`
10 (2)	011–––	–	CI	`c.flwsp fd, imm`	`flw fd, (ZeroExt(imm)*4)(sp)`
10 (2)	1000––	–	CR	`c.jr rs1 (rs1≠0,rs2=0)`	`jalr x0, rs1, 0`
10 (2)	1000––	–	CR	`c.mv rd, rs2 (rd ≠0,rs2≠0)`	`add rd, x0, rs2`
10 (2)	1001––	–	CR	`c.ebreak (rs1=0,rs2=0)`	`ebreak`
10 (2)	1001––	–	CR	`c.jalr rs1 (rs1≠0,rs2=0)`	`jalr ra, rs1, 0`
10 (2)	1001––	–	CR	`c.add rd, rs2 (rs1≠0,rs2≠0)`	`add rd, rd, rs2`
10 (2)	101–––	–	CSS	`c.fsdsp fs2, imm`	`fsd fs2, (ZeroExt(imm)*8)(sp)`
10 (2)	110–––	–	CSS	`c.swsp rs2, imm`	`sw rs2, (ZeroExt(imm)*4)(sp)`
10 (2)	111–––	–	CSS	`c.fswsp fs2, imm`	`fsw fs2, (ZeroExt(imm)*4)(sp)`

rs1', rs2', rd': 3-bit register designator for registers 8–15: 000_2 = `x8` or `f8`, 001_2 = `x9` or `f9`, etc.

Table B.7 **RISC-V pseudoinstructions**

Pseudoinstruction	RISC-V Instructions	Description	Operation
nop	addi x0, x0, 0	no operation	
li rd, imm$_{11:0}$	addi rd, x0, imm$_{11:0}$	load 12-bit immediate	rd = SignExtend(imm$_{11:0}$)
li rd, imm$_{31:0}$	lui rd, imm$_{31:12}$ * addi rd, rd, imm$_{11:0}$	load 32-bit immediate	rd = imm$_{31:0}$
mv rd, rs1	addi rd, rs1, 0	move (also called "register copy")	rd = rs1
not rd, rs1	xori rd, rs1, −1	one's complement	rd = ~rs1
neg rd, rs1	sub rd, x0, rs1	two's complement	rd = −rs1
seqz rd, rs1	sltiu rd, rs1, 1	set if = 0	rd = (rs1 == 0)
snez rd, rs1	sltu rd, x0, rs1	set if ≠ 0	rd = (rs1 ≠ 0)
sltz rd, rs1	slt rd, rs1, x0	set if < 0	rd = (rs1 < 0)
sgtz rd, rs1	slt rd, x0, rs1	set if > 0	rd = (rs1 > 0)
beqz rs1, label	beq rs1, x0, label	branch if = 0	if (rs1 == 0) PC = label
bnez rs1, label	bne rs1, x0, label	branch if ≠ 0	if (rs1 ≠ 0) PC = label
blez rs1, label	bge x0, rs1, label	branch if ≤ 0	if (rs1 ≤ 0) PC = label
bgez rs1, label	bge rs1, x0, label	branch if ≥ 0	if (rs1 ≥ 0) PC = label
bltz rs1, label	blt rs1, x0, label	branch if < 0	if (rs1 < 0) PC = label
bgtz rs1, label	blt x0, rs1, label	branch if > 0	if (rs1 > 0) PC = label
ble rs1, rs2, label	bge rs2, rs1, label	branch if ≤	if (rs1 ≤ rs2) PC = label
bgt rs1, rs2, label	blt rs2, rs1, label	branch if >	if (rs1 > rs2) PC = label
bleu rs1, rs2, label	bgeu rs2, rs1, label	branch if ≤ (unsigned)	if (rs1 ≤ rs2) PC = label
bgtu rs1, rs2, label	bltu rs2, rs1, offset	branch if > (unsigned)	if (rs1 > rs2) PC = label
j label	jal x0, label	jump	PC = label
jal label	jal ra, label	jump and link	PC = label, ra = PC + 4
jr rs1	jalr x0, rs1, 0	jump register	PC = rs1
jalr rs1	jalr ra, rs1, 0	jump and link register	PC = rs1, ra = PC + 4
ret	jalr x0, ra, 0	return from function	PC = ra
call label	jal ra, label	call nearby function	PC = label, ra = PC + 4
call label	auipc ra, offset$_{31:12}$ * jalr ra, ra, offset$_{11:0}$	call far away function	PC = PC + offset, ra = PC + 4
la rd, symbol	auipc rd, symbol$_{31:12}$ * addi rd, rd, symbol$_{11:0}$	load address of global variable	rd = PC + symbol
l{b|h|w} rd, symbol	auipc rd, symbol$_{31:12}$ * l{b|h|w} rd, symbol$_{11:0}$(rd)	load global variable	rd = [PC + symbol]
s{b|h|w} rs2, symbol, rs1	auipc rs1, symbol$_{31:12}$ * s{b|h|w} rs2, symbol$_{11:0}$(rs1)	store global variable	[PC + symbol] = rs2
csrr rd, csr	csrrs rd, csr, x0	read CSR	rd = csr
csrw csr, rs1	csrrw x0, csr, rs1	write CSR	csr = rs1

* If bit 11 of the immediate / offset / symbol is 1, the upper immediate is incremented by 1. symbol and offset are the 32-bit PC-relative addresses of a label and a global variable, respectively.

Table B.8 **Privileged / CSR instructions**

op	funct3	Type	Instruction	Description	Operation
1110011 (115)	000	I	ecall	transfer control to OS (imm=0)	
1110011 (115)	000	I	ebreak	transfer control to debugger (imm=1)	
1110011 (115)	000	I	uret	return from user exception (rs1=0,rd=0,imm=2)	PC = uepc
1110011 (115)	000	I	sret	return from supervisor exception (rs1=0,rd=0,imm=258)	PC = sepc
1110011 (115)	000	I	mret	return from machine exception (rs1=0,rd=0,imm=770)	PC = mepc
1110011 (115)	001	I	csrrw rd,csr,rs1	CSR read/write (imm=CSR number)	rd = csr,csr = rs1
1110011 (115)	010	I	csrrs rd,csr,rs1	CSR read/set (imm=CSR number)	rd = csr,csr = csr | rs1
1110011 (115)	011	I	csrrc rd,csr,rs1	CSR read/clear (imm=CSR number)	rd = csr,csr = csr & ~rs1
1110011 (115)	101	I	csrrwi rd,csr,uimm	CSR read/write immediate (imm=CSR number)	rd = csr,csr = ZeroExt(uimm)
1110011 (115)	110	I	csrrsi rd,csr,uimm	CSR read/set immediate (imm=CSR number)	rd = csr, csr = csr | ZeroExt(uimm)
1110011 (115)	111	I	csrrci rd,csr,uimm	CSR read/clear immediate (imm=CSR number)	rd = csr, csr = csr & ~ZeroExt(uimm)

For privileged / CSR instructions, the 5-bit unsigned immediate, uimm, is encoded in the **rs1** field.

In Praise of *Digital Design and Computer Architecture*

RISC-V Edition

Harris and Harris have shown how to design a RISC-V processor from the gates all the way up through the microarchitecture. Their clear explanations combined with their comprehensive approach give a full picture of both digital design and the RISC-V architecture. With the exciting opportunity that students have to run large digital designs on modern FPGAs, this approach is both informative and enlightening.

David A. Patterson University of California, Berkeley

What broad fields Harris and Harris have distilled into one book! As semiconductor manufacturing matures, the importance of digital design and computer architecture will only increase. Readers will find an approachable and comprehensive treatment of both topics and will walk away with a clear understanding of the RISC-V instruction set architecture.

Andrew Waterman SiFive

There are excellent texts for teaching digital design and there are excellent texts for teaching computer hardware organization - and this textbook does both! It is also unique in its ability to connect the dots. The writing makes the RISC-V architecture understandable by building on the basics, and I have found the exercises to be a great resource across multiple courses.

Roy Kravitz Portland State University

When I first read Harris and Harris's MIPS textbook back in 2008, I thought that it was one of the best books I had ever read for teaching computer architecture. I started using it in my courses immediately. Thirteen years later, I have had the honor of reviewing this new RISC-V edition, and my opinion of their books has not changed: *Digital Design and Computer Architecture: RISC-V Edition* is an excellent book, very clear, thorough, with a high educational value, and in line with the courses that we teach in the areas of digital design and computer architecture. I look forward to using this RISC-V textbook in my courses.

Daniel Chaver Martinez University Complutense of Madrid

Digital Design and Computer Architecture

RISC-V Edition

Digital Design and Computer Architecture

RISC-V Edition

Sarah L. Harris
David Harris

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK
MORGAN KAUFMANN

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress.

ISBN: 978-0-12-820064-3

Publisher: Katey Birtcher
Senior Acquisitions Editor: Steve Merken
Editorial Project Manager: Ruby Gammell/Andrae Akeh
Senior Project Manager: Manikandan Chandrasekaran
Cover Designer: Victoria Pearson Esser

Typeset by MPS Limited, Chennai, India

Printed in the United States of America

Last digit is the print number: 9 8 7 6 5 4 3 2 1

Contents

Preface

This book is unique in its treatment in that it presents digital logic design from the perspective of computer architecture, starting at the beginning with 1's and 0's and leading through to the design of a microprocessor.

We believe that building a microprocessor is a special rite of passage for engineering and computer science students. The inner workings of a processor seem almost magical to the uninitiated yet prove to be straightforward when carefully explained. Digital design in and of itself is a powerful and exciting subject. Assembly language programming unveils the inner language spoken by the processor. Microarchitecture is the link that brings it all together.

The first two versions of this increasingly popular text cover the MIPS and ARM architectures. As one of the original Reduced Instruction Set Computing architectures, MIPS is clean and exceptionally easy to understand and build. MIPS remains an important architecture, as it has inspired many of the subsequent architectures, including RISC-V. The ARM architecture has exploded in popularity over the past several decades because of its efficiency and rich ecosystem. More than 50 billion ARM processors have been shipped, and more than 75% of humans on the planet use products with ARM processors.

Over the past decade, RISC-V has emerged as an increasingly important architecture, both pedagogically and commercially. As the first widely used open-source computer architecture, RISC-V offers the simplicity of MIPS with the flexibility and features of modern processors.

Pedagogically, the learning objectives of the MIPS, ARM, and RISC-V editions are identical. The RISC-V architecture has a number of features, including extendibility and compressed instructions, that contribute to its efficiency but add a small amount of complexity. The three microarchitectures are also similar, with MIPS and RISC-V architectures sharing many similarities. We expect to offer MIPS, ARM, and RISC-V editions as long as the market demands.

FEATURES

Side-by-Side Coverage of SystemVerilog and VHDL

Hardware description languages (HDLs) are at the center of modern digital design practices. Unfortunately, designers are evenly split between the two dominant languages, SystemVerilog and VHDL. This book introduces HDLs in Chapter 4 as soon as combinational and sequential logic design has been covered. HDLs are then used in Chapters 5 and 7 to design larger building blocks and entire processors. Nevertheless, Chapter 4 can be skipped and the later chapters are still accessible for courses that choose not to cover HDLs.

This book is unique in its side-by-side presentation of SystemVerilog and VHDL, enabling the reader to learn the two languages. Chapter 4 describes principles that apply to both HDLs, and then provides language-specific syntax and examples in adjacent columns. This side-by-side treatment makes it easy for an instructor to choose either HDL and for the reader to transition from one to the other, either in a class or in professional practice.

RISC-V Architecture and Microarchitecture

Chapters 6 and 7 offer in-depth coverage of the RISC-V architecture and microarchitecture. RISC-V is an ideal architecture because it is a real architecture shipped in an increasing number of commercial products, yet it is streamlined and easy to learn. Moreover, because of its popularity in the commercial and hobbyist worlds, simulation and development tools exist for the RISC-V architecture.

Real-World Perspectives

In addition to the real-world perspective in discussing the RISC-V architecture, Chapter 6 illustrates the architecture of Intel x86 processors to offer another perspective. Chapter 9 (available as an online supplement) also describes peripherals in the context of SparkFun's RED-V RedBoard, a popular development board that centers on SiFive's Freedom E310 RISC-V processor. These real-world perspective chapters show how the concepts in the chapters relate to the chips found in many PCs and consumer electronics.

Accessible Overview of Advanced Microarchitecture

Chapter 7 includes an overview of modern high-performance microarchitectural features, including branch prediction, superscalar, and out-of-order operation, multithreading, and multicore processors. The treatment is accessible to a student in a first course and shows how the microarchitectures in the book can be extended to modern processors.

End-of-Chapter Exercises and Interview Questions

The best way to learn digital design is to do it. Each chapter ends with numerous exercises to practice the material. The exercises are followed by a set of interview questions that our industrial colleagues have asked students who are applying for work in the field. These questions provide a helpful glimpse into the types of problems that job applicants will typically encounter during the interview process. Exercise solutions are available via the book's companion and instructor websites.

ONLINE SUPPLEMENTS

Supplementary materials are available online at ddcabook.com or the publisher's website: https://www.elsevier.com/books-and-journals/book-companion/9780128200643. These companion sites (accessible to all readers) include the following:

- Links to video lectures
- Solutions to odd-numbered exercises
- Figures from the text in PDF and PPTX formats
- Links to professional-strength computer-aided design (CAD) tools from Intel®
- Instructions on how to use PlatformIO (an extension of Visual Studio Code) to compile, assemble, and simulate C and assembly code for RISC-V processors
- Hardware description language (HDL) code for the RISC-V processor
- Intel's Quartus helpful hints
- Lecture slides in PowerPoint (PPTX) format
- Sample course and laboratory materials
- List of errata

The instructor site (accessible to instructors who register at https://inspectioncopy.elsevier.com) includes the following:

- Solutions to all exercises
- Laboratory solutions

EdX MOOC

This book also has a companion Massive Open Online Course (MOOC) through EdX. The course includes video lectures, interactive practice problems, and interactive problem sets and labs. The MOOC is divided into two parts: Digital Design (ENGR 85A) and Computer Architecture (ENGR85B) offered by HarveyMuddX (on EdX, search for "Digital Design HarveyMuddX" and "Computer Architecture HarveyMuddX"). EdX does not charge for access to the videos but does charge for the interactive exercises and certificate. EdX offers discounts for students with financial need.

HOW TO USE THE SOFTWARE TOOLS IN A COURSE

Intel's Quartus Software

The Quartus software, either Web or Lite Edition, is a free version of Intel's professional-strength Quartus™ FPGA design tools. It allows students to enter their digital designs in schematic or using either the SystemVerilog or the VHDL hardware description language (HDL). After entering the design, students can simulate their circuits using the ModelSim™-Intel FPGA Edition or Starter Edition, which is available with Intel's Quartus software. Quartus also includes a built-in logic synthesis tool that supports both SystemVerilog and VHDL.

The difference between the Web or Lite Edition and the Pro Edition is that the Web or Lite Edition supports a subset of the most common Altera FPGAs. The free versions of ModelSim degrade performance for simulations with more than 10,000 lines of HDL, whereas the professional version of ModelSim does not.

PlatformIO

PlatformIO, which is an extension of Visual Studio Code, serves as a software development kit (SDK) for RISC-V. With the explosion of SDKs for each new platform, PlatformIO has streamlined the process of programming and using various processors by providing a unified interface for a large number of platforms and devices. It is available as a free download and can be used with SparkFun's RED-V RedBoard, as described in the labs provided on the companion website. PlatformIO provides access to a commercial RISC-V compiler and allows students to write both C and assembly programs, compile them, and then run and debug them on SparkFun's RED-V RedBoard (see Chapter 9 and the accompanying labs).

Venus RISC-V Assembly Simulator

The Venus Simulator (available at: https://www.kvakil.me/venus/) is a web-based RISC-V assembly simulator. Programs are written (or copy/pasted) in the Editor tab and then simulated and run in the Simulator tab. Registers and memory contents can be viewed as the program runs.

LABS

The companion site includes links to a series of labs that cover topics from digital design through computer architecture. The labs teach students how to use the Quartus tools to enter, simulate, synthesize, and implement their designs. The labs also include topics on C and assembly language programming using PlatformIO and SparkFun's RED-V RedBoard.

After synthesis, students can implement their designs using the Altera DE2, DE2-115, DE0, or other FPGA board. The labs are written to target the DE2 or DE-115 boards. These powerful and competitively priced boards are available from de2-115.terasic.com. The board contains an FPGA that can be programmed to implement student designs. We provide labs that describe how to implement a selection of designs on the DE2-115 board using the Quartus software.

To run the labs, students will need to download and install Intel's Quartus Web or Lite Edition and Visual Studio Code with the PlatformIO extension. Instructors may also choose to install the tools on lab machines. The labs include instructions on how to implement the projects on the DE2/DE2-115 board. The implementation step may be skipped, but we have found it of great value. We have tested the labs on Windows, but the tools are also available for Linux.

RVfpga

RISC-V FPGA, also referred to as RVfpga, is a free two-course sequence that can be completed after learning the material in this book. The first course shows how to target a commercial RISC-V core to an FPGA, program it using RISC-V assembly or C, add peripherals to it, and analyze and modify the core and memory system, including adding instructions to the core. This course uses the open-source SweRVolf system-on-chip (SoC) (https://github.com/chipsalliance/Cores-SweRVolf), which is based on Western Digital's open-source commercial SweRV EH1 core (https://www.westerndigital.com/company/innovations/risc-v). The course also shows how to use Verilator, an open-source HDL simulator, and Western Digital's Whisper, an open-source RISC-V instruction set simulator (ISS). RVfpga-SoC, the second course, shows how to build an SoC based on SweRVolf using building blocks such as the SweRV EH1 core, interconnect, and memories. The course then guides the user in loading and running the Zephyr operating system on the RISC-V SoC. All necessary software and system source code (Verilog/SystemVerilog files) are free, and the courses may be completed in simulation, so no hardware is required. RVfpga materials are freely available with registration from the Imagination Technologies University Programme: https://university.imgtec.com/rvfpga/.

BUGS

As all experienced programmers know, any program of significant complexity undoubtedly contains bugs. So, too, do books. We have taken great care to find and squash the bugs in this book. However, some errors undoubtedly do remain. We will maintain a list of errata on the book's webpage.

Please send your bug reports to ddcabugs@gmail.com. The first person to report a substantive bug with a fix that we use in a future printing will be rewarded with a $1 bounty!

ACKNOWLEDGMENTS

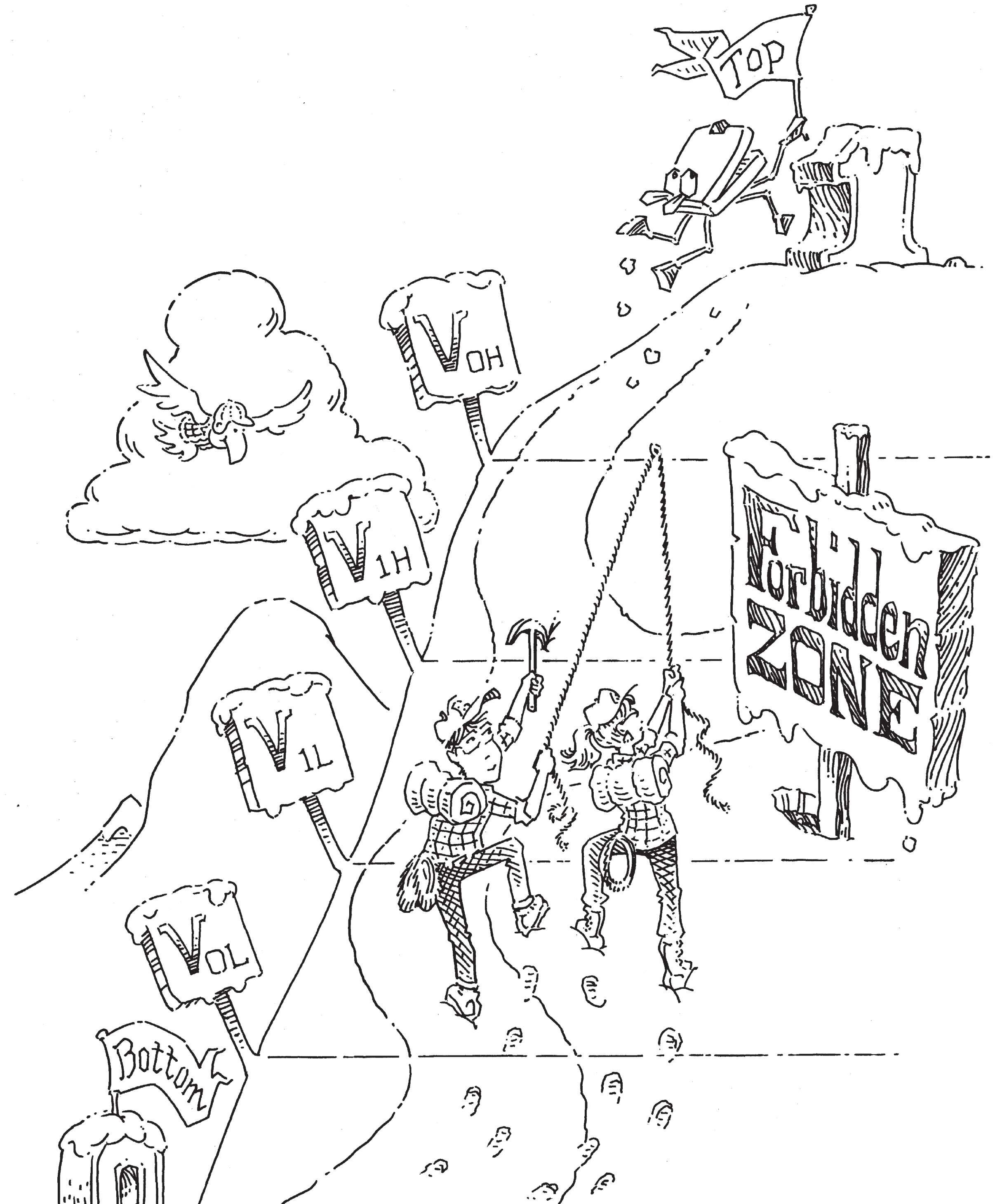
We appreciate the hard work of Steve Merken, Nate McFadden, Ruby Gammell, Andrae Akeh, Manikandan Chandrasekaran, and the rest of the team at Morgan Kaufmann who made this book happen. We love the art of Duane Bibby, whose cartoons enliven the chapters.

We thank Matthew Watkins, who contributed the section on Heterogeneous Multiprocessors in Chapter 7, and Josh Brake, who contributed to Chapter 9 on Embedded I/O Systems. We greatly appreciate the work of Mateo Markovic and Geordie Ryder, who reviewed the book and contributed to the exercise solutions. Numerous other reviewers also substantially improved the book. They include Daniel Chaver Martinez, Roy Kravitz, Zvonimir Bandic, Giuseppe Di Luna, Steffen Paul, Ravi Mittal, Jennifer Winikus, Hesham Omran, Angel Solis, Reiner Dizon, and Olof Kindgren. We also appreciate the students in our courses at Harvey Mudd College and UNLV who have given us helpful feedback on drafts of this textbook. And last, but not least, we both thank our families for their love and support.

About the Authors

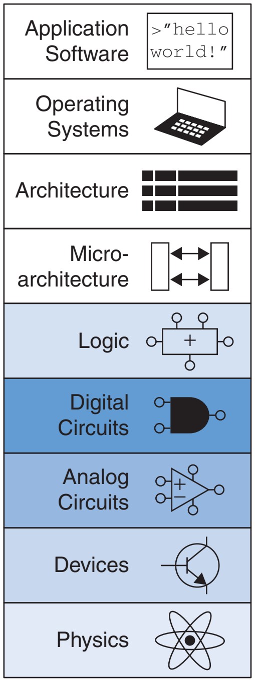
Sarah L. Harris is an Associate Professor of Electrical and Computer Engineering at the University of Nevada, Las Vegas. She received her Ph.D. and M.S. in Electrical Engineering from Stanford University. Sarah has also worked with Hewlett-Packard, the San Diego Supercomputer Center, and NVIDIA. Sarah loves teaching, exploring and developing new technologies, traveling, playing the guitar and various other instruments, and spending time with her family. Her recent research includes designing bio-inspired prosthetics and implementing machine-learning algorithms in hardware.

David Harris is the Harvey S. Mudd Professor of Engineering Design and Associate Department Chair at Harvey Mudd College. He received his Ph.D. in electrical engineering from Stanford University and his M.Eng. in electrical engineering and computer science from MIT. Before attending Stanford, he worked at Intel as a logic and circuit designer on the Itanium and Pentium II processors. Since then, he has consulted at Broadcom, Sun Microsystems, Hewlett-Packard, Evans & Sutherland, and other design companies. David's passions include teaching, building chips and airplanes, and exploring the outdoors. When he is not at work, he can usually be found hiking, flying, and spending time with his family. David holds about a dozen patents and is the author of three other textbooks on chip design, as well as seven guidebooks to the Southern California mountains.


Top
V OH
V 1H
V 1L
V OL
Forbidden ZONE
Bottom
0

From Zero to One

1



1.1 THE GAME PLAN

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 100 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US $21 billion in 1985 to $400 billion in 2020, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and 0's as outputs. We then explore how to combine logic gates into more complicated modules, such as adders and memories. Then, we shift gears to programming in assembly language, the native tongue of the microprocessor. Finally, we put gates together to build a microprocessor that runs these assembly language programs.

A great advantage of digital systems is that the building blocks are quite simple: just 1's and 0's. They do not require grungy mathematics or a profound knowledge of physics. Instead, the designer's challenge is to combine these simple blocks into complicated systems. A microprocessor may be the first system that you build that is too complex to fit in your

Digital Design and Computer Architecture, RISC-V Edition. **DOI:** 10.1016/B978-0-12-820064-3.00001-5

head all at once. One of the major themes woven through this book is how to manage complexity.

1.2 THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to managing complexity. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

1.2.1 Abstraction

The critical technique for managing complexity is *abstraction*: hiding details when they are not important. A system can be viewed from many different levels of abstraction. For example, American politicians abstract the world into cities, counties, states, and countries. A county contains multiple cities and a state contains many counties. When a politician is running for president, the politician is mostly interested in how the state as a whole will vote rather than how each county votes, so the state is the most useful level of abstraction. On the other hand, the Census Bureau measures the population of every city, so the agency must consider the details of a lower level of abstraction.

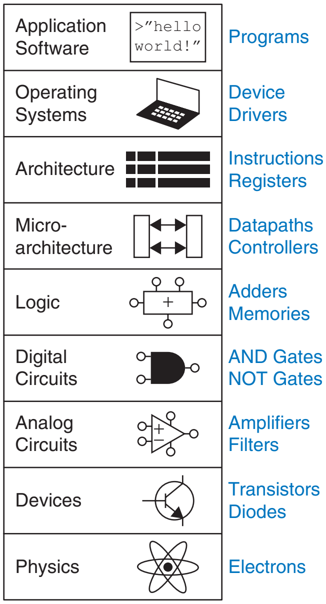


Figure 1.1 Levels of abstraction for an electronic computing system

Figure 1.1 illustrates levels of abstraction for an electronic computer system, along with typical building blocks at each level. At the lowest level of abstraction is the physics, the motion of electrons. The behavior of electrons is described by quantum mechanics and Maxwell's equations. Our system is constructed from electronic *devices* such as transistors (or vacuum tubes, once upon a time). These devices have well-defined connection points called *terminals* and can be modeled by the relationship between voltage and current as measured at each terminal. By abstracting to this device level, we can ignore the individual electrons. The next level of abstraction is *analog circuits*, in which devices are assembled to create components such as amplifiers. Analog circuits input and output a continuous range of voltages. *Digital circuits*, such as logic gates, restrict the voltages to discrete ranges, which we will use to indicate 0 and 1. In logic design, we build more complex structures, such as adders or memories, from digital circuits.

Microarchitecture links the logic and architecture levels of abstraction. The *architecture* level of abstraction describes a computer from the programmer's perspective. For example, the Intel x86 architecture used by microprocessors in most *personal computers* (*PCs*) is defined by a set of instructions and registers (memory for temporarily storing variables) that the programmer is allowed to use. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture. A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core i7, the Intel 80486, and the AMD Athlon all implement the x86 architecture with different microarchitectures.

Moving into the software realm, the operating system handles low-level details, such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user. Thanks to the power of abstraction, your grandmother can surf the Web without any regard for the quantum vibrations of electrons or the organization of the memory in her computer.

This book focuses on the levels of abstraction from digital circuits through computer architecture. When you are working at one level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working. For example, a computer scientist cannot fully optimize code without understanding the architecture for which the program is being written. A device engineer cannot make wise trade-offs in transistor design without understanding the circuits in which the transistors will be used. We hope that by the time you finish reading this book, you can pick the level of abstraction appropriate to solving your problem and evaluate the impact of your design choices on other levels of abstraction.

Each chapter in this book begins with an abstraction icon indicating the focus of the chapter in deep blue, with secondary topics shown in lighter shades of blue.

1.2.2 Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline. One of the famous early examples of interchangeable parts was in automobile manufacturing. Although the modern gas-powered car dates back to the German Benz Patent-Motorwagen of 1886, early cars were handcrafted by skilled tradesmen, a time-consuming and expensive process. Henry Ford made a key advance in 1908 by focusing on mass production with interchangeable parts and moving assembly lines.

The discipline of interchangeable parts revolutionized the industry. By limiting the components to a standardized set with well-defined tolerances, cars could be assembled and repaired much faster and with less skill. The car builder no longer concerned himself with lower levels of abstraction, such as fitting a door to a nonstandardized opening.

Ford launched the Model T with a bold manifesto:

I will build a motor car for the great multitude. It will be large enough for the family, but small enough for the individual to run and care for. It will be constructed of the best materials, by the best men to be hired, after the simplest designs that modern engineering can devise. But it will be so low in price that no man making a good salary will be unable to own one—and enjoy with his family the blessing of hours of pleasure in God's great open spaces. [My Life and Work, by Samuel Crowther and Henry Ford, 1922]

Model T Ford Photo from https://commons.wikimedia.org/wiki/File: TModel_launch_Geelong.jpg.

The Model T Ford became the most-produced car of its era, selling over 15 million units. Another example of discipline was Ford's famous saying: "Any customer can have a car painted any color that he wants so long as it is black."

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a subset of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design. By limiting ourselves to digital circuits, we can easily combine components into sophisticated systems that ultimately outperform those built from analog components in many applications. For example, digital televisions, compact disks (CDs), and cell phones are replacing their analog predecessors.

1.2.3 The Three -Y's

In addition to abstraction and discipline, designers use the three "-y's" to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

- *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- *Modularity* states that modules have well-defined functions and interfaces so that they connect easily without unanticipated side effects.
- *Regularity* seeks uniformity among modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

To illustrate these "-y's," we return to the example of automobile manufacturing. A car was one of the most intricate objects in common use in the early 20th century. Using the principle of hierarchy, we can break the Model T Ford into components, such as the chassis, engine, and seats. The engine, in turn, contained four cylinders, a carburetor, a magneto, and a cooling system. The carburetor contained fuel and air intakes, a choke and throttle, a needle valve, and so forth, as shown in Figure 1.2. The fuel intake was made from a threaded elbow and a coupling nut. Thus, the complex system is recursively broken down into simple interchangeable components that can be mass produced.

Modularity teaches that each component should have a well-defined function and interface. For example, the coupling nut has a function of holding the fuel feed line to the intake elbow in a way that does not

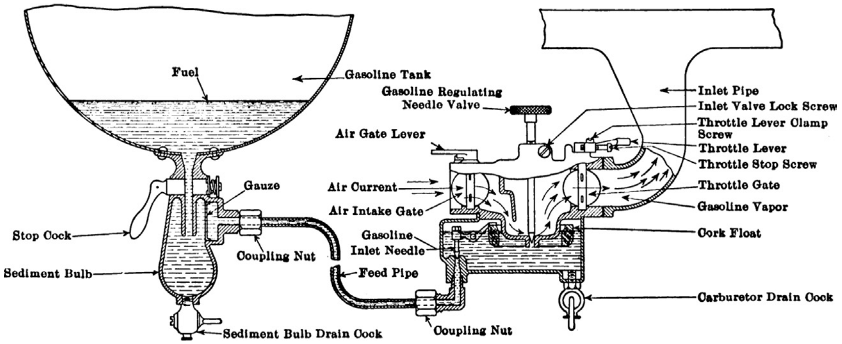


Figure 1.2 Cutaway view of Model T fuel system, showing fuel supply on left and carburetor on right. (https://en.wikipedia.org/wiki/Ford_Model_T_engine#/media/File:Pagé_1917_Model_T_Ford_Car_Figure_14.png)

leak yet can be easily removed when the feed line needs replacement. It is of a standardized diameter and thread pitch, tightened to a standardized torque by a standardized wrench. A car maker can buy the nut from many different suppliers, as long as the correct size is specified. Modularity dictates that there should be no side effects: the coupling nut should not deform the elbow and should preferably be located where it can be tightened or removed without taking off other equipment in the engine.

Regularity teaches that interchangeable parts are a good idea. With regularity, a leaking carburetor can be replaced by an identical part. The carburetors can be efficiently built on an assembly line instead of being painstakingly handcrafted.

We will return to these principles of hierarchy, modularity, and regularity throughout the book.

1.3 THE DIGITAL ABSTRACTION

Most physical variables are continuous. For example, the voltage on a wire, the frequency of an oscillation, or the position of a mass are all continuous quantities. Digital systems, on the other hand, represent information with *discrete-valued variables*—that is, variables with a finite number of distinct values.

An early digital system using variables with ten discrete values was Charles Babbage's Analytical Engine. Babbage labored from 1834 to 1871, designing and attempting to build this mechanical computer. The Analytical Engine used gears with ten positions labeled 0 through 9, much like a mechanical odometer in a car. Figure 1.3 shows a prototype of the Analytical Engine, in which each row processes one digit. Babbage chose 25 rows of gears, so the machine has 25-digit precision.

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a "1" and

Charles Babbage, 1791–1871
Attended Cambridge University and married Georgiana Whitmore in 1814. Invented the Analytical Engine, the world's first mechanical computer. Also invented the cowcatcher and the universal postage rate. Interested in lock picking, but abhorred street musicians (image courtesy of Fourmilab Switzerland, www.fourmilab.ch).

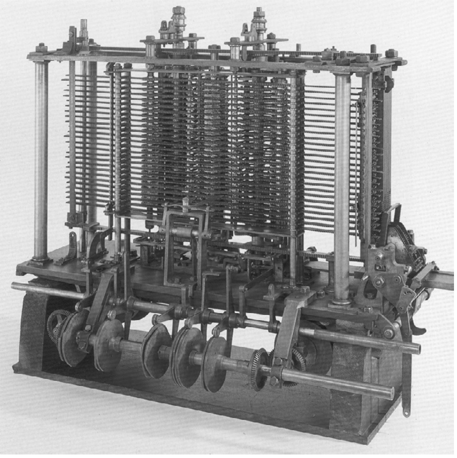


Figure 1.3 Babbage's Analytical Engine, under construction at the time of his death in 1871 (image courtesy of Science Museum/ Science and Society Picture Library)

a low voltage indicates a "0," because it is easier to distinguish between two voltages than ten.

The *amount of information D* in a discrete valued variable with *N* distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits} \tag{1.1}$$

A binary variable conveys $\log_2 2 = 1$ bit of information. Indeed, the word *bit* is short for *b*inary dig*it*. Each of Babbage's gears carried $\log_2 10 = 3.322$ bits of information because it could be in one of $2^{3.322} = 10$ unique positions. A continuous signal theoretically contains an infinite amount of information because it can take on an infinite number of values. In practice, noise and measurement error limit the information to only 10 to 16 bits for most continuous signals. If the measurement must be made rapidly, the information content is lower (e.g., 8 bits).

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent "1" and zero volts to represent "0." In this book, we will use the terms "1," "TRUE," and "HIGH" synonymously. Similarly, we will use "0", "FALSE," and "LOW" interchangeably.

The beauty of the *digital abstraction* is that digital designers can focus on 1's and 0's, ignoring whether the Boolean variables are physically represented with specific voltages, rotating gears, or even hydraulic fluid levels. A computer programmer can work without needing to know the intimate details of the computer hardware. On the other hand, understanding the details of the hardware allows the programmer to optimize the software better for that specific computer.

George Boole, 1815–1864
Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote *An Investigation of the Laws of Thought* (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT (image courtesy of the American Institute of Physics).

An individual bit doesn't carry much information. In the next section, we examine how groups of bits can be used to represent numbers. In later chapters, we will also use groups of bits to represent letters and programs.

1.4 NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient. This section introduces the various number systems that will be used throughout the rest of the book.

1.4.1 Decimal Numbers

In elementary school, you learned to count and do arithmetic in *decimal*. Just as you (probably) have ten fingers, there are ten decimal digits: 0, 1, 2, …, 9. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are 1, 10, 100, 1000, and so on. Decimal numbers are referred to as *base 10*. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.4 shows how the decimal number 9742_{10} is written as the sum of each of its digits multiplied by the weight of the corresponding column.

An N-digit decimal number represents one of 10^N possibilities: 0, 1, 2, 3, …, $10^N - 1$. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

1.4.2 Binary Numbers

Bits represent one of two values, 0 or 1, and are joined together to form *binary numbers*. Each column of a binary number has twice the weight of the previous column, so binary numbers are *base 2*. In binary, the column weights (again, from right to left) are 1, 2, 4, 8, 16, 32, 64, 128,

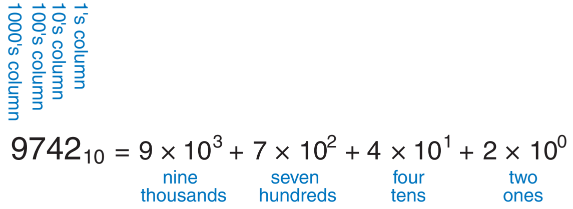


Figure 1.4 **Representation of a decimal number**

256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on. If you work with binary numbers often, you'll save time if you remember these powers of two up to 2^{16}.

An N-bit binary number represents one of 2^N possibilities: 0, 1, 2, 3, …, $2^N - 1$. Table 1.1 shows 1-, 2-, 3-, and 4-bit binary numbers and their decimal equivalents.

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution Figure 1.5 shows the conversion.

Table 1.1 **Binary numbers and their decimal equivalent**

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

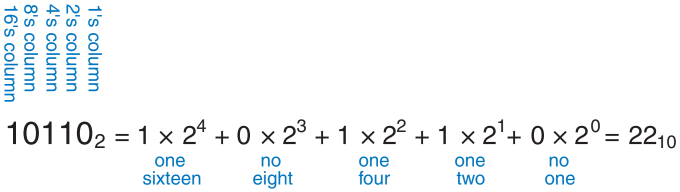

1's column
2's column
4's column
8's column
16's column
$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$
one sixteen
no eight
one four
one two
no one

Figure 1.5 **Conversion of a binary number to decimal**

Example 1.2 DECIMAL TO BINARY CONVERSION

Convert the decimal number 84_{10} to binary.

Solution Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than or equal to the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving $84 - 64 = 20$. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving $20 - 16 = 4$. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving $4 - 4 = 0$. Thus, there must be 0s in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. $84/2 = 42$, so 0 goes in the 1's column. $42/2 = 21$, so 0 goes in the 2's column. $21/2 = 10$ with the remainder of 1 going in the 4's column. $10/2 = 5$, so 0 goes in the 8's column. $5/2 = 2$ with the remainder of 1 going in the 16's column. $2/2 = 1$, so 0 goes in the 32's column. Finally, $1/2 = 0$ with the remainder of 1 going in the 64's column. Again, $84_{10} = 1010100_2$.

1.4.3 Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of $2^4 = 16$ possibilities. Hence, it is sometimes more convenient to work in *base 16*, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F, as shown in Table 1.2. Columns in base 16 have weights of 1, 16, 16^2 (or 256), 16^3 (or 4096), and so on.

Hexadecimal, a term coined by IBM in 1963, derives from the Greek *hexi* (six) and Latin *decem* (ten). A more proper term would use the Latin *sexa* (six), but *sexadecimal* sounded too risqué.

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

Convert the hexadecimal number $2ED_{16}$ to binary and to decimal.

Solution Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. $2_{16} = 0010_2$, $E_{16} = 1110_2$ and $D_{16} = 1101_2$, so $2ED_{16} = 001011101101_2$. Conversion to decimal requires the arithmetic shown in Figure 1.6.

Table 1.2 **Hexadecimal number system**

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

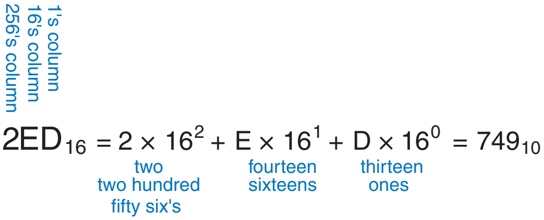

1's column
16's column
256's column
2ED16 = 2 × 16² + E × 16¹ + D × 16⁰ = 74910
two two hundred fifty six's
fourteen sixteens
thirteen ones

Figure 1.6 **Conversion of a hexadecimal number to decimal**

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

Solution Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence, $1111010_2 = 7A_{16}$.

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than or equal to the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving 333 − 256 = 77. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving 77 − 16 × 4 = 13. $13_{10} = D_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14D_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14D_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. 333/16 = 20, with the remainder of $13_{10} = D_{16}$ going in the 1's column. 20/16 = 1, with the remainder of 4 going in the 16's column. 1/16 = 0, with the remainder of 1 going in the 256's column. Again, the result is $14D_{16}$.

1.4.4 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a *byte*. It represents one of $2^8 = 256$ possibilities. The size of objects stored in computer memories is customarily measured in bytes rather than bits.

A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities. One hexadecimal digit stores one nibble and two hexadecimal digits store one full byte. Nibbles are no longer a commonly used unit, but the term is cute.

Microprocessors handle data in chunks called *words*. The size of a word depends on the architecture of the microprocessor. When this chapter was written in 2021, most computers had 64-bit processors, indicating that they operate on 64-bit words. At the time, older computers handling 32-bit words were also widely available. Simpler microprocessors, especially those used in gadgets such as toasters, use 8- or 16-bit words.

Within a group of bits, the bit in the 1's column is called the *least significant bit* (*lsb*), and the bit at the other end is called the *most significant bit* (*msb*), as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a word, the bytes are identified as *least significant byte* (*LSB*) through *most significant byte* (*MSB*), as shown in Figure 1.7(b) for a 4-byte number written with eight hexadecimal digits.

By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 2^{10}. For example, 2^{10} bytes is one

A *microprocessor* is a processor built on a single chip. Until the 1970's, processors were too complicated to fit on one chip, so mainframe processors were built from boards containing many chips. Intel introduced the first 4-bit microprocessor, called the 4004, in 1971. Now, even the most sophisticated supercomputers are built using microprocessors. We will use the terms *microprocessor* and *processor* interchangeably throughout this book.

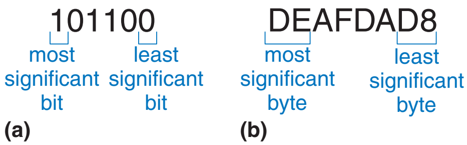


Figure 1.7 **Least and most significant bits and bytes**

To distinguish between powers of ten and powers of two, a naming convention specific to powers of two has emerged: 2^{10} is called a kibi (Ki), 2^{20} mebi (Mi), 2^{30} gibi (Gi), 2^{40} tebi (Ti), 2^{50} pebi (Pi), 2^{60} exbi (Ei), and so on. For example, 2^{30} bytes is a gibibyte (GiB).

kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$. If you know $2^{10} \approx 1$ thousand, $2^{20} \approx 1$ million, $2^{30} \approx 1$ billion, and remember the powers of two up to 2^9, it is easy to estimate any power of two in your head.

Example 1.6 ESTIMATING POWERS OF TWO

Find the approximate value of 2^{24} without using a calculator.

Solution Split the exponent into a multiple of ten and the remainder.

$2^{24} = 2^{20} \times 2^4$. $2^{20} \approx 1$ million. $2^4 = 16$. So, $2^{24} \approx 16$ million. Technically, $2^{24} = 16{,}777{,}216$, but 16 million is close enough for marketing purposes.

On-chip memory and RAM is measured in powers of two, but hard drives are measured in powers of ten. For example, 32 GB (GiB) of RAM is actually 34,359,738,368 bytes of memory, whereas 32 GB on a hard drive is 32,000,000 bytes of memory.

1024 bytes is called a *kilobyte* (KB) or *kibibyte* (KiB). 1024 bits is called a *kilobit* (Kb or Kbit) or *kibibit* (Kib or Kibit). Similarly, MB/MiB, Mb/Mib, GB/GiB, and Gb/Gib are used for millions and billions of bytes and bits. Memory capacity is usually measured in bytes. Communication speed is usually measured in powers of ten bits/second. For example, the maximum speed of a dial-up modem is usually 56 kbits/sec, which is 56,000 bits/sec.

1.4.5 Binary Addition

Binary addition is much like decimal addition but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column. Figure 1.8 compares addition of decimal and binary numbers. In the rightmost column of Figure 1.8(a), 7 + 9 = 16, which cannot fit in a single digit because it is greater than 9. So, we record the 1's digit, 6, and carry the 10's digit, 1, over to the next column. Likewise, in binary, if the sum of two numbers is greater than 1, we carry the 2's digit over to the next column. For example, in the rightmost column of Figure 1.8(b), the sum

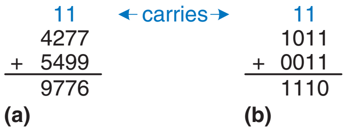


Figure 1.8 **Addition examples showing carries: (a) decimal (b) binary**

$1 + 1 = 2_{10} = 10_2$ cannot fit in a single binary digit. So, we record the 1's digit (0) and carry the 2's digit (1) of the result to the next column. In the second column, the sum is $1 + 1 + 1 = 3_{10} = 11_2$. Again, we record the 1's digit (1) and carry the 2's digit (1) to the next column. For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*.

Example 1.7 BINARY ADDITION

Compute $0111_2 + 0101_2$.

Solution Figure 1.9 shows that the sum is 1100_2. The carries are indicated in blue. We can check our work by repeating the computation in decimal. $0111_2 = 7_{10}$. $0101_2 = 5_{10}$. The sum is $12_{10} = 1100_2$.

$$\begin{array}{r} 111 \\ 0111 \\ +\ 0101 \\ \hline 1100 \end{array}$$

Figure 1.9 **Binary addition example**

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits. A 4-bit number, for example, has the range [0, 15]. 4-bit binary addition overflows if the result exceeds 15. The fifth bit of the sum is discarded, producing an incorrect result in the remaining four bits. Overflow can be detected by checking for a carry out of the most significant column.

Example 1.8 ADDITION WITH OVERFLOW

Compute $1101_2 + 0101_2$. Does overflow occur?

Solution Figure 1.10 shows that the sum is 10010_2. This result overflows the range of a 4-bit binary number. If it must be stored as four bits, the most significant bit is discarded, leaving the incorrect result of 0010_2. If the computation had been done using numbers with five or more bits, the result 10010_2 would have been correct.

$$\begin{array}{r} 11\ 1 \\ 1101 \\ +\ 0101 \\ \hline 10010 \end{array}$$

Figure 1.10 **Binary addition example with overflow**

1.4.6 Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent *signed* binary numbers. The two most widely employed are called *sign/magnitude* and *two's complement*.

Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a minus sign followed by the magnitude. An N-bit sign/magnitude number uses the most

The \$7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

Photograph courtesy of ESA/CNES/ARIANESPACE-Service Optique CS6.

The result of inverting the bits in a number is called the *one's complement* of that number. For example, the one's complement of 0110111 is 1001000. It is called the one's complement because when you add the numbers together, you get all 1's. For example, 0110111 + 1001000 = 1111111.

significant bit as the sign and the remaining $N-1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

Example 1.9 SIGN/MAGNITUDE NUMBERS

Write 5 and −5 as 4-bit sign/magnitude numbers.

Solution Both numbers have a magnitude of $5_{10} = 101_2$. Thus, $5_{10} = 0101_2$ and $-5_{10} = 1101_2$.

Unfortunately, ordinary binary addition does not work for sign/magnitude numbers. For example, using ordinary addition on $-5_{10} + 5_{10}$ gives $1101_2 + 0101_2 = 10010_2$, which is nonsense.

An N-bit sign/magnitude number spans the range $[-2^{N-1}+1, 2^{N-1}-1]$. Sign/magnitude numbers are slightly odd in that both +0 and −0 exist. Both indicate zero. As you may expect, it can be troublesome to have two different representations for the same number.

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00\ldots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\ldots111_2 = 2^{N-1}-1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\ldots000_2 = -2^{N-1}$. And −1 is written as all ones: $11\ldots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed (e.g., from +5 to −5 or from −17 to +17) by inverting the bits in the number and then adding 1 to the least significant bit position. This process is called the *reversing the sign* method. It is useful in finding the representation of a negative number or determining the magnitude of a negative number.

Example 1.10 TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution Start by representing the magnitude: $+2_{10} = 0010_2$. To get -2_{10}, reverse the sign by inverting the bits and adding 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So, -2_{10} is 1110_2.

The process of *reversing the sign* has historically been called "taking the two's complement." However, we have found that this terminology sometimes makes people think that they *always* have to "take the two's complement" when working with two's complement numbers, but that is not true! This process of reversing the sign is only sometimes used when working with negative two's complement numbers.

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

Find the decimal value of the 4-bit two's complement number 0x9 (i.e., 1001_2).

Solution 1001_2 has a leading 1, so it must be negative. To find its magnitude, reverse the sign by inverting the bits and adding 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N + 1^{\text{th}}$ result bit) is discarded.

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution (a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now, add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now, add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

Two's complement numbers earned their name from the fact that creating the *complement* (negative) of a number is found by subtracting it from 2^N (referred to by its base: *two's*)—thus, the two's complement. For example, to create the 4-bit complement of 7 (i.e., −7), we perform: $2^4 - 7 = 10000 - 0111 = (1111 + 1) - 0111 = (1111 - 0111) + 1 = 1000 + 1 = 1001$. This is the 4-bit two's complement representation of −7. Notice that the final step was the same as the reversing the sign method, that is, invert the bits of 0111 (to 1000) and add 1.

The two's complement of 0 is found by inverting all the bits (producing $11\ldots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate −0. Zero is considered positive because its sign bit is 0.

Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However, the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: −8 to 7. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no −0. The most negative number $10\ldots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\ldots111_2$) and adding 1, which produces $10\ldots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.

Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

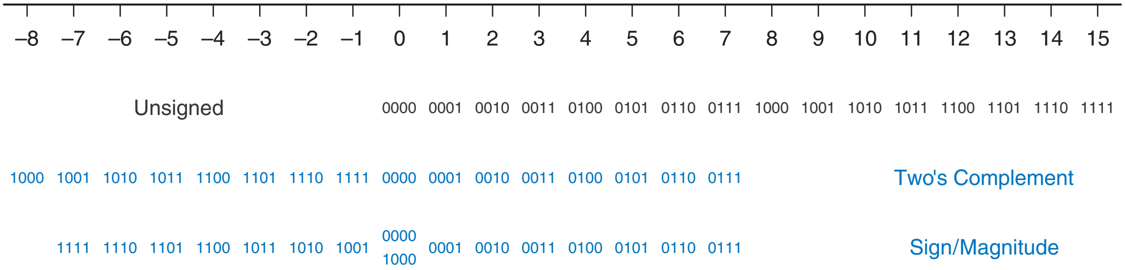
When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and −3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively, which are 7-bit representations of 3 and –3.

Comparison of Number Systems

The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude. Table 1.3 compares the range of N-bit numbers in each of these three systems. Two's complement numbers are convenient because they represent both positive and negative integers and because ordinary addition works for all numbers. Subtraction is performed by negating the second number (i.e., reversing

Table 1.3 Range of *N*-bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$



	Unsigned	Two's Complement	Sign/Magnitude
–8		1000	
–7		1001	1111
–6		1010	1110
–5		1011	1101
–4		1100	1100
–3		1101	1011
–2		1110	1010
–1		1111	1001
0	0000	0000	0000 1000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111
8	1000		
9	1001		
10	1010		
11	1011		
12	1100		
13	1101		
14	1110		
15	1111		

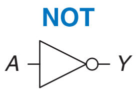
Figure 1.11 Number line and 4-bit binary encodings

the sign), and then adding. Unless stated otherwise, assume that all signed binary numbers use two's complement representation.

Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range [0, 15] in regular binary order. Two's complement numbers span the range [−8, 7]. The nonnegative numbers [0, 7] share the same encodings as unsigned numbers. The negative numbers [−8, −1] are encoded such that a larger unsigned binary value represents a number closer to 0. Notice that the weird number, 1000, represents −8 and has no positive counterpart. Sign/magnitude numbers span the range [−7, 7]. The most significant bit is the sign bit. The positive numbers [1, 7] share the same encodings as unsigned numbers. The negative numbers are symmetric but have the sign bit set. 0 is represented by both 0000 and 1000. Thus, *N*-bit sign/magnitude numbers represent only $2^N - 1$ integers because of the two representations for 0. *N*-bit unsigned binary or two's complement numbers represent 2^N integers.

1.5 LOGIC GATES

Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input (or inputs) and the output. Inputs are usually drawn on the left (or top) and outputs on the right (or bottom). Digital designers typically use letters near the beginning of the alphabet for gate inputs and the letter *Y* for the gate output. The relationship between the inputs and the output can be described with a truth table or a Boolean equation. A *truth table* lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A *Boolean equation* is a mathematical expression using binary variables.



$Y = \overline{A}$

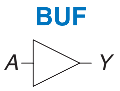
A	Y
0	1
1	0

Figure 1.12 **NOT gate**

1.5.1 NOT Gate

A *NOT gate* has one input, *A*, and one output, *Y*, as shown in Figure 1.12. The NOT gate's output is the inverse of its input. If *A* is FALSE, then *Y* is TRUE. If *A* is TRUE, then *Y* is FALSE. This relationship is summarized by the truth table and Boolean equation in the figure. The line over *A* in the Boolean equation is pronounced *NOT*, so $Y = \overline{A}$ is read "*Y* equals NOT A." The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$, or $Y = {\sim}A$. We will use $Y = \overline{A}$ exclusively, but don't be puzzled if you encounter another notation elsewhere.



$Y = A$

A	Y
0	0
1	1

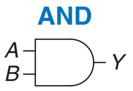
Figure 1.13 **Buffer**

1.5.2 Buffer

The other one-input logic gate is called a *buffer* and is shown in Figure 1.13. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. However, from the analog point of view, the buffer might have desirable characteristics, such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates. This is an example of why we need to consider multiple levels of abstraction to fully understand a system; the digital abstraction hides the real purpose of a buffer.

The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1.14 **AND gate**

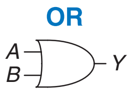
1.5.3 AND Gate

Two-input logic gates are more interesting. The *AND gate* shown in Figure 1.14 produces a TRUE output, *Y*, if and only if both *A* and *B* are TRUE. Otherwise, the output is FALSE. By convention, the inputs are listed in the order 00, 01, 10, 11, as if you were counting in binary. The Boolean equation for an AND gate can be written in several ways: $Y = A \bullet B$, $Y = AB$, or $Y = A \cap B$. The $\cap$ symbol is pronounced "intersection" and is preferred by logicians. We prefer $Y = AB$, read "*Y* equals *A* and *B*," because we are lazy.

According to Larry Wall, inventor of the Perl programming language, "the three principal virtues of a programmer are Laziness, Impatience, and Hubris."

1.5.4 OR Gate

The *OR gate* shown in Figure 1.15 produces a TRUE output, Y, if either A or B (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$. The $\cup$ symbol is pronounced union and is preferred by logicians. Digital designers normally use the + notation, $Y = A + B$ is pronounced "Y equals A or B."



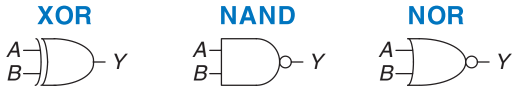
$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 1.15 **OR gate**

1.5.5 Other Two-Input Gates

Figure 1.16 shows other common two-input logic gates. *XOR* (exclusive OR, pronounced "ex-OR") is TRUE if A or B, but not both, are TRUE. The XOR operation is indicated by $\oplus$, a plus sign with a circle around it. Any gate can be followed by a bubble to invert its operation. The *NAND gate* performs NOT AND. Its output is TRUE unless both inputs are TRUE. The *NOR gate* performs NOT OR. Its output is TRUE if neither A nor B is TRUE. An N-input XOR gate is sometimes called a *parity* gate and produces a TRUE output if an odd number of inputs are TRUE. As with two-input gates, the input combinations in the truth table are listed in counting order.



XOR: $Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NAND: $Y = \overline{AB}$

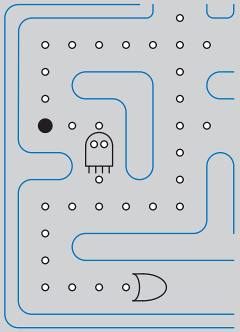
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR: $Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 1.16 **More two-input logic gates**

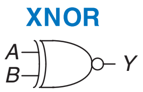
A silly way to remember the OR symbol is that its input side is curved like Pacman's mouth, so the gate is hungry and willing to eat any TRUE inputs it can find!



Example 1.15 XNOR GATE

Figure 1.17 shows the symbol and Boolean equation for a two-input *XNOR* (pronounced ex-NOR) *gate* that performs the inverse of an XOR. Complete the truth table.

Solution Figure 1.18 shows the truth table. The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an *equality* gate because its output is TRUE when the inputs are equal.



$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

Figure 1.17 **XNOR gate**

1.5.6 Multiple-Input Gates

Many Boolean functions of three or more inputs exist. The most common are AND, OR, XOR, NAND, NOR, and XNOR. An N-input AND gate produces a TRUE output when all N inputs are TRUE. An N-input OR gate produces a TRUE output when at least one input is TRUE.

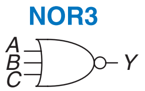
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.18 XNOR truth table

Example 1.16 THREE-INPUT NOR GATE

Figure 1.19 shows the symbol and Boolean equation for a three-input NOR gate. Complete the truth table.

Solution Figure 1.20 shows the truth table. The output is TRUE only if none of the inputs are TRUE.



$Y = \overline{A + B + C}$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Figure 1.19 Three-input NOR gate

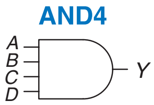
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 1.20 Three-input NOR truth table

Example 1.17 FOUR-INPUT AND GATE

Figure 1.21 shows the symbol and Boolean equation for a four-input AND gate. Create a truth table.

Solution Figure 1.22 shows the truth table. The output is TRUE only if all of the inputs are TRUE.



$Y = ABCD$

Figure 1.21 Four-input AND gate

1.6 BENEATH THE DIGITAL ABSTRACTION

A digital system uses discrete-valued variables. However, the variables are represented by continuous physical quantities, such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder. Hence, the designer must choose a way to relate the continuous value to the discrete value.

For example, consider representing a binary signal A with a voltage on a wire. Let 0 volts (V) indicate $A = 0$ and 5 V indicate $A = 1$. Any real system must tolerate some noise, so 4.97 V probably ought to be interpreted as $A = 1$ as well. But what about 4.3 V? Or 2.8 V? Or 2.500000 V?

1.6.1 Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called $V_{DD.}$ In 1970's and 1980's technology, V_{DD} was generally 5 V. As chips have progressed to smaller transistors, V_{DD} has dropped to 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, or even lower to save power and avoid overloading the transistors.

1.6.2 Logic Levels

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown in Figure 1.23. The first gate is called the *driver* and the second gate is called the *receiver*. The output of the driver is connected to the input of the receiver. The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD}. If the receiver gets an input in the range of 0 to V_{IL}, it will consider the input to be LOW. If the receiver gets an input in the range of V_{IH} to V_{DD}, it will consider the input to be HIGH. If, for some reason such as noise or faulty components, the receiver's input should fall in the *forbidden zone* between V_{IL} and V_{IH}, the behavior of the gate is unpredictable. V_{OH}, V_{OL}, V_{IH}, and V_{IL} are called the output and input high and low logic levels.

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 1.22 Four-input AND truth table

1.6.3 Noise Margins

If the output of the driver is to be correctly interpreted at the input of the receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$. Thus, even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level. The *noise margin* (*NM*) is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input. As can be seen in Figure 1.23, the low and high noise margins are, respectively,

$$NM_L = V_{IL} - V_{OL} \tag{1.2}$$

$$NM_H = V_{OH} - V_{IH} \tag{1.3}$$

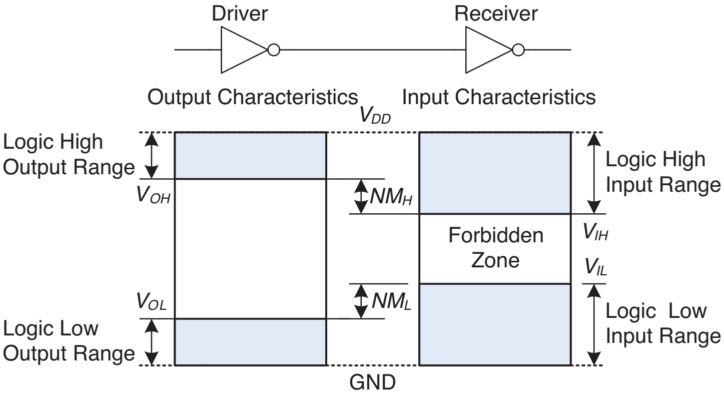


Figure 1.23 Logic levels and noise margins

Example 1.18 CALCULATING NOISE MARGINS

Consider the inverter circuit of Figure 1.24. V_{O1} is the output voltage of inverter I1, and V_{I2} is the input voltage of inverter I2. Both inverters have the following characteristics: $V_{DD} = 5\,\text{V}$, $V_{IL} = 1.35\,\text{V}$, $V_{IH} = 3.15\,\text{V}$, $V_{OL} = 0.33\,\text{V}$, and $V_{OH} = 3.84\,\text{V}$. What are the inverter low and high noise margins? Can the circuit tolerate 1 V of noise between V_{O1} and V_{I2}?

V_{DD} stands for the voltage on the *drain* of a metal-oxide-semiconductor transistor, used to build most modern chips. The power supply voltage is also sometimes called V_{CC}, standing for the voltage on the *collector* of a bipolar junction transistor used to build chips in an older technology. Ground is sometimes called V_{SS} because it is the voltage on the *source* of a metal-oxide-semiconductor transistor. See Section 1.7 for more information on transistors.

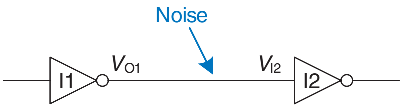


Figure 1.24 Inverter circuit

Solution The inverter noise margins are: $NM_L = V_{IL} - V_{OL} = (1.35\,\text{V} - 0.33\,\text{V}) = 1.02\,\text{V}$, $NM_H = V_{OH} - V_{IH} = (3.84\,\text{V} - 3.15\,\text{V}) = 0.69\,\text{V}$. The circuit can tolerate 1 V of noise when the output is LOW ($NM_L = 1.02\,\text{V}$) but not when the output is HIGH ($NM_H = 0.69\,\text{V}$). For example, suppose the driver, I1, outputs its worst-case HIGH value, $V_{O1} = V_{OH} = 3.84\,\text{V}$. If noise causes the voltage to droop by 1 V before reaching the input of the receiver, $V_{I2} = (3.84\,\text{V} - 1\,\text{V}) = 2.84\,\text{V}$. This is less than the acceptable input HIGH value, $V_{IH} = 3.15\,\text{V}$, so the receiver may not sense a proper HIGH input.

DC indicates behavior when an input voltage is held constant or changes slowly enough for the rest of the system to keep up. The term's historical root comes from *direct current*, a method of transmitting power across a line with a constant voltage. In contrast, the *transient response* of a circuit is the behavior when an input voltage changes rapidly. Section 2.9 explores transient response further.

1.6.4 DC Transfer Characteristics

To understand the limits of the digital abstraction, we must delve into the analog behavior of a gate. The DC *transfer characteristics* of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. They are called transfer characteristics because they describe the relationship between input and output voltages.

An ideal inverter would have an abrupt switching threshold at $V_{DD}/2$, as shown in Figure 1.25(a). For $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. For $V(A) > V_{DD}/2$, $V(Y) = 0$. In such a case, $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ and $V_{OL} = 0$.

A real inverter changes more gradually between the extremes, as shown in Figure 1.25(b). When the input voltage $V(A)$ is 0, the output voltage $V(Y) = V_{DD}$. When $V(A) = V_{DD}$, $V(Y) = 0$. However, the transition between these endpoints is smooth and may not be centered at exactly $V_{DD}/2$. This raises the question of how to define the logic levels.

A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y)/dV(A)$ is −1. These two points are called the *unity gain points*. Choosing logic levels at the unity gain points usually maximizes the noise margins. If V_{IL} were reduced, V_{OH} would only increase by a small amount. But if V_{IL} were increased, V_{OH} would drop precipitously.

1.6.5 The Static Discipline

To avoid inputs falling into the forbidden zone, digital logic gates are designed to conform to the *static discipline*. The static discipline requires that, given logically valid inputs, every circuit element will produce logically valid outputs.

By conforming to the static discipline, digital designers sacrifice the freedom of using arbitrary analog circuit elements in return for

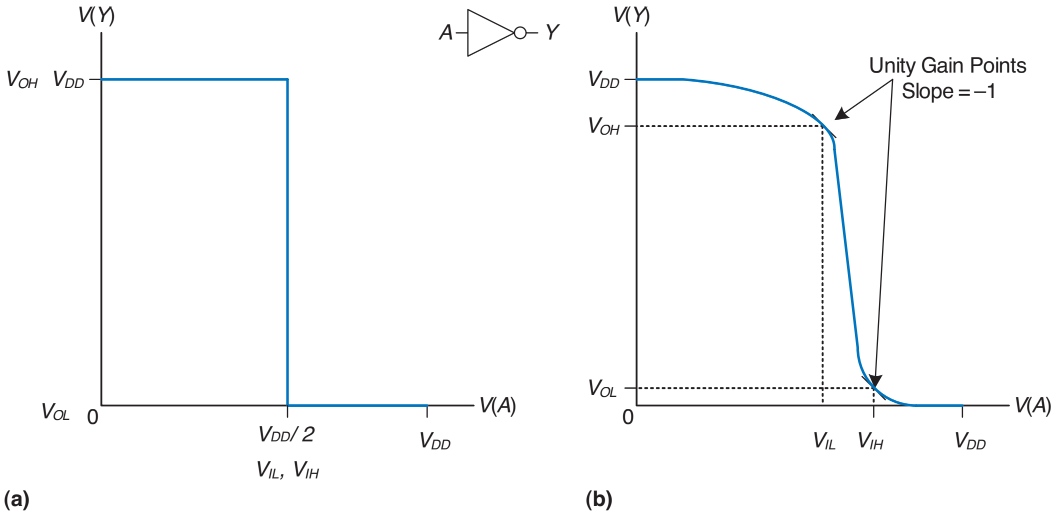


Figure 1.25 DC transfer characteristics and logic levels

the simplicity and robustness of digital circuits. They raise the level of abstraction from analog to digital, increasing design productivity by hiding needless detail.

The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Therefore, gates are grouped into *logic families* such that all gates in a logic family obey the static discipline when used with other gates in the family. Logic gates in the same logic family snap together like Legos in that they use consistent power supply voltages and logic levels.

Four major logic families that predominated from the 1970's through the 1990's are Transistor-Transistor Logic (TTL), Complementary Metal-Oxide-Semiconductor Logic (CMOS, pronounced sea-moss), Low Voltage TTL Logic (LVTTL), and Low Voltage CMOS Logic (LVCMOS). Their logic levels are compared in Table 1.4. Since then, logic families have balkanized with a proliferation of even lower power supply voltages. Appendix A.6 revisits popular logic families in more detail.

Table 1.4 Logic levels of 5 V and 3.3 V logic families

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Table 1.5 **Compatibility of logic families**

	Receiver			
Driver	TTL	CMOS	LVTTL	LVCMOS
TTL	OK	NO: $V_{OH} < V_{IH}$	MAYBE[a]	MAYBE[a]
CMOS	OK	OK	MAYBE[a]	MAYBE[a]
LVTTL	OK	NO: $V_{OH} < V_{IH}$	OK	OK
LVCMOS	OK	NO: $V_{OH} < V_{IH}$	OK	OK

[a] As long as a 5 V HIGH level does not damage the receiver input

Example 1.19 LOGIC FAMILY COMPATIBILITY

Which of the logic families in Table 1.4 can communicate with each other reliably?

Solution Table 1.5 lists which logic families have compatible logic levels. Note that a 5 V logic family such as TTL or CMOS may produce an output voltage as HIGH as 5 V. If this 5 V signal drives the input of a 3.3 V logic family such as LVTTL or LVCMOS, it can damage the receiver unless the receiver is specially designed to be "5-volt compatible."

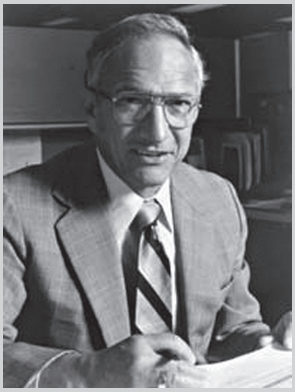
1.7 CMOS TRANSISTORS*

This section and other sections marked with a * are optional and are not necessary to understand the main flow of the book.

Babbage's Analytical Engine was built from gears, and early electrical computers used relays or vacuum tubes. Modern computers use transistors because they are cheap, small, and reliable. *Transistors* are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. The two main types of transistors are *bipolar junction transistors* and *metal-oxide-semiconductor field effect transistors* (*MOSFETs* or *MOS transistors*, pronounced "moss-fets" or "M-O-S", respectively).

In 1958, Jack Kilby at Texas Instruments built the first integrated circuit containing two transistors. In 1959, Robert Noyce at Fairchild Semiconductor patented a method of interconnecting multiple transistors on a single silicon chip. At the time, transistors cost about $10 each.

Thanks to more than three decades of unprecedented manufacturing advances, engineers can now pack roughly one billion MOSFETs onto a 1-cm^2 chip of silicon, and these transistors cost less than 10 microcents apiece. The capacity and cost continue to improve by an order of magnitude every 8 years or so. MOSFETs are now the building blocks of



Robert Noyce, 1927–1990
Born in Burlington, Iowa. Received a B.A. in physics from Grinnell College and a Ph.D. in physics from MIT. Nicknamed "Mayor of Silicon Valley" for his profound influence on the industry.

Cofounded Fairchild Semiconductor in 1957 and Intel in 1968. Coinvented the integrated circuit. Many engineers from his teams went on to found other seminal semiconductor companies (photograph © 2006, Intel Corporation. Reproduced by permission).

almost all digital systems. In this section, we will peer beneath the digital abstraction to see how logic gates are built from MOSFETs.

1.7.1 Semiconductors

MOS transistors are built from silicon, the predominant atom in rock and sand. Silicon (Si) is a group IV atom, so it has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline *lattice*. Figure 1.26(a) shows the lattice in two dimensions for ease of drawing, but remember that the lattice actually forms a cubic crystal. In the figure, a line represents a covalent bond. By itself, silicon is a poor conductor because all the electrons are tied up in covalent bonds. However, it becomes a better conductor when small amounts of impurities, called *dopant* atoms, are carefully added. If a group V dopant such as arsenic (As) is added, the dopant atoms have an extra electron that is not involved in the bonds. That electron can easily move about the lattice, leaving an ionized dopant atom (As^+) behind, as shown in Figure 1.26(b). The electron carries a negative charge, so we call arsenic an *n-type* dopant. On the other hand, if a group III dopant such as boron (B) is added, the dopant atoms are missing an electron, as shown in Figure 1.26(c). This missing electron is called a *hole*. An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B^-) and leaving a hole at the neighboring silicon atom. In a similar fashion, the hole can migrate around the lattice. The hole is a lack of negative charge, so it acts like a positively charged particle. Hence, we call boron a *p-type* dopant. Because the conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants, silicon is called a *semiconductor*.

1.7.2 Diodes

The junction between p-type and n-type silicon is called a *diode*. The p-type region is called the *anode* and the n-type region is called the *cathode*, as illustrated in Figure 1.27. When the voltage on the anode rises

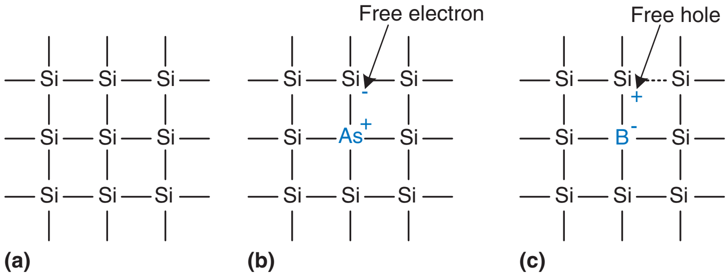


Figure 1.26 Silicon lattice and dopant atoms

above the voltage on the cathode, the diode is *forward biased*, and current flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is *reverse biased*, and no current flows. The diode symbol intuitively shows that current only flows in one direction.

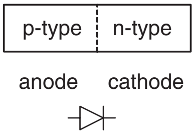


Figure 1.27 The p-n junction diode structure and symbol

1.7.3 Capacitors

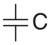


Figure 1.28 Capacitor symbol

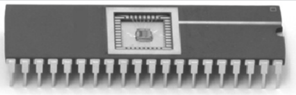
A *capacitor* consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$. The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = Q/V$. The capacitance is proportional to the size of the conductors and inversely proportional to the distance between them. The symbol for a capacitor is shown in Figure 1.28.

Capacitance is important because charging or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate. Speed and energy will be discussed throughout this book.

Technicians in an Intel clean room wear Gore-Tex bunny suits to prevent particulates from their hair, skin, and clothing from contaminating the microscopic transistors on silicon wafers (photograph © 2006, Intel Corporation. Reproduced by permission).

1.7.4 nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin, flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron[1] in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called *chips* or *dice* that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.



A 40-pin dual-inline package (DIP) contains a small chip (scarcely visible) in the center that is connected to 40 metal pins, 20 on a side, by gold wires thinner than a strand of hair (photograph by Kevin Mapp. © Harvey Mudd College).

The MOSFET sandwich consists of a conducting layer called the *gate* on top of an insulating layer of *silicon dioxide* (SiO_2) on top of the silicon wafer, called the *substrate*. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called *oxide*

[1] 1 μm = 1 micron = 10^{-6} m.

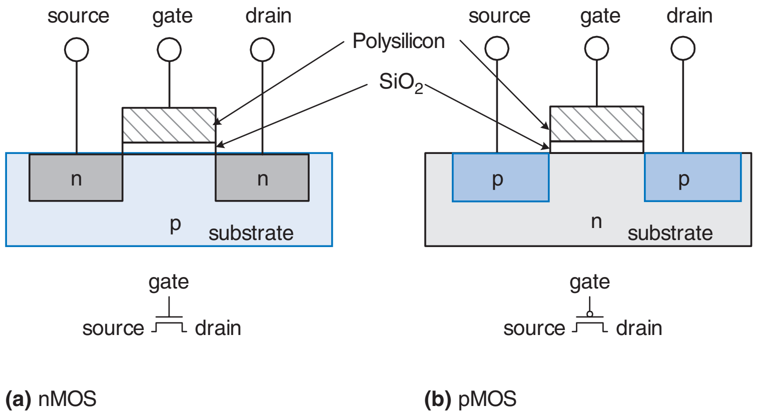


Figure 1.29 nMOS and pMOS transistors

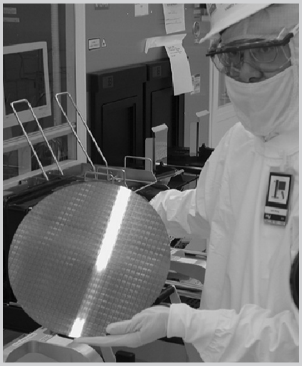
in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a *dielectric* separates the metal and semiconductor plates.

There are two flavors of MOSFETs: nMOS and pMOS (pronounced "n-moss" and "p-moss"). Figure 1.29 shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent to the gate called the *source* and the *drain* and are built on a p-type semiconductor substrate. The *pMOS* transistors are just the opposite, consisting of p-type source and drain regions in an n-type *substrate*.

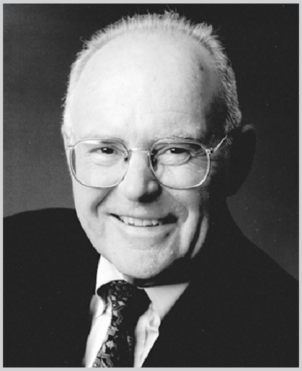
A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, the lowest voltage in the system. First, consider the situation when the gate is also at 0V, as shown in Figure 1.30(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in Figure 1.30(b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the *channel*. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from

The source and drain terminals are physically symmetric. However, we say that charge flows from the source to the drain. In an nMOS transistor, the charge is carried by electrons, which flow from negative voltage to positive voltage. In a pMOS transistor, the charge is carried by holes, which flow from positive voltage to negative voltage. If we draw schematics with the most positive voltage at the top and the most negative at the bottom, the source of (negative) charges in an nMOS transistor is the bottom terminal and the source of (positive) charges in a pMOS transistor is the top terminal.



A technician holds a 12-inch wafer containing hundreds of microprocessor chips (photograph © 2006, Intel Corporation. Reproduced by permission).



Gordon Moore, 1929–
Born in San Francisco. Received a B.S. in chemistry from UC Berkeley and a Ph.D. in chemistry and physics from Caltech. Cofounded Intel in 1968 with Robert Noyce. Observed in 1965 that the number of transistors on a computer chip doubles every year. This trend has become known as *Moore's Law*. Since 1975, transistor counts have doubled every two years.

A corollary of Moore's Law is that microprocessor performance doubles every 18 to 24 months. Semiconductor sales have also increased exponentially. Unfortunately, power consumption has increased exponentially as well.

Moore's Law has driven the incredible advances of the semiconductor industry for 50 years as the feature size of transistors has dropped from more than 10 μm to only 5 nm. However, this progress has slowed because building transistors much smaller than the wavelength of light is expensive. (Photograph © 2006, Intel Corporation. Reproduced by permission.)

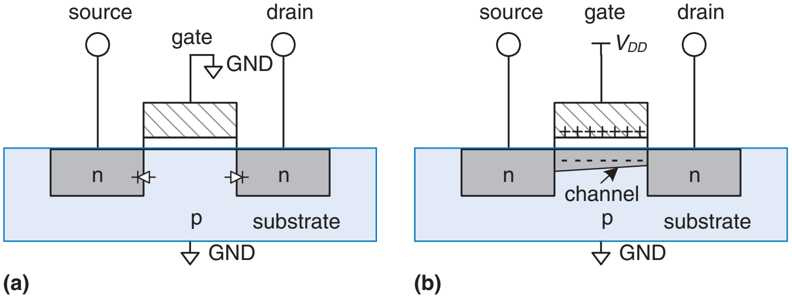


Figure 1.30 nMOS transistor operation

source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the *threshold voltage*, V_t, and is typically 0.3 to 0.7V.

pMOS transistors work in just the opposite fashion, as might be guessed from the bubble on their symbol shown in Figure 1.31. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the source will only swing between 0 and $V_{DD} - V_t$ when its drain ranges from 0 to V_{DD}. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called *wells* where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or CMOS. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS processes give us two types of electrically controlled switches, as shown in Figure 1.31. The voltage at the

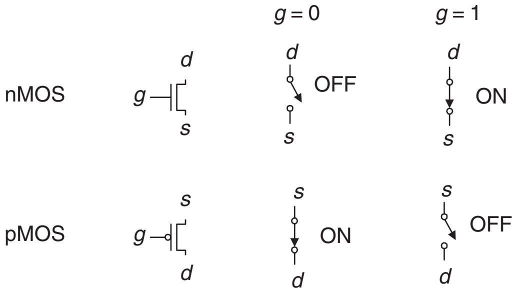


Figure 1.31 Switch models of MOSFETs

gate (g) regulates the flow of current between the source (s) and drain (d). nMOS transistors are OFF when the gate is 0 and ON when the gate is 1. pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

1.7.5 CMOS NOT Gate

Figure 1.32 shows a schematic of a NOT gate built with CMOS transistors. The triangle indicates GND, and the flat bar indicates V_{DD}; these labels will be omitted from future schematics. The nMOS transistor, N1, is connected between GND and the Y output. The pMOS transistor, P1, is connected between V_{DD} and the Y output. Both transistor gates are controlled by the input, A.

If $A = 0$, N1 is OFF and P1 is ON. Hence, Y is connected to V_{DD} but not to GND, and is pulled up to a logic 1. P1 passes a good 1. If $A = 1$, N1 is ON and P1 is OFF, and Y is pulled down to a logic 0. N1 passes a good 0. Checking against the truth table in Figure 1.12, we see that the circuit is indeed a NOT gate.

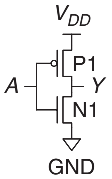


Figure 1.32 NOT gate schematic

1.7.6 Other CMOS Logic Gates

Figure 1.33 shows a schematic of a two-input NAND gate. In schematic diagrams, wires are always joined at three-way junctions. They are joined at four-way junctions only if a dot is shown. The nMOS transistors N1 and N2 are connected in series; both nMOS transistors must be ON to pull the output down to GND. The pMOS transistors P1 and P2 are in parallel; only one pMOS transistor must be ON to pull the output up to V_{DD}. Table 1.6 lists the operation of the pull-down and pull-up networks and the state of the output, demonstrating that the gate does function as a NAND. For example, when $A = 1$ and $B = 0$, N1 is ON, but N2 is OFF, blocking the path from Y to GND. P1 is OFF, but P2 is ON, creating a path from V_{DD} to Y. Therefore, Y is pulled up to 1.

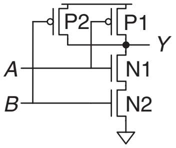


Figure 1.33 Two-input NAND gate schematic

Figure 1.34 shows the general form used to construct any inverting logic gate, such as NOT, NAND, or NOR. nMOS transistors are good at passing 0's, so a pull-down network of nMOS transistors is placed

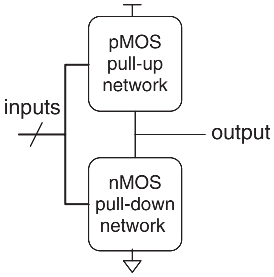


Figure 1.34 General form of an inverting logic gate

Table 1.6 NAND gate operation

A	B	Pull-Down Network	Pull-Up Network	Y
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

Experienced designers claim that electronic devices operate because they contain *magic smoke*. They confirm this theory with the observation that if the magic smoke is ever let out of the device, it ceases to work.

between the output and GND to pull the output down to 0. pMOS transistors are good at passing 1's, so a pull-up network of pMOS transistors is placed between the output and V_{DD} to pull the output up to 1. The networks may consist of transistors in series or in parallel. When transistors are in parallel, the network is ON if either transistor is ON. When transistors are in series, the network is ON only if both transistors are ON. The slash across the input wire indicates that the gate may receive multiple inputs.

If both the pull-up and pull-down networks were ON simultaneously, a *short circuit* would exist between V_{DD} and GND. The output of the gate might be in the forbidden zone and the transistors would consume large amounts of power, possibly enough to burn out. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output *floats*. Its value is again undefined. Floating outputs are usually undesirable, but in Section 2.6 we will see how they can occasionally be used to the designer's advantage.

In a properly functioning logic gate, one of the networks should be ON and the other OFF at any given time, so that the output is pulled HIGH or LOW but not shorted or floating. We can guarantee this by using the rule of *conduction complements*. When nMOS transistors are in series, the pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

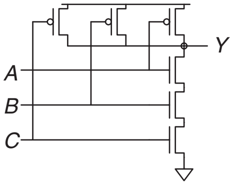


Figure 1.35 **Three-input NAND gate schematic**

Example 1.20 THREE-INPUT NAND SCHEMATIC

Draw a schematic for a three-input NAND gate using CMOS transistors.

Solution The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel. Such a gate is shown in Figure 1.35; you can verify the function by checking that it has the correct truth table.

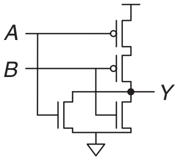


Figure 1.36 **Two-input NOR gate schematic**

Example 1.21 TWO-INPUT NOR SCHEMATIC

Draw a schematic for a two-input NOR gate using CMOS transistors.

Solution The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series. Such a gate is shown in Figure 1.36.

Example 1.22 TWO-INPUT AND SCHEMATIC

Draw a schematic for a two-input AND gate.

Solution It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT, as shown in Figure 1.37.

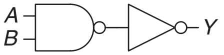


Figure 1.37 Two-input AND gate schematic

1.7.7 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well. Recall that nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well. Figure 1.38 shows such a circuit, called a *transmission gate* or *pass gate*. The two sides of the switch are called *A* and *B* because a switch is bidirectional and has no preferred input or output side. The control signals are called *enables*, *EN* and $\overline{EN}$. When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so *A* and *B* are not connected. When $EN = 1$ and $\overline{EN} = 0$, the transmission gate is ON or enabled, and any logic value can flow between *A* and *B*.

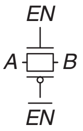


Figure 1.38 Transmission gate

1.7.8 Pseudo-nMOS Logic

An *N*-input CMOS NOR gate uses *N* nMOS transistors in parallel and *N* pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel. Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore, the parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.39. This pMOS transistor is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output *Y* HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls *Y* down close enough to GND to produce a logic 0.

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs. For example, Figure 1.40 shows a pseudo-nMOS four-input NOR. Pseudo-nMOS gates are useful for certain memory and logic arrays discussed in Chapter 5. The disadvantage is that

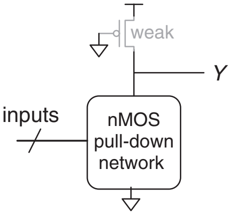


Figure 1.39 Generic pseudo-nMOS gate

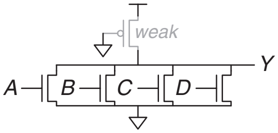


Figure 1.40 Pseudo-nMOS four-input NOR gate

a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

Pseudo-nMOS gates got their name from the 1970's, when manufacturing processes only had nMOS transistors. A weak nMOS transistor was used to pull the output HIGH because pMOS transistors were not available.

1.8 POWER CONSUMPTION*

Power consumption is the amount of energy used per unit time. Power consumption is of great importance in digital systems. The battery life of portable systems such as cell phones and laptop computers is limited by power consumption. For example, a cell phone battery holds about 10 watt-hours (W-hr) of energy, meaning that it could deliver 1 watt (W) for 10 hours or 2 W for 5 hours, and so forth. For your phone battery to last a full day, its average consumption should be under 1 W. Laptops typically have 50 to 100 W-hr batteries and consume under 10 W in normal operation, of which the screen is a large fraction. Power is also significant for systems that are plugged in because electricity costs money and emissions and because the system will overheat if it draws too much power. A desktop computer consuming 200 W for 8 hours each day would use approximately 600 kilowatt-hours (kW-hr) of electricity per year. At an average cost of 12 cents and one pound of CO_2 emission per kW-hr, this is $72 of electricity each year as well as 600 pounds of CO_2 emissions.

You'll help our planet by making sure your computer sleeps when you aren't using it.

Digital systems draw both *dynamic* and *static* power. Dynamic power is the power used to charge capacitance as signals change between 0 and 1. Static power is the power used even when signals do not change and the system is idle.

Logic gates and the wires that connect them have capacitance. The energy drawn from the power supply to charge a capacitance C to voltage V_{DD} is CV_{DD}^2. If the system operates at *frequency* f and the fraction of the cycles on which the capacitor charges and discharges is α (called the *activity factor*), the dynamic power consumption is

$$P_{\text{dynamic}} = \alpha C V_{DD}^2 f \tag{1.4}$$

Figure 1.41 illustrates activity factors. Figure 1.41(a) shows a clock signal, which rises and falls once every cycle and, thus, has an activity factor of 1. The *clock period* from one rising edge to the next is called T_c (the *cycle time*), and is the reciprocal of the clock frequency f. Figure 1.41(b) shows a data signal that switches once every clock cycle. The parallel lines on the timing diagram indicate that the signal might be high or might be low; we aren't concerned with its value. The crossover indicates that the signal changes once, early in each clock cycle. Hence, the activity

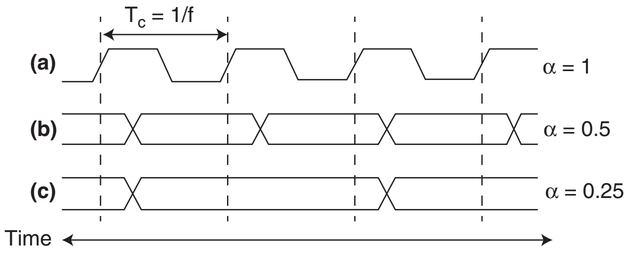


Figure 1.41 **Illustration of activity factors**

factor is 0.5 (rising half the cycles and falling half the cycles). Figure 1.41(c) shows a random data signal that switches in half the cycles and remains constant in the other half of the cycles. Therefore, it has an activity factor of 0.25. Real digital systems often have idle components that are not switching, so an activity factor of 0.1 is more typical.

Electrical systems draw some current even when they are idle. When transistors are OFF, they leak a small amount of current. Some circuits, such as the pseudo-nMOS gate discussed in Section 1.7.8, have a path from V_{DD} to GND through which current flows continuously. The total static current, I_{DD}, is also called the *leakage current* or the *quiescent supply current* flowing between V_{DD} and GND. The static power consumption is proportional to this static current:

$$P_{\text{static}} = I_{DD}V_{DD} \tag{1.5}$$

Example 1.23 POWER CONSUMPTION

A particular cell phone has an 8 W-hr battery and operates at 0.707 V. Suppose that, when it is in use, the cell phone operates at 2 GHz. The total capacitance of the circuitry is 10 nF (10^{-8} Farads), and the activity factor is 0.05. When voice or data are active (10% of its time in use), it also broadcasts 3 W of power out of its antenna. When the phone is not in use, the dynamic power drops to almost zero because the signal processing is turned off. But the phone also draws 100 mA of quiescent current whether it is in use or not. Determine the battery life of the phone (a) if it is not being used and (b) if it is being used continuously.

Solution The static power is P_{static} = (0.100 A)(0.707 V) = 71 milliwatts (mW). (a) If the phone is not being used, this is the only power consumption, so the battery life is (8 W-hr)/(0.071 W) = 113 hours (about 5 days). (b) If the phone is being used, the dynamic power is $P_{\text{dynamic}} = (0.05)(10^{-8}\text{ F})(0.707\text{ V})^2(2 \times 10^9\text{ Hz})$ = 0.5 W. The average broadcast power is (3 W)(0.1) = 0.3 W.

Together with the static and broadcast power, the total active power is 0.5 W + 0.071 W + 0.3 W = 0.871 W, so the battery life is 8 W-hr/0.0871 W = 9.2 hours. This example somewhat oversimplifies the actual operation of a cell phone, but it illustrates the key ideas of power consumption.

1.9 SUMMARY AND A LOOK AHEAD

There are 10 kinds of people in this world: those who can count in binary and those who can't.

This chapter has introduced principles for understanding and designing complex systems. Although the real world is analog, digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH. Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

- **NOT:** Output is TRUE when input is FALSE
- **AND:** Output is TRUE when all inputs are TRUE
- **OR:** Output is TRUE when any inputs are TRUE
- **XOR:** Output is TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches. nMOS transistors turn ON when the gate is 1. pMOS transistors turn ON when the gate is 0.

In Chapters 2 through 5, we continue the study of digital logic. Chapter 2 addresses *combinational logic*, in which the outputs depend only on the current inputs. The logic gates introduced already are examples of combinational logic. You will learn to design circuits involving multiple gates to implement a relationship between inputs and outputs specified by a truth table or Boolean equation. Chapter 3 addresses *sequential logic*, in which the outputs depend on both current and past inputs. *Registers* are common sequential elements that remember their previous input. *Finite state machines*, built from registers and combinational logic, are a powerful way to build complicated systems in a systematic fashion. We also study timing of digital systems to analyze how fast a system can operate. Chapter 4 describes hardware description languages (HDLs). HDLs are related to conventional programming languages but are used to simulate and build hardware rather than software. Most digital systems today are designed with HDLs. SystemVerilog and VHDL are the two prevalent languages; they are covered side-by-side in this book. Chapter 5 studies other combinational and sequential building blocks, such as adders, multipliers, and memories.

Chapter 6 shifts to computer architecture. It describes the RISC-V processor, a recently developed open-source microprocessor beginning to see wide development across industry and academia. The RISC-V architecture is defined by its registers and assembly language instruction set. You will learn to write programs in assembly language for the RISC-V processor so that you can communicate with the processor in its native language.

Chapters 7 and 8 bridge the gap between digital logic and computer architecture. Chapter 7 investigates microarchitecture, the arrangement of digital building blocks, such as adders and registers, needed to construct a processor. In that chapter, you learn to build your own RISC-V processor. Indeed, you learn three microarchitectures illustrating different trade-offs of performance and cost. Processor performance has increased exponentially, requiring ever more sophisticated memory systems to feed the insatiable demand for data. Chapter 8 delves into memory system architecture. Chapter 9 (available as a web supplement—see Preface) describes how computers communicate with peripheral devices, such as monitors, Bluetooth radios, and motors.

Exercises

Exercise 1.1 Explain in one paragraph at least three levels of abstraction that are used by

(a) biologists studying the operation of cells.

(b) chemists studying the composition of matter.

Exercise 1.2 Explain in one paragraph how the techniques of hierarchy, modularity, and regularity may be used by

(a) automobile designers.

(b) businesses to manage their operations.

Exercise 1.3 Ben Bitdiddle is building a house. Explain how he can use the principles of hierarchy, modularity, and regularity to save time and money during construction.

Exercise 1.4 An analog voltage is in the range of 0–5 V. If it can be measured with an accuracy of ±50 mV, at most how many bits of information does it convey?

Exercise 1.5 A classroom has an old clock on the wall whose minute hand broke off.

(a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time?

(b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

Exercise 1.6 The Babylonians developed the *sexagesimal* (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the number 4000_{10} in sexagesimal?

Exercise 1.7 How many different numbers can be represented with 16 bits?

Exercise 1.8 What is the largest unsigned 32-bit binary number?

Exercise 1.9 What is the largest 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.10 What is the largest 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.11 What is the smallest (most negative) 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.12 What is the smallest (most negative) 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.13 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1010_2

(b) 110110_2

(c) 11110000_2

(d) 0001000101001112_2

Exercise 1.14 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1110_2

(b) 100100_2

(c) 11010111_2

(d) 011101010100100_2

Exercise 1.15 Repeat Exercise 1.13, but convert to hexadecimal.

Exercise 1.16 Repeat Exercise 1.14, but convert to hexadecimal.

Exercise 1.17 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $A5_{16}$

(b) $3B_{16}$

(c) $FFFF_{16}$

(d) $D0000000_{16}$

Exercise 1.18 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $4E_{16}$

(b) $7C_{16}$

(c) $ED3A_{16}$

(d) $403FB001_{16}$

Exercise 1.19 Repeat Exercise 1.17, but convert to unsigned binary.

Exercise 1.20 Repeat Exercise 1.18, but convert to unsigned binary.

Exercise 1.21 Convert the following two's complement binary numbers to decimal.

(a) 1010_2

(b) 110110_2

(c) 01110000_2

(d) 10011111_2

Exercise 1.22 Convert the following two's complement binary numbers to decimal.

(a) 1110_2

(b) 100011_2

(c) 01001110_2

(d) 10110101_2

Exercise 1.23 Repeat Exercise 1.21, assuming that the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.24 Repeat Exercise 1.22, assuming that the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.25 Convert the following decimal numbers to unsigned binary numbers.

(a) 42_{10}

(b) 63_{10}

(c) 229_{10}

(d) 845_{10}

Exercise 1.26 Convert the following decimal numbers to unsigned binary numbers.

(a) 14_{10}

(b) 52_{10}

(c) 339_{10}

(d) 711_{10}

Exercise 1.27 Repeat Exercise 1.25, but convert to hexadecimal.

Exercise 1.28 Repeat Exercise 1.26, but convert to hexadecimal.

Exercise 1.29 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 42_{10}

(b) -63_{10}

(c) 124_{10}

(d) -128_{10}

(e) 133_{10}

Exercise 1.30 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 24_{10}

(b) -59_{10}

(c) 128_{10}

(d) -150_{10}

(e) 127_{10}

Exercise 1.31 Repeat Exercise 1.29, but convert to 8-bit sign/magnitude numbers.

Exercise 1.32 Repeat Exercise 1.30, but convert to 8-bit sign/magnitude numbers.

Exercise 1.33 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0101_2

(b) 1010_2

Exercise 1.34 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0111_2

(b) 1001_2

Exercise 1.35 Repeat Exercise 1.33 if the numbers are unsigned rather than two's complement.

Exercise 1.36 Repeat Exercise 1.34 if the numbers are unsigned rather than two's complement.

Exercise 1.37 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.25 to octal.

Exercise 1.38 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.26 to octal.

Exercise 1.39 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 42_8

(b) 63_8

(c) 255_8

(d) 3047_8

Exercise 1.40 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 23_8

(b) 45_8

(c) 371_8

(d) 2560_8

Exercise 1.41 How many 5-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.42 How many 7-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.43 How many bytes are in a 32-bit word? How many nibbles are in the 32-bit word?

Exercise 1.44 How many bytes are in a 64-bit word?

Exercise 1.45 A particular DSL modem operates at 768 kbits/sec. How many bytes can it receive in 1 minute?

Exercise 1.46 USB 3.0 can send data at 5 Gbits/sec. How many bytes can it send in 1 minute?

Exercise 1.47 Hard drive manufacturers use the term "megabyte" to mean 10^6 bytes and "gigabyte" to mean 10^9 bytes. How many real GBs (i.e., GiBs) of music can you store on a 50 GB hard drive?

Exercise 1.48 Estimate the value of 2^{31} without using a calculator.

Exercise 1.49 A memory on the Pentium II microprocessor is organized as a rectangular array of bits with 2^8 rows and 2^9 columns. Estimate how many bits it has without using a calculator.

Exercise 1.50 Draw a number line analogous to Figure 1.11 for 3-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.51 Draw a number line analogous to Figure 1.11 for 2-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.52 Perform the following additions of unsigned binary numbers. Indicate whether the sum overflows a 4-bit result.

(a) $1001_2 + 0100_2$

(b) $1101_2 + 1011_2$

Exercise 1.53 Perform the following additions of unsigned binary numbers. Indicate whether the sum overflows an 8-bit result.

(a) $10011001_2 + 01000100_2$

(b) $11010010_2 + 10110110_2$

Exercise 1.54 Repeat Exercise 1.52, assuming that the binary numbers are in two's complement form.

Exercise 1.55 Repeat Exercise 1.53, assuming that the binary numbers are in two's complement form.

Exercise 1.56 Convert the following decimal numbers to 6-bit two's complement binary numbers and add them. Indicate whether the sum overflows a 6-bit result.

(a) $16_{10} + 9_{10}$

(b) $27_{10} + 31_{10}$

(c) $-4_{10} + 19_{10}$

(d) $3_{10} + -32_{10}$

(e) $-16_{10} + -9_{10}$

(f) $-27_{10} + -31_{10}$

Exercise 1.57 Repeat Exercise 1.56 for the following numbers.

(a) $7_{10} + 13_{10}$

(b) $17_{10} + 25_{10}$

(c) $-26_{10} + 8_{10}$

(d) $31_{10} + -14_{10}$

(e) $-19_{10} + -22_{10}$

(f) $-2_{10} + -29_{10}$

Exercise 1.58 Perform the following additions of unsigned hexadecimal numbers. Indicate whether the sum overflows an 8-bit (two hex digit) result.

(a) $7_{16} + 9_{16}$

(b) $13_{16} + 28_{16}$

(c) $AB_{16} + 3E_{16}$

(d) $8F_{16} + AD_{16}$

Exercise 1.59 Perform the following additions of unsigned hexadecimal numbers. Indicate whether the sum overflows an 8-bit (two hex digit) result.

(a) $22_{16} + 8_{16}$

(b) $73_{16} + 2C_{16}$

(c) $7F_{16} + 7F_{16}$

(d) $C2_{16} + A4_{16}$

Exercise 1.60 Convert the following decimal numbers to 5-bit two's complement binary numbers and subtract them. Indicate whether the difference overflows a 5-bit result.

(a) $9_{10} - 7_{10}$

(b) $12_{10} - 15_{10}$

(c) $-6_{10} - 11_{10}$

(d) $4_{10} - -8_{10}$

Exercise 1.61 Convert the following decimal numbers to 6-bit two's complement binary numbers and subtract them. Indicate whether the difference overflows a 6-bit result.

(a) $18_{10} - 12_{10}$

(b) $30_{10} - 9_{10}$

(c) $-28_{10} - 3_{10}$

(d) $-16_{10} - 21_{10}$

Exercise 1.62 In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111, 1 as 10000, and so forth. Biased number systems are sometimes used in floating-point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}.

(a) What decimal value does the binary number 10000010_2 represent?

(b) What binary number represents the value 0?

(c) What is the representation and value of the most negative number?

(d) What is the representation and value of the most positive number?

Exercise 1.63 Draw a number line analogous to Figure 1.11 for 3-bit biased numbers with a bias of 3 (see Exercise 1.62 for a definition of biased numbers).

Exercise 1.64 In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

(a) Write 289_{10} in BCD.

(b) Convert 100101010001_{BCD} to decimal.

(c) Convert 01101001_{BCD} to binary.

(d) Explain why BCD might be a useful way to represent numbers.

Exercise 1.65 Answer the following questions related to BCD systems (see Exercise 1.64 for the definition of BCD).

(a) Write 371_{10} in BCD.

(b) Convert 000110000111_{BCD} to decimal.

(c) Convert 10010101_{BCD} to binary.

(d) Explain the disadvantages of BCD when compared with binary representations of numbers.

Exercise 1.66 A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number system: 325 + 42 = 411. If this equation is correct, how many fingers would you expect Martians to have?

Exercise 1.67 Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, "All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation." Alyssa disagrees. She says, "No, but all such numbers have an even number of 1's in their representation." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.68 Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, "I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result." Alyssa says, "No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.69 Write a program in your favorite language (e.g., C, Java, Perl) to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should print the decimal equivalent.

Exercise 1.70 Repeat Exercise 1.69 but convert from an arbitrary base b_1 to another base b_2, as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9. The user should enter b_1, b_2, and then the number to convert in base b_1. The program should print the equivalent number in base $b_{2.}$

Exercise 1.71 Draw the symbol, Boolean equation, and truth table for

(a) a three-input OR gate

(b) a three-input exclusive OR (XOR) gate

(c) a four-input XNOR gate

Exercise 1.72 Draw the symbol, Boolean equation, and truth table for

(a) a four-input OR gate

(b) a three-input XNOR gate

(c) a five-input NAND gate

Exercise 1.73 A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.42.

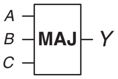


Figure 1.42 **Three-input majority gate**

Exercise 1.74 A three-input *AND-OR* (*AO*) *gate* shown in Figure 1.43 produces a TRUE output if both *A* and *B* are TRUE or if *C* is TRUE. Complete a truth table for the gate.

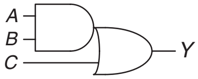


Figure 1.43 **Three-input AND-OR gate**

Exercise 1.75 A three-input *OR-AND-INVERT* (*OAI*) *gate* shown in Figure 1.44 produces a FALSE output if *C* is TRUE and *A* or *B* is TRUE. Otherwise, it produces a TRUE output. Complete a truth table for the gate.

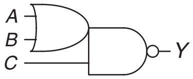


Figure 1.44 **Three-input OR-AND-INVERT gate**

Exercise 1.76 There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

Exercise 1.77 How many different truth tables exist for Boolean functions of *N* variables?

Exercise 1.78 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.45 would serve as an inverter? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and N_H)? If not, explain why not.

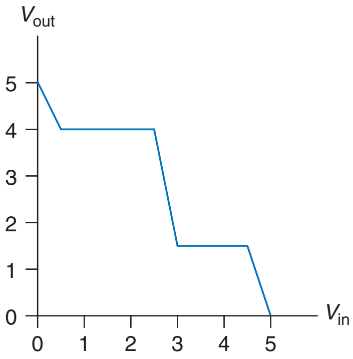


Figure 1.45 **DC transfer characteristics**

Exercise 1.79 Repeat Exercise 1.78 for the transfer characteristics shown in Figure 1.46.

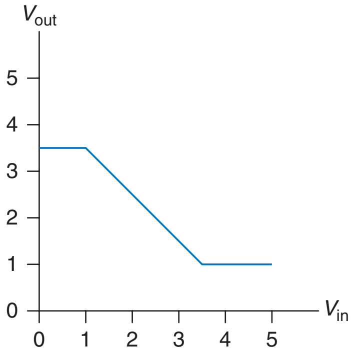


Figure 1.46 **DC transfer characteristics**

Exercise 1.80 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.47 would serve as a buffer? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

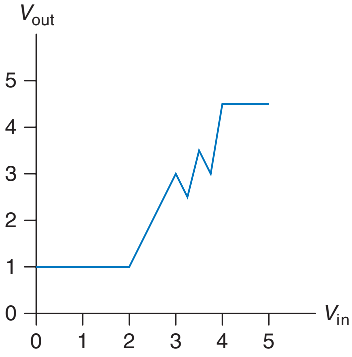


Figure 1.47 **DC transfer characteristics**

Exercise 1.81 Ben Bitdiddle has invented a circuit with the transfer characteristics shown in Figure 1.48 that he would like to use as a buffer. Will it work? Why or why not? He would like to advertise that it is compatible with LVCMOS and LVTTL logic. Can Ben's buffer correctly receive inputs from those logic families? Can its output properly drive those logic families? Explain.

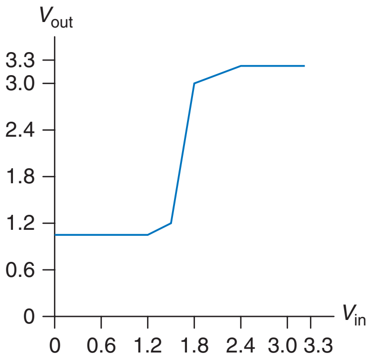


Figure 1.48 Ben's buffer DC transfer characteristics

Exercise 1.82 While walking down a dark alley, Ben Bitdiddle encounters a two-input gate with the transfer function shown in Figure 1.49. The inputs are A and B and the output is Y.

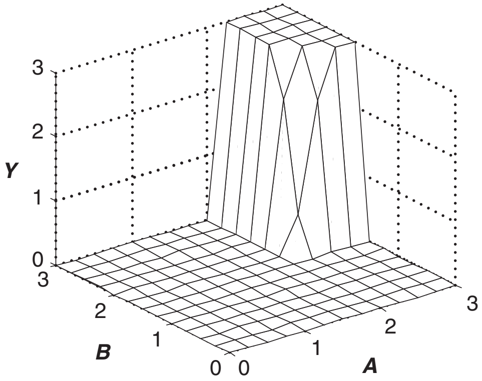


Figure 1.49 Two-input DC transfer characteristics

(a) What kind of logic gate did he find?

(b) What are the approximate high and low logic levels?

Exercise 1.83 Repeat Exercise 1.82 for Figure 1.50.

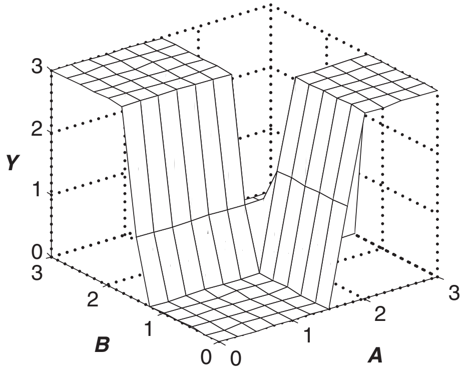


Figure 1.50 Two-input DC transfer characteristics

Exercise 1.84 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) four-input NAND gate

(b) three-input OR-AND-INVERT gate (see Exercise 1.75)

(c) three-input AND-OR gate (see Exercise 1.74)

Exercise 1.85 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input AND gate

(c) two-input OR gate

Exercise 1.86 A *minority gate* produces a TRUE output if and only if fewer than half of its inputs are TRUE. Otherwise, it produces a FALSE output. Sketch a transistor-level circuit for a three-input CMOS minority gate. Use a minimum number of transistors.

Exercise 1.87 Write a truth table for the function performed by the gate in Figure 1.51. The truth table should have two inputs, *A* and *B*. What is the name of this function?

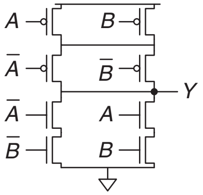


Figure 1.51 Mystery schematic

Exercise 1.88 Write a truth table for the function performed by the gate in Figure 1.52. The truth table should have three inputs, *A*, *B*, and C.

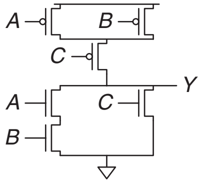


Figure 1.52 Mystery schematic

Exercise 1.89 Implement the following three-input gates using only pseudo-nMOS logic gates. Your gates receive three inputs, *A*, *B*, and C. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input NAND gate

(c) three-input AND gate

Exercise 1.90 *Resistor-Transistor Logic* (*RTL*) uses nMOS transistors to pull the gate output LOW and a weak resistor to pull the output HIGH when none of the paths to ground are active. A NOT gate built using RTL is shown in Figure 1.53. Sketch a three-input RTL NOR gate. Use a minimum number of transistors.

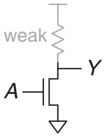


Figure 1.53 **RTL NOT gate**

Interview Questions

These questions have been asked at interviews for digital design jobs.

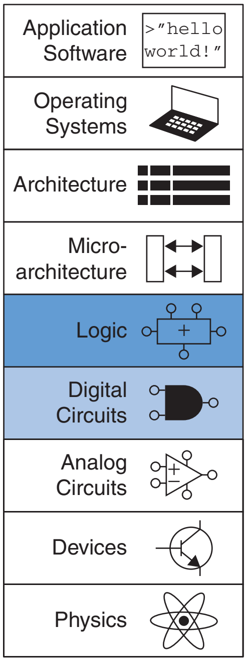
Question 1.1 Sketch a transistor-level circuit for a CMOS four-input NOR gate.

Question 1.2 The king receives 64 gold coins in taxes but has reason to believe that one is counterfeit. He summons you to identify the fake coin. You have a balance that can hold coins on each side. How many times do you need to use the balance to find the lighter, fake coin?

Question 1.3 The professor, the teaching assistant, the digital design student, and the freshman track star need to cross a rickety bridge on a dark night. The bridge is so shaky that only two people can cross at a time. They have only one flashlight among them and the span is too long to throw the flashlight, so somebody must carry it back to the other people. The freshman track star can cross the bridge in 1 minute. The digital design student can cross the bridge in 2 minutes. The teaching assistant can cross the bridge in 5 minutes. The professor always gets distracted and takes 10 minutes to cross the bridge. What is the fastest time to get everyone across the bridge?

Combinational Logic Design

2



2.1 INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and outputs responding.

Peering inside the black box, circuits are composed of nodes and elements. An *element* is itself a circuit with inputs, outputs, and a specification. A *node* is a wire, whose voltage conveys a discrete-valued variable. Nodes are classified as *input*, *output*, or *internal*. Inputs receive values from the external world. Outputs deliver values to the external world. Wires that are not inputs or outputs are called internal nodes. Figure 2.2

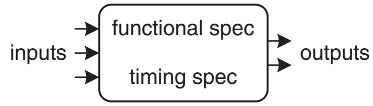


Figure 2.1 Circuit as a black box with inputs, outputs, and specifications

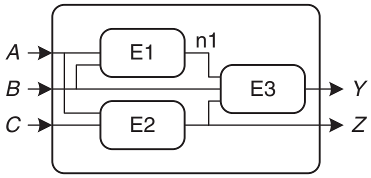
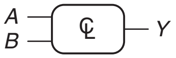


Figure 2.2 Elements and nodes

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00002-7

illustrates a circuit with three elements, E1, E2, and E3, and six nodes. Nodes A, B, and C are inputs. Y and Z are outputs. n1 is an internal node between E1 and E3.

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence. A combinational circuit is *memoryless*, but a sequential circuit has *memory*. This chapter focuses on combinational circuits, and Chapter 3 examines sequential circuits.



$Y = F(A, B) = A + B$

Figure 2.3 Combinational logic circuit

The functional specification of a combinational circuit expresses the output values in terms of the current input values. The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. We will initially concentrate on the functional specification, then return to the timing specification later in this chapter.

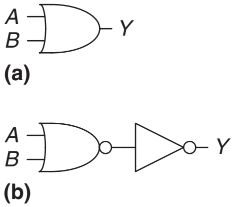
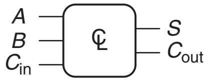


Figure 2.4 Two OR implementations

Figure 2.3 shows a combinational circuit with two inputs and one output. On the left of the figure are the inputs, A and B, and on the right is the output, Y. The symbol ℄ inside the box indicates that it is implemented using only combinational logic. In this example, the function F is specified to be OR: $Y = F(A, B) = A + B$. In words, we say that the output Y is a function of the two inputs, A and B—namely, $Y =$ A OR B.

Figure 2.4 shows two possible *implementations* for the combinational logic circuit in Figure 2.3. As we will see repeatedly throughout the book, there are often many implementations for a single function. You choose which to use given the building blocks at your disposal and your design constraints. These constraints often include area, speed, power, and design time.



$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2.5 Multiple-output combinational circuit

Figure 2.5 shows a combinational circuit with multiple outputs. This particular combinational circuit is called a *full adder*, which we will revisit in Section 5.2.1. The two equations specify the function of the outputs, S and C_{out}, in terms of the inputs, A, B, and C_{in}.

To simplify drawings, we often use a single line with a slash through it and a number next to it to indicate a *bus*, a bundle of multiple signals. The number specifies how many signals are in the bus. For example, Figure 2.6(a) represents a block of combinational logic with three inputs and two outputs. If the number of bits is unimportant or obvious from the context, the slash may be shown without a number. Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.

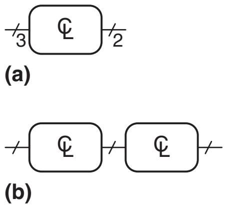


Figure 2.6 Slash notation for multiple signals

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit elements.

A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

The rules of combinational composition are sufficient but not strictly necessary. Certain circuits that disobey these rules are still combinational, so long as the outputs depend only on the current values of the inputs. However, determining whether oddball circuits are combinational is more difficult, so we will usually restrict ourselves to combinational composition as a way to build combinational circuits.

Example 2.1 COMBINATIONAL CIRCUITS

Which of the circuits in Figure 2.7 are combinational circuits according to the rules of combinational composition?

Solution Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

Large circuits such as microprocessors can be very complicated, so we use the principles from Chapter 1 to manage the complexity. Viewing a circuit as a black box with a well-defined interface and function is an application of abstraction and modularity. Building the circuit out of smaller circuit elements is an application of hierarchy. The rules of combinational composition are an application of discipline.

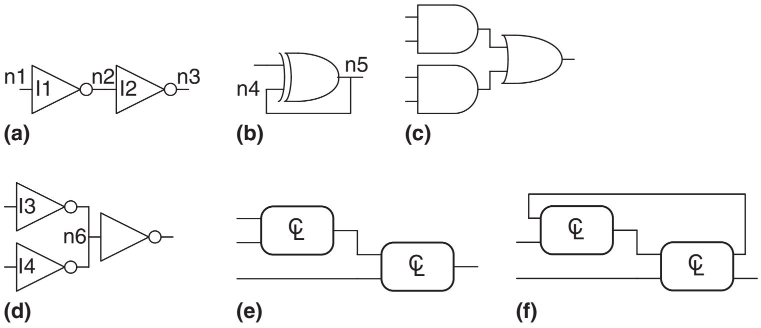


Figure 2.7 Example circuits

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations. We show how to implement these equations using logic gates and how to analyze the speed of these circuits.

2.2 BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic. This section defines some terminology commonly used in Boolean equations and then shows how to write a Boolean equation for any logic function, given its truth table.

2.2.1 Terminology

The *complement* of a variable A is its inverse $\overline{A}$. The variable or its complement is called a *literal*. For example, A, $\overline{A}$, B, and $\overline{B}$ are literals. We call A the *true form* of the variable and $\overline{A}$ the *complementary form*; "true form" does not mean that A is TRUE but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant*. $\overline{A}B$, $A\overline{B}\,\overline{C}$, and B are all implicants for a function of three variables. A *minterm* is a product involving all of the inputs to the function. $A\overline{B}\,\overline{C}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not because it does not involve C. Similarly, the OR of one or more literals is called a *sum*. A *maxterm* is a sum involving all of the inputs to the function. $A + \overline{B} + C$ is a maxterm for a function of the three variables A, B, and C.

The *order of operations* is important when interpreting Boolean equations. Does $Y = A + BC$ mean $Y = (A$ OR $B)$ AND C or $Y = A$ OR $(B$ AND C)? In Boolean equations, NOT has the highest *precedence*, followed by AND, then OR. Just as in ordinary equations, products are performed before sums. Therefore, the equation is read as $Y = A$ OR $(B$ AND C). Equation 2.1 gives another example of order of operations.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \tag{2.1}$$

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	0	$A\,B$	m_3

Figure 2.8 Truth table and minterms

2.2.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs. Each row in a truth table is associated with a minterm that is TRUE for that row. Figure 2.8 shows a truth table of two inputs, A and B. Each row shows its corresponding minterm. For example, the minterm for the first row is $\overline{A}\,\overline{B}$ because $\overline{A}\,\overline{B}$ is TRUE when $A = 0$, $B = 0$. The minterms are numbered starting with 0; the top row corresponds to minterm 0, m_0, the next row to minterm 1, m_1, and so on.

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. For example, in Figure 2.8, there is only one row (or minterm) for which the output Y is TRUE, shown circled in blue. Thus, $Y = \overline{A}B$. Figure 2.9 shows a truth table with more than one row in which the output is TRUE. Taking the sum of each of the circled minterms gives $Y = \overline{A}B + AB$.

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

Figure 2.9 Truth table with multiple TRUE minterms

This is called the *sum-of-products* (*SOP*) *canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms). Although there are many ways to write the same function, such as $Y = B\overline{A} + BA$, we will sort the minterms in the same order that they appear in the truth table so that we always write the same Boolean expression for the same truth table.

Canonical form is just a fancy word for standard form. You can use the term to impress your friends and scare your enemies.

The sum-of-products canonical form can also be written in *sigma notation* using the summation symbol, Σ. With this notation, the function from Figure 2.9 would be written as:

$$F(A,B) = \Sigma(m_1, m_3)$$

or

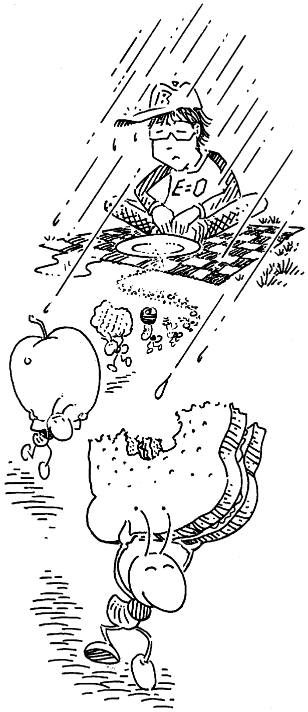
$$F(A,B) = \Sigma(1,3) \tag{2.2}$$

Example 2.2 SUM-OF-PRODUCTS (SOP) FORM

Ben Bitdiddle is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE *only* if Ben enjoys the picnic.

Solution First, define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers. Figure 2.10 shows the truth table for Ben's picnic experience.

Using sum-of-products form, we write the equation as: $E = \overline{A}\,\overline{R}$ or $E = \Sigma(0)$. We can build the equation using two inverters and a two-input AND gate, shown in Figure 2.11(a). You may recognize this truth table as the NOR function from Section 1.5.5: $E = A$ NOR $R = \overline{A + R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}\,\overline{R}$ and $\overline{A + R}$, are equivalent.



A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.10 Ben's truth table

The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Figure 2.12 shows a random three-input truth table. The sum-of-products form of the logic function is

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

or

$$Y = \Sigma(0,4,5) \tag{2.3}$$

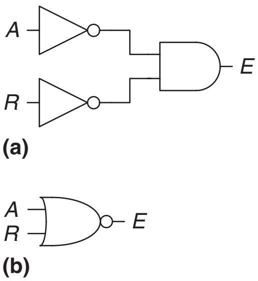


Figure 2.11 Ben's circuit

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.12 Random three-input truth table

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Figure 2.13 Truth table with multiple FALSE maxterms

Unfortunately, sum-of-products canonical form does not necessarily generate the simplest equation. In Section 2.3, we show how to write the same function using fewer terms.

2.2.3 Product-of-Sums Form

An alternative way of expressing Boolean functions is the *product-of-sums* (*POS*) *canonical form*. Each row of a truth table corresponds to a maxterm that is FALSE for that row. For example, the maxterm for the first row of a two-input truth table is $(A + B)$ because $(A + B)$ is FALSE when $A = 0$, $B = 0$. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE. The product-of-sums canonical form can also be written in *pi notation* using the product symbol, Π.

Example 2.3 PRODUCT-OF-SUMS (POS) FORM

Write an equation in product-of-sums form for the truth table in Figure 2.13.

Solution The truth table has two rows in which the output is FALSE. Hence, the function can be written in product-of-sums form as $Y = (A + B)(\overline{A} + B)$ or, using pi notation, $Y = \Pi(M_0, M_2)$ or $Y = \Pi(0, 2)$. The first maxterm, $(A + B)$, guarantees that $Y = 0$ for $A = 0$, $B = 0$, because any value AND 0 is 0. Likewise, the second maxterm, $(\overline{A} + B)$, guarantees that $Y = 0$ for $A = 1$, $B = 0$. Figure 2.13 is the same truth table as Figure 2.9, showing that the same function can be written in more than one way.

Similarly, a Boolean equation for Ben's picnic from Figure 2.10 can be written in product-of-sums form by circling the three rows of 0's to obtain $E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$ or $E = \Pi(1, 2, 3)$. This is uglier than the sum-of-products equation, $E = \overline{A}\overline{R}$, but the two equations are logically equivalent.

Sum-of-products produces a shorter equation when the output is TRUE on only a few rows of a truth table; product-of-sums is simpler when the output is FALSE on only a few rows of a truth table.

2.3 BOOLEAN ALGEBRA

In the previous section, we learned how to write a Boolean expression given a truth table. However, that expression does not necessarily lead to the simplest set of logic gates. Just as you use algebra to simplify mathematical equations, you can use *Boolean algebra* to simplify Boolean equations. The rules of Boolean algebra are much like those of ordinary algebra but are in some cases simpler because variables have only two possible values: 0 or 1.

Boolean algebra is based on a set of axioms that we assume are correct. Axioms are unprovable in the sense that a definition cannot be

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

proved. From these axioms, we prove all the theorems of Boolean algebra. These theorems have great practical significance because they teach us how to simplify logic to produce smaller and less costly circuits.

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators • (AND) and + (OR) are interchanged, the statement will still be correct. We use the prime symbol (′) to denote the *dual* of a statement.

2.3.1 Axioms

Table 2.1 states the axioms of Boolean algebra. These five axioms and their duals define Boolean variables and the meanings of NOT, AND, and OR. Axiom A1 states that a Boolean variable B is 0 if it is not 1. The axiom's dual, A1′, states that the variable is 1 if it is not 0. Together, A1 and A1′ tell us that we are working in a Boolean or binary field of 0's and 1's. Axioms A2 and A2′ define the NOT operation. Axioms A3 to A5 define AND; their duals, A3′ to A5′ define OR.

2.3.2 Theorems of One Variable

Theorems T1 to T5 in Table 2.2 describe how to simplify equations involving one variable.

The *identity* theorem, T1, states that for any Boolean variable B, B AND $1 = B$. Its dual states that B OR $0 = B$. In hardware, as shown in Figure 2.14, T1 means that if one input of a two-input AND gate is always 1, we can remove the AND gate and replace it with a wire connected to the variable input (B). Likewise, T1′ means that if one input of a two-input OR gate is always 0, we can replace the OR gate with a wire connected to B. In general, gates cost money, power, and delay, so replacing a gate with a wire is beneficial.

The *null element theorem*, T2, says that B AND 0 is always equal to 0. Therefore, 0 is called the *null* element for the AND operation, because it nullifies the effect of any other input. The dual states that B OR 1 is always equal to 1. Hence, 1 is the null element for the OR operation. In

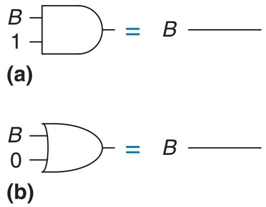


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1′

The null element theorem leads to some outlandish statements that are actually true! It is particularly dangerous when left in the hands of advertisers: YOU WILL GET A MILLION DOLLARS or we'll send you a toothbrush in the mail. (You'll most likely be receiving a toothbrush in the mail.)

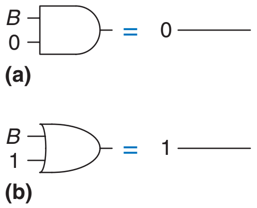


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

hardware, as shown in Figure 2.15, if one input of an AND gate is 0, we can replace the AND gate with a wire that is tied LOW (to 0). Likewise, if one input of an OR gate is 1, we can replace the OR gate with a wire that is tied HIGH (to 1).

Idempotency, T3, says that a variable AND itself is equal to just itself. Likewise, a variable OR itself is equal to itself. The theorem gets its name from the Latin roots: *idem* (same) and *potent* (power). The operations return the same thing you put into them. Figure 2.16 shows that idempotency again permits replacing a gate with a wire.

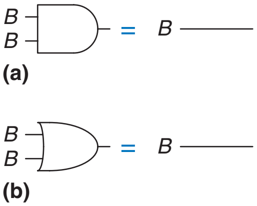


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

Involution, T4, is a fancy way of saying that complementing a variable twice results in the original variable. In digital electronics, two wrongs make a right. Two inverters in series logically cancel each other out and are logically equivalent to a wire, as shown in Figure 2.17. The dual of T4 is itself.

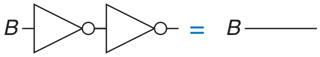


Figure 2.17 Involution theorem in hardware: T4

The *complement theorem*, T5 (Figure 2.18), states that a variable AND its complement is 0 (because one of them has to be 0). And, by duality, a variable OR its complement is 1 (because one of them has to be 1).

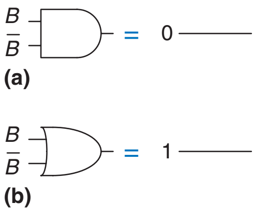


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

2.3.3 Theorems of Several Variables

Theorems T6 to T12 in Table 2.3 describe how to simplify equations involving more than one Boolean variable.

Commutativity and *associativity*, T6 and T7, work the same as in traditional algebra. By commutativity, the *order* of inputs for an AND or OR function does not affect the value of the output. By associativity, the specific groupings of inputs in AND or OR operations do not affect the value of the output.

The *distributivity theorem*, T8, is the same as in traditional algebra, but its dual, T8′, is not. By T8, AND distributes over OR, and by T8′, OR distributes over AND. In traditional algebra, multiplication distributes over addition but addition does not distribute over multiplication so that $(B + C) \times (B + D) \neq B + (C \times D)$.

The *covering*, *combining*, and *consensus* theorems, T9 to T11, permit us to eliminate redundant variables. With some thought, you should be able to convince yourself that these theorems are correct.

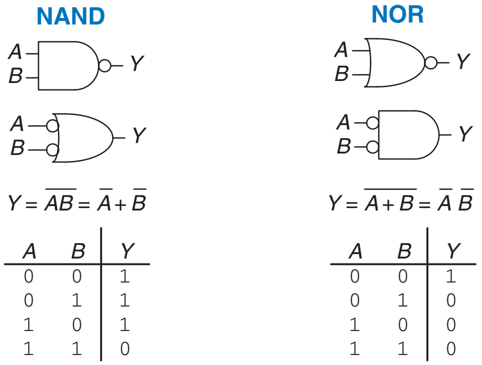
Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\overline{B} \bullet D)$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots)$	De Morgan's Theorem

De Morgan's Theorem, T12, is a particularly powerful tool in digital design. The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

According to De Morgan's theorem, a NAND gate is equivalent to an OR gate with inverted inputs. Similarly, a NOR gate is equivalent to an AND gate with inverted inputs. Figure 2.19 shows these *De Morgan equivalent gates* for NAND and NOR gates. The two symbols shown for each function are called *duals*. They are logically equivalent and can be used interchangeably.

The inversion circle is called a *bubble*. Intuitively, you can imagine that "pushing" a bubble through the gate causes it to come out at the other



$Y = \overline{AB} = \overline{A} + \overline{B}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$Y = \overline{A + B} = \overline{A}\,\overline{B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.19 De Morgan equivalent gates

Augustus De Morgan, died 1871
A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was *x* years of age in the year x^2."

side and flips the body of the gate from AND to OR or vice versa. For example, the NAND gate in Figure 2.19 consists of an AND body with a bubble on the output. Pushing the bubble to the left results in an OR body with bubbles on the inputs. The underlying rules for bubble pushing are

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on *all* gate inputs forward toward the output puts a bubble on the output.

Section 2.5.2 uses bubble pushing to help analyze circuits.

Notice that $\overline{AB}$ is *not* equal to $\overline{A}\,\overline{B}$.

Example 2.4 DERIVE THE PRODUCT-OF-SUMS FORM

Figure 2.20 shows the truth table for a Boolean function Y and its complement $\overline{Y}$. Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Figure 2.20 Truth table showing Y and $\overline{Y}$

Solution Figure 2.21 shows the minterms (circled) contained in $\overline{Y}$. The sum-of-products canonical form of $\overline{Y}$ is

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B \qquad (2.4)$$

A	B	Y	$\overline{Y}$	minterm
0	0	0	**1**	$\overline{A}\ \overline{B}$
0	1	0	**1**	$\overline{A}\ B$
1	0	1	**0**	$A\ \overline{B}$
1	1	1	**0**	$A\ B$

Figure 2.21 Truth table showing minterms for $\overline{Y}$

Taking the complement of both sides and applying De Morgan's Theorem twice, we get

$$\overline{\overline{Y}} = Y = \overline{\overline{A}\,\overline{B} + \overline{A}B} = (\overline{\overline{A}\,\overline{B}})(\overline{\overline{A}B}) = (A + B)(A + \overline{B}) \qquad (2.5)$$

2.3.4 The Truth Behind It All

The curious reader might wonder how to prove that a theorem is true. In Boolean algebra, proofs of theorems with a finite number of variables are easy: just show that the theorem holds for all possible values of these variables. This method is called *perfect induction* and can be done with a truth table.

Example 2.5 PROVING THE CONSENSUS THEOREM USING PERFECT INDUCTION

Prove the consensus theorem, T11, from Table 2.3.

Solution Check both sides of the equation for all eight combinations of B, C, and D. The truth table in Figure 2.22 illustrates these combinations. Because $BC + \overline{B}D + CD = BC + \overline{B}D$ for all cases, the theorem is proved.

B	C	D	$BC+\overline{B}D+CD$	$BC+\overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Figure 2.22 **Truth table proving T11**

2.3.5 Simplifying Equations

The theorems of Boolean algebra help us simplify Boolean equations. For example, consider the sum-of-products expression from the truth table of Figure 2.9: $Y = \overline{A}B + AB$. By the combining theorem (T10), the equation simplifies to $Y = B$. This may have been obvious looking at the truth table. In general, multiple steps may be necessary to simplify more complex equations.

The basic principle of simplifying sum-of-products equations is to combine terms using the relationship $PA + P\overline{A} = P$, where P may be any term. How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants in the equation to form a new implicant with fewer literals. The implicants in a minimal equation must all be prime implicants. Otherwise, they could be combined to reduce the number of literals.

Tables 2.2 and 2.3 list the Boolean axioms and theorems in their fundamental forms. However, they also apply to more complex terms. For example, we apply the combining theorem to the expression $(A+\overline{B}C)D + (A+\overline{B}C)\overline{D}$ to produce $(A+\overline{B}C)$.

Example 2.6 EQUATION MINIMIZATION

Minimize Equation 2.3: $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$.

Solution We start with the original equation and apply Boolean theorems step by step, as shown in Table 2.4.

Have we simplified the equation completely at this point? Let's take a closer look. From the original equation, the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $A\overline{B}\,\overline{C}$ differ only in the variable A. So we combined the minterms to form $\overline{B}\,\overline{C}$. However, if we look at the original equation, we note that the last two minterms $A\overline{B}\,\overline{C}$ and $A\overline{B}C$ also differ by a single literal (C and $\overline{C}$). Thus, using the same method, we could have combined these two minterms to form the minterm $A\overline{B}$. We say that implicants $\overline{B}\,\overline{C}$ and $A\overline{B}$ *share* the minterm $A\overline{B}\,\overline{C}$.

So, are we stuck with simplifying only one of the minterm pairs, or can we simplify both? Using the idempotency theorem, we can duplicate terms as many times as we want: $B = B + B + B + B \ldots$. Using this principle, we simplify the equation completely to its two prime implicants, $\overline{B}\,\overline{C} + A\overline{B}$, as shown in Table 2.5.

Table 2.4 Equation minimization

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}C$	T8: Distributivity
2	$\overline{B}\,\overline{C}(1) + A\overline{B}C$	T5: Complements
3	$\overline{B}\,\overline{C} + A\overline{B}C$	T1: Identity

Table 2.5 Improved equation minimization

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1) + A\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C} + A\overline{B}$	T1: Identity

Although it is a bit counterintuitive, *expanding* an implicant (e.g., turning AB into $ABC + AB\overline{C}$) is sometimes useful in minimizing equations. By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called *Karnaugh maps* that makes the process easier.

Why bother simplifying a Boolean equation if it remains logically equivalent? Simplifying reduces the number of gates used to physically implement the function, thus making it smaller, cheaper, and possibly faster. The next section describes how to implement Boolean equations with logic gates.

2.4 FROM LOGIC TO GATES

The labs that accompany this textbook (see Preface) show how to use *computer-aided design* (CAD) tools to design, simulate, and test digital circuits.

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them. For example, the schematic in Figure 2.23 shows a possible hardware implementation of our favorite logic function, Equation 2.3:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C \tag{2.3}$$

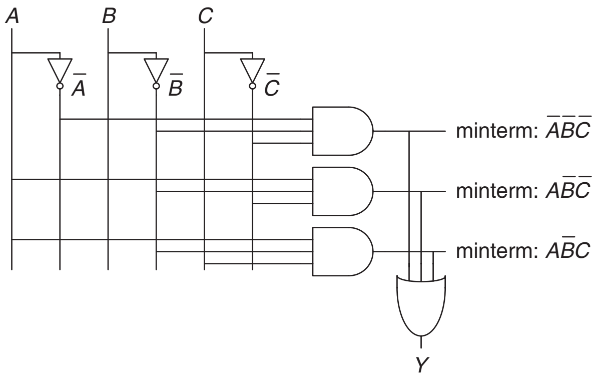

A
B
C
$\overline{A}$
$\overline{B}$
$\overline{C}$
minterm: $\overline{A}\,\overline{B}\,\overline{C}$
minterm: $A\overline{B}\,\overline{C}$
minterm: $A\overline{B}C$
Y

Figure 2.23 Schematic of $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking about what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

The last three guidelines are illustrated in Figure 2.24.

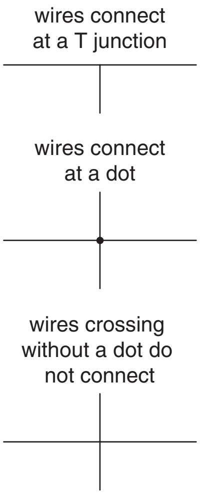

wires connect at a T junction
wires connect at a dot
wires crossing without a dot do not connect

Figure 2.24 Wire connections

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to Figure 2.23. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This style is called a *programmable logic array* (*PLA*) because the inverters, AND gates, and OR gates are arrayed in a systematic fashion. PLAs will be discussed further in Section 5.6.

Figure 2.25 shows an implementation of the simplified equation we found using Boolean algebra in Example 2.6. Notice that the simplified circuit has significantly less hardware than that of Figure 2.23. It may also be faster because it uses gates with fewer inputs.

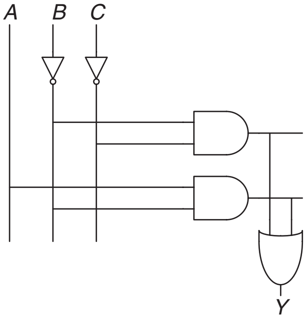


Figure 2.25 Schematic of $Y = \overline{B}\,\overline{C} + A\overline{B}$

We can reduce the number of gates even further (albeit by a single inverter) by taking advantage of inverting gates. Observe that $\overline{B}\,\overline{C}$ is an AND with inverted inputs. Figure 2.26 shows a schematic using this optimization to eliminate the inverter on C. Recall that by De Morgan's

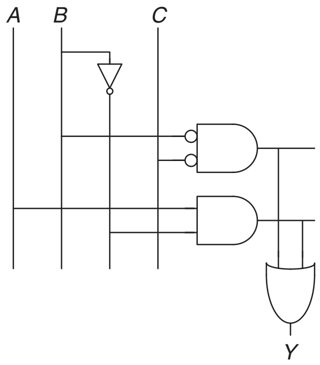


Figure 2.26 **Schematic using fewer gates**

theorem, the AND with inverted inputs is equivalent to a NOR gate. Depending on the implementation technology, it may be cheaper to use the fewest gates or to use certain types of gates in preference to others. For example, NANDs and NORs are preferred over ANDs and ORs in CMOS implementations.

Many circuits have multiple outputs, each of which computes a separate Boolean function of the inputs. We can write a separate truth table for each output, but it is often convenient to write all of the outputs on a single truth table and sketch one schematic with all of the outputs.

Example 2.7 MULTIPLE-OUTPUT CIRCUITS

The dean, the department chair, the teaching assistant, and the dorm social chair each use the auditorium from time to time. Unfortunately, they occasionally conflict, leading to disasters such as the one that occurred when the dean's fundraising meeting with crusty trustees happened at the same time as the dorm's BTB[1] party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, A_3, …, A_0, and four outputs, Y_3, …, Y_0. These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts her input when she requests the auditorium for the next day. The system asserts at most one output, granting the auditorium to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority.

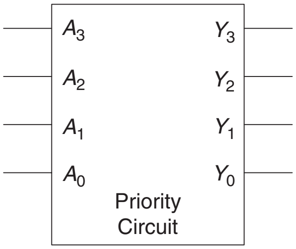
Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.

Solution This function is called a four-input *priority circuit*. Its symbol and truth table are shown in Figure 2.27.

We could write each output in sum-of-products form and reduce the equations using Boolean algebra. However, the simplified equations are clear by inspection from the functional description (and the truth table): Y_3 is TRUE whenever A_3 is asserted, so $Y_3 = A_3$. Y_2 is TRUE if A_2 is asserted and A_3 is not asserted, so $Y_2 = \overline{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher-priority inputs is asserted: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$ And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$ The schematic is shown in Figure 2.28. An experienced designer can often implement a logic circuit by inspection. Given a clear specification, simply turn the words into equations and the equations into gates.

Notice that if A_3 is asserted in the priority circuit, the outputs *don't care* what the other inputs are. We use the symbol X to describe inputs

[1] Black light, twinkies, and beer.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2.27 Priority circuit

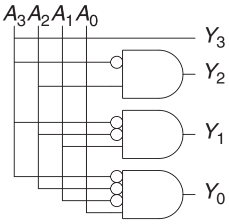


Figure 2.28 Priority circuit schematic

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)

that the output doesn't care about. Figure 2.29 shows that the four-input priority circuit truth table becomes much smaller with don't cares. From this truth table, we can easily read the Boolean equations in sum-of-products form by ignoring inputs with X's. Don't cares can also appear in truth table outputs, as we will see in Section 2.7.3.

X is an overloaded symbol that means "don't care" in truth tables and "contention" in logic simulation (see Section 2.6.1). Think about the context so you don't mix up the meanings. Some authors use D or ? instead for "don't care" to avoid this ambiguity.

2.5 MULTILEVEL COMBINATIONAL LOGIC

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts. Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

2.5.1 Hardware Reduction

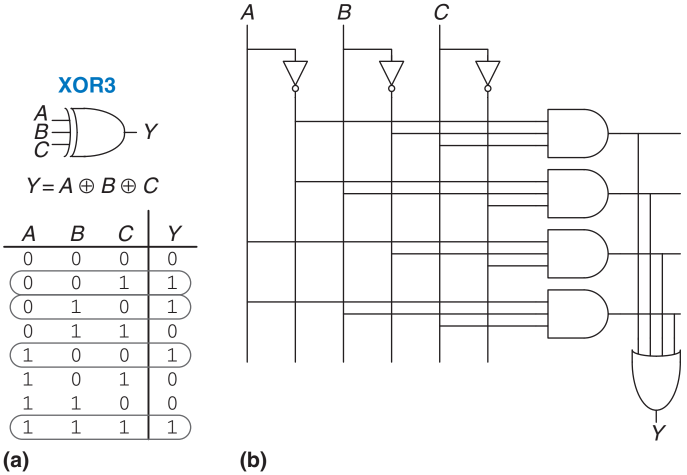
Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. For example, consider building a three-input XOR using the two-level techniques we have studied so far.

Recall that an *N*-input XOR produces a TRUE output when an odd number of inputs are TRUE. Figure 2.30 shows the truth table for a three-input XOR with the rows circled that produce TRUE outputs. From the truth table, we read off a Boolean equation in sum-of-products form in Equation 2.6. Unfortunately, there is no way to simplify this equation into fewer implicants.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{2.6}$$

On the other hand, $A \oplus B \oplus C = (A \oplus B) \oplus C$ (prove this to yourself by perfect induction if you are in doubt). Therefore, the three-input XOR can be built out of a cascade of two-input XORs, as shown in Figure 2.31.

Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two-level sum-of-products implementation. A much better option is to use a tree of two-input XOR gates, as shown in Figure 2.32.



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.30 Three-input XOR: (a) functional specification and (b) two-level logic implementation

Selecting the best multilevel implementation of a specific logic function is not a simple process. Moreover, "best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption. In Chapter 5, you will see that the "best" circuit in one technology is not necessarily the best in another. For example, we have been using ANDs and ORs, but in CMOS, NANDs and NORs are more efficient. With some experience, you will find that you can create a good multilevel design by inspection for most circuits. You will develop some of this experience as you study circuit examples through the rest of this book. As you are learning, explore various design options and think about the trade-offs. Computer-aided design (CAD) tools are also available to search a vast space of possible multilevel designs and seek the one that best fits your constraints given the available building blocks.

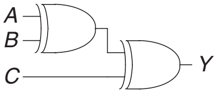


Figure 2.31 Three-input XOR using two-input XORs

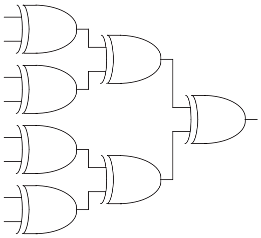


Figure 2.32 Eight-input XOR using seven two-input XORs

2.5.2 Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits prefer NANDs and NORs over ANDs and ORs. But reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Figure 2.33 shows a multilevel circuit whose function is not immediately clear by inspection. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined. Building on the principles from Section 2.3.3, the guidelines for bubble pushing are as follows:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (e.g., Y) instead of the complement of the output ($\overline{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

Figure 2.34 shows how to redraw Figure 2.33 according to the bubble pushing guidelines. Starting at the output Y, the NAND gate has a bubble on the output that we wish to eliminate. We push the output bubble back to form an OR with inverted inputs, shown in

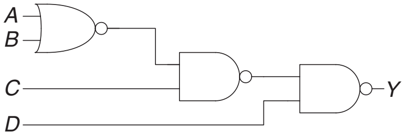


Figure 2.33 Multilevel circuit using NANDs and NORs

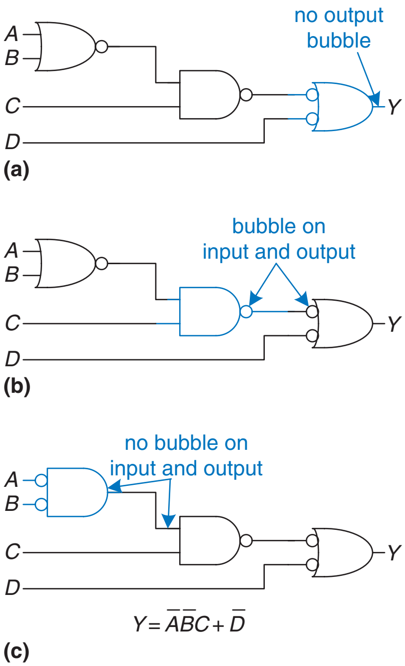


Figure 2.34 Bubble-pushed circuit

Figure 2.34(a). Working to the left, the rightmost gate has an input bubble that cancels with the output bubble of the middle NAND gate, so no change is necessary, as shown in Figure 2.34(b). The middle gate has no input bubble, so we transform the leftmost gate to have no output bubble, as shown in Figure 2.34(c). Now, all of the bubbles in the circuit cancel except at the inputs, so the function can be read by inspection in terms of ANDs and ORs of true or complementary inputs: $Y = \overline{A}\overline{B}C + \overline{D}$.

For emphasis of this last point, Figure 2.35 shows a circuit logically equivalent to the one in Figure 2.34. The functions of internal nodes are labeled in blue. Because bubbles in series cancel, we can ignore the bubbles on the output of the middle gate and on one input of the rightmost gate to produce the logically equivalent circuit of Figure 2.35.

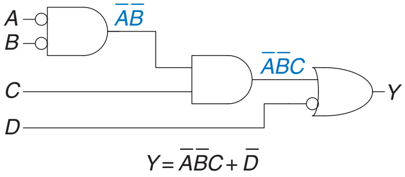

A
B
$\overline{A}\overline{B}$
$\overline{A}\overline{B}C$
C
D
Y
Y = $\overline{A}\overline{B}C + \overline{D}$

Figure 2.35 Logically equivalent bubble-pushed circuit

Example 2.8 BUBBLE PUSHING FOR CMOS LOGIC

Most designers think in terms of AND and OR gates, but suppose you would like to implement the circuit in Figure 2.36 in CMOS logic, which favors NAND and NOR gates. Use bubble pushing to convert the circuit to NANDs, NORs, and inverters.

Solution A brute force solution is to just replace each AND gate with a NAND and an inverter, and each OR gate with a NOR and an inverter, as shown in Figure 2.37. This requires eight gates. Notice that the inverter is drawn with the bubble on the front rather than back, to emphasize how the bubble can cancel with the preceding inverting gate.

For a better solution, observe that bubbles can be added to the output of a gate and the input of the next gate without changing the function, as shown in Figure 2.38(a). The final AND is converted to a NAND and an inverter, as shown in Figure 2.38(b). This solution requires only five gates.

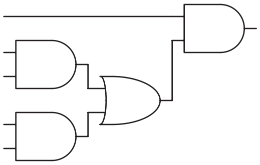


Figure 2.36 Circuit using ANDs and ORs

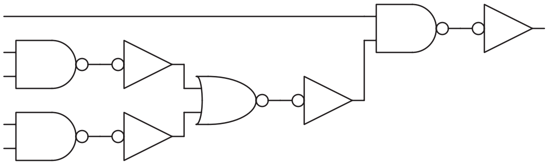


Figure 2.37 Poor circuit using NANDs and NORs

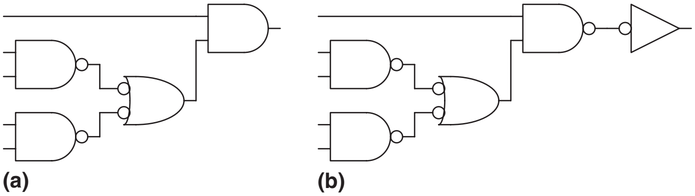


Figure 2.38 Better circuit using NANDs and NORs

2.6 X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

2.6.1 Illegal Value: X

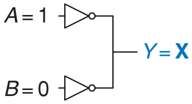


Figure 2.39 Circuit with contention

The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.39 shows a case where node *Y* is driven both HIGH and LOW. This situation, called *contention*, is considered to be an error and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone. Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.

X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume that it is an X to warn you of the problem.

As mentioned in Section 2.4, digital designers also use the symbol X to indicate "don't care" values in truth tables. Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant (can be either 0 or 1). When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

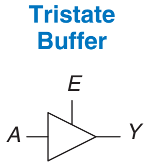
2.6.2 Floating Value: Z

The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high* Z. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system. A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.

One common way to produce a floating node is to forget to connect a voltage to a circuit input or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically, as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.

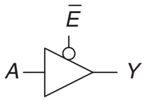
The *tristate buffer*, shown in Figure 2.40, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input *A*, output *Y*, and *enable E*. When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

The tristate buffer in Figure 2.40 has an *active high* enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure 2.41 shows a



E	*A*	*Y*
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\overline{E}$	*A*	*Y*
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

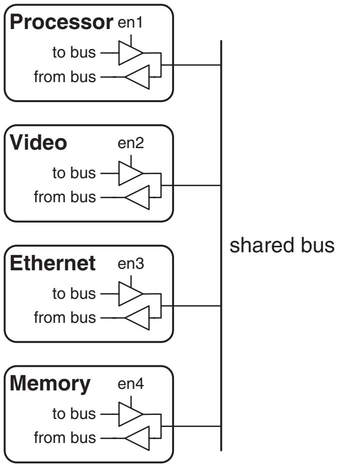


Figure 2.42 **Tristate bus connecting multiple chips**

tristate buffer with an *active low* enable. When the enable is LOW (0), the buffer is enabled. We show that the signal is active low by putting a bubble on its input wire. We often indicate an active low input by drawing a bar over its name, $\overline{E}$, or appending the letters "b" or "bar" after its name, *Eb* or *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in Figure 2.42. Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time. Such tristate busses were once common. However, in modern computers, higher speeds are possible with *point-to-point links*, in which chips are connected to each other directly rather than over a shared bus.

Maurice Karnaugh, 1924–
Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952.

Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

2.7 KARNAUGH MAPS

After working through several minimizations of Boolean equations using Boolean algebra, you will realize that, if you're not careful, you sometimes end up with a completely *different* equation instead of a simplified equation. *Karnaugh maps* (*K-maps*) are a graphical method for simplifying Boolean equations. They were invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs. K-maps work well for problems with up to four variables. More important, they give insight into manipulating Boolean equations.

Recall that logic minimization involves combining terms. Two terms containing an implicant P and the true and complementary forms of some variable A are combined to eliminate $A : PA + P\overline{A} = P$. Karnaugh maps make these combinable terms easy to see by putting them next to each other in a grid.

Figure 2.43 shows the truth table and K-map for a three-input function. The top row of the K-map gives the four possible values for the A and B inputs. The left column gives the two possible values for the C input. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row. For example, the top left square corresponds to the first row in the truth table and indicates that the output value $Y = 1$ when $ABC = 000$. Just like each row in a truth table, each square in a K-map represents a single minterm. For the purpose of explanation, Figure 2.43(c) shows the minterm corresponding to each square in the K-map.

Each square, or minterm, differs from an adjacent square by a change in a single variable. This means that adjacent squares share all the same literals except one, which appears in true form in one square and in complementary form in the other. For example, the squares representing the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$ are adjacent and differ only in the variable C. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10. This order is called a *Gray code*. It differs from ordinary binary order (00, 01, 10, 11) in that adjacent entries differ only in a single variable. For example, 01:11 changes only A from 0 to 1, while 01:10 would change A from 0 to 1 and B from 1 to 0. Hence, writing the combinations in binary order would not have produced our desired property of adjacent squares differing only in one variable.

The K-map also "wraps around." The squares on the far right are effectively adjacent to the squares on the far left in that they differ only in one variable, A. In other words, you could take the map and roll it into a cylinder, then join the ends of the cylinder to form a torus (i.e., a donut), and still guarantee that adjacent squares would differ only in one variable.

2.7.1 Circular Thinking

In the K-map in Figure 2.43, only two minterms are present in the equation, $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$, as indicated by the 1's in the left column. Reading the minterms from the K-map is exactly equivalent to reading equations in sum-of-products form directly from the truth table.

Gray codes were patented (U.S. Patent 2,632,058) by Frank Gray, a Bell Labs researcher, in 1953. They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit.

Gray codes generalize to any number of bits. For example, a 3-bit Gray code sequence is:

```
000, 001, 011, 010,
110, 111, 101, 100
```

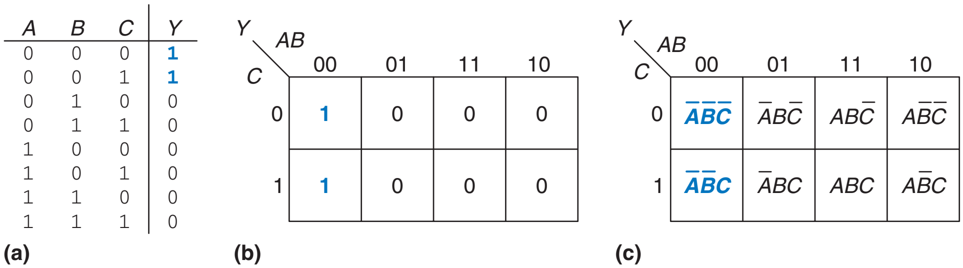
Lewis Carroll posed a related puzzle in *Vanity Fair* in 1879.

> "The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the puzzle consists of linking these together by interposing other words, each of which shall differ from the next word in one letter only. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word."

For example, SHIP to DOCK:

```
SHIP, SLIP, SLOP,
SLOT, SOOT, LOOT,
LOOK, LOCK, DOCK.
```

Can you find a shorter sequence?



(a)

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

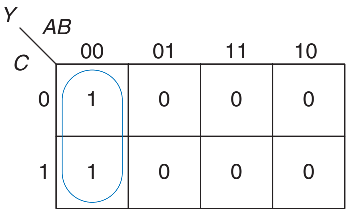
(b)

Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

(c)

Y: C \ AB	00	01	11	10
0	$\overline{A}\,\overline{B}\,\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\,\overline{C}$
1	$\overline{A}\,\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms



Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

Figure 2.44 K-map minimization

As before, we can use Boolean algebra to minimize equations in sum-of-products form.

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C = \overline{A}\,\overline{B}(\overline{C} + C) = \overline{A}\,\overline{B} \qquad (2.7)$$

K-maps help us do this simplification graphically by *circling* 1's in adjacent squares, as shown in Figure 2.44. For each circle, we write the corresponding implicant. Remember from Section 2.2 that an implicant is the product of one or more literals. Variables whose true *and* complementary forms are both in the circle are excluded from the implicant. In this case, the variable C has both its true form (1) and its complementary form (0) in the circle, so we do not include it in the implicant. In other words, Y is TRUE when $A = B = 0$, independent of C. So, the implicant is $\overline{A}\,\overline{B}$. The K-map gives the same answer we reached using Boolean algebra.

2.7.2 Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as large as possible. Then, read off the implicants that were circled.

More formally, recall that a Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants. Each circle on the K-map represents an implicant. The largest possible circles are prime implicants.

For example, in the K-map of Figure 2.44, $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$ are implicants, but *not* prime implicants. Only $\overline{A}\overline{B}$ is a prime implicant in that K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

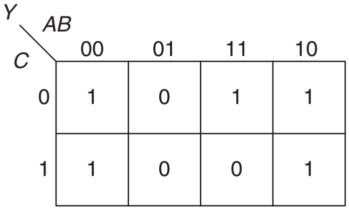
Example 2.9 MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP

Suppose we have the function $Y = F(A, B, \text{C})$ with the K-map shown in Figure 2.45. Minimize the equation using the K-map.

Solution Circle the 1's in the K-map using as few circles as possible, as shown in Figure 2.46. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two (2 × 1 and 2 × 2). We form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.

For example, in the 2 × 1 circle, the true and complementary forms of B are included in the circle, so we *do not* include B in the prime implicant. However, only the true form of $A\,(A)$ and complementary form of $C\,(\overline{C})$ are in this circle, so we include these variables in the prime implicant $A\overline{C}$. Similarly, the 2 × 2 circle covers all squares where $B = 0$, so the prime implicant is $\overline{B}$.

Notice how the top-right square (minterm) is covered twice to make the prime implicant circles as large as possible. As we saw with Boolean algebra techniques, this is equivalent to sharing a minterm to reduce the size of the implicant. Also notice how the circle covering four squares wraps around the sides of the K-map.



Y AB C	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Figure 2.45 K-map for Example 2.9

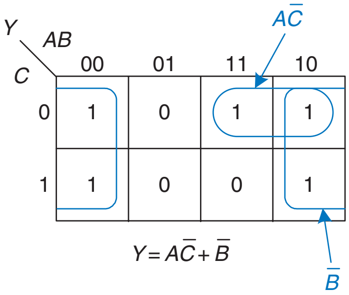


Figure 2.46 Solution for Example 2.9

Example 2.10 SEVEN-SEGMENT DISPLAY DECODER

A *seven-segment display decoder* takes a 4-bit data input $D_{3:0}$ and produces seven outputs to control light-emitting diodes to display a digit from 0 to 9. The seven outputs are often called segments *a* through *g*, or S_a–S_g, as defined in Figure 2.47. The digits are shown in Figure 2.48. Write a truth table for the outputs, and use K-maps to find Boolean equations for outputs S_a and S_b. Assume that illegal input values (10–15) produce a blank readout.

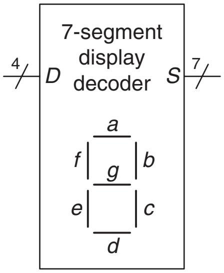


Figure 2.47 Seven-segment display decoder icon

Solution The truth table is given in Table 2.6. For example, an input of 0000 should turn on all segments except S_g.

Each of the seven outputs is an independent function of four variables. The K-maps for outputs S_a and S_b are shown in Figure 2.49. Remember that adjacent squares may differ in only a single variable, so we label the rows and columns in Gray code order: 00, 01, 11, 10. Be careful to also remember this ordering when *entering* the output values into the squares.

Next, circle the prime implicants. Use the fewest number of circles necessary to cover all the 1's. A circle can wrap around the edges (vertical *and* horizontal), and a 1 may be circled more than once. Figure 2.50 shows the prime implicants and the simplified Boolean equations.

Note that the minimal set of prime implicants is not unique. For example, the 0000 entry in the S_a K-map was circled, along with the 1000 entry to produce the $\overline{D}_2\overline{D}_1\overline{D}_0$ minterm. The circle could have included the 0010 entry instead, producing a $\overline{D}_3\overline{D}_2\overline{D}_0$ minterm, as shown with dashed lines in Figure 2.51.

Figure 2.52 (see page 80) illustrates a common error in which a nonprime implicant was chosen to cover the 1 in the upper left corner. This minterm, $\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$, gives a sum-of-products equation that is *not* minimal. The minterm could have been combined with either of the adjacent ones to form a larger circle, as was done in the previous two figures.

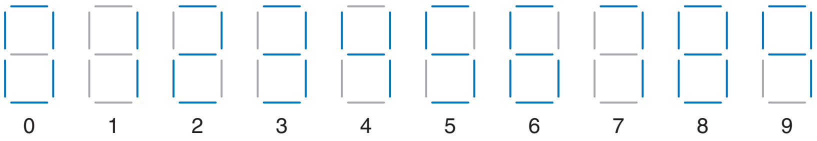
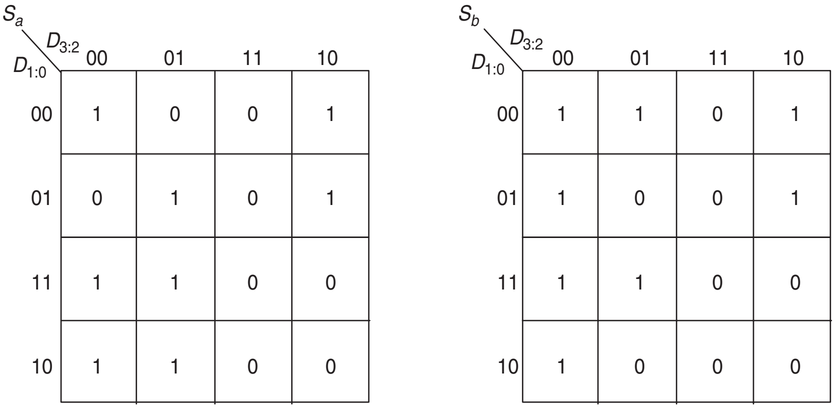


Figure 2.48 Seven-segment display digits

Table 2.6 Seven-segment display decoder truth table

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0



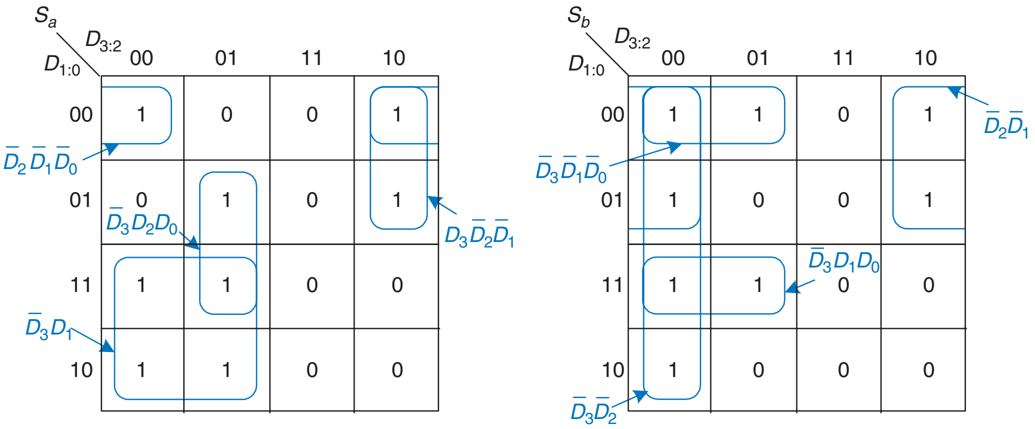
S_a

$D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

S_b

$D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

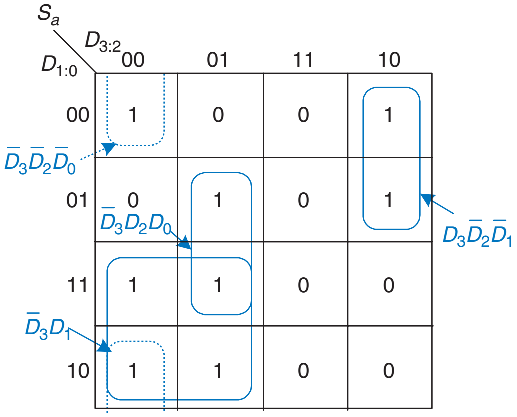
Figure 2.49 Karnaugh maps for S_a and S_b



$S_a = \bar{D}_3D_1 + \bar{D}_3D_2D_0 + D_3\bar{D}_2\bar{D}_1 + \bar{D}_2\bar{D}_1\bar{D}_0$

$S_b = \bar{D}_3\bar{D}_2 + \bar{D}_2\bar{D}_1 + \bar{D}_3D_1D_0 + \bar{D}_3\bar{D}_1\bar{D}_0$

Figure 2.50 **K-map solution for Example 2.10**



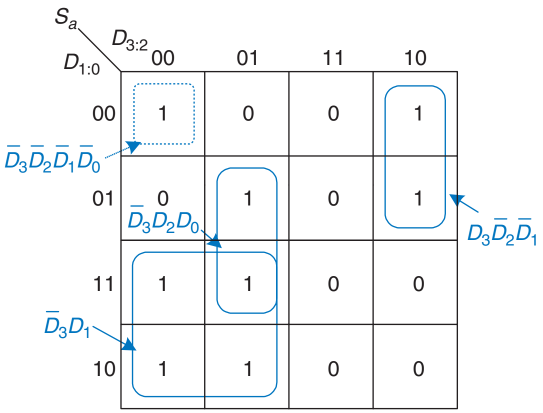
$S_a = \bar{D}_3D_1 + \bar{D}_3D_2D_0 + D_3\bar{D}_2\bar{D}_1 + \bar{D}_3\bar{D}_2\bar{D}_0$

Figure 2.51 **Alternative K-map for S_a showing different set of prime implicants**

2.7.3 Don't Cares

Recall that "don't care" entries for truth table inputs were introduced in Section 2.4 to reduce the number of rows in the table when some variables do not affect the output. They are indicated by the symbol X, which means that the entry can be either 0 or 1.

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never


S_a
$D_{3:2}$
$D_{1:0}$
$\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$
$\overline{D}_3D_2D_0$
$D_3\overline{D}_2\overline{D}_1$
$\overline{D}_3D_1$

$D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

$$S_a = \overline{D}_3D_1 + \overline{D}_3D_2D_0 + D_3\overline{D}_2\overline{D}_1 + \overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$$

Figure 2.52 Alternative K-map for *S_a* showing incorrect nonprime implicant

happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.

In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

Example 2.11 SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES

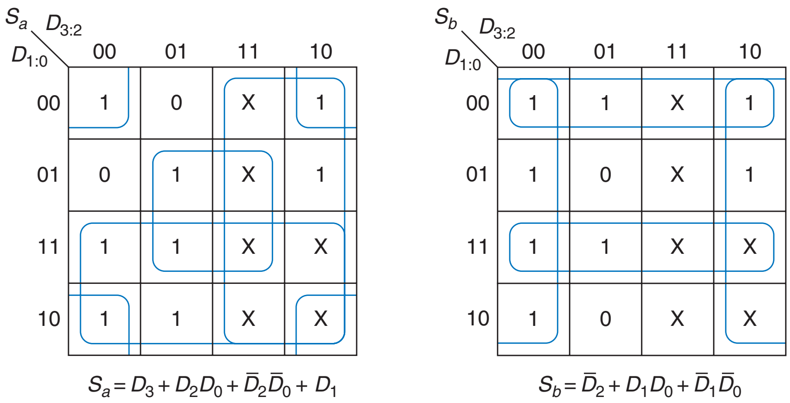
Repeat Example 2.10 if we don't care about the output values for illegal input values of 10 to 15.

Solution The K-map is shown in Figure 2.53 with X entries representing don't care. Because don't cares can be 0 or 1, we circle a don't care if it allows us to cover the 1's with fewer or bigger circles. Circled don't cares are treated as 1's, whereas uncircled don't cares are 0's. Observe how a 2 × 2 square wrapping around all four corners is circled for segment S_a. Use of don't cares simplifies the logic substantially.

2.7.4 The Big Picture

Boolean algebra and Karnaugh maps are two methods of logic simplification. Ultimately, the goal is to find a low-cost method of implementing a particular logic function.

In modern engineering practice, computer programs called *logic synthesizers* produce simplified circuits from a description of the logic function, as we will see in Chapter 4. For large problems, logic synthesizers are much more efficient than humans. For small problems, a human with a bit of experience can find a good solution by inspection. Neither of the



S_a ($D_{1:0}$ \ $D_{3:2}$)	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X

$S_a = D_3 + D_2D_0 + \overline{D}_2\overline{D}_0 + D_1$

S_b ($D_{1:0}$ \ $D_{3:2}$)	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

$S_b = \overline{D}_2 + D_1D_0 + \overline{D}_1\overline{D}_0$

Figure 2.53 K-map solution with don't cares

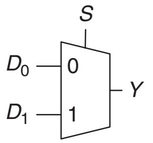
authors has ever used a Karnaugh map in real life to solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs, based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 2:1 multiplexer symbol and truth table

2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs D_0 and D_1, a select input S, and one output Y. The multiplexer chooses between the two data inputs, based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$. S is also called a *control signal* because it controls what the multiplexer does.

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived

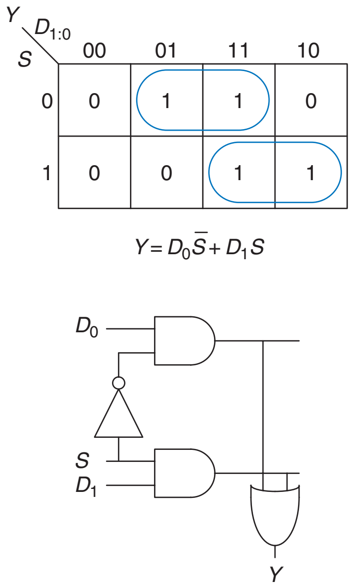


Figure 2.55 2:1 multiplexer implementation using two-level logic

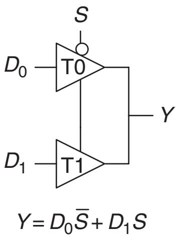


Figure 2.56 Multiplexer using tristate buffers

using a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

Alternatively, multiplexers can be built from tristate buffers as shown in Figure 2.56. The tristate enables are arranged such that exactly one tristate buffer is active at all times. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y. When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y.

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N:1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

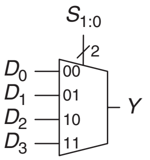


Figure 2.57 4:1 multiplexer

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

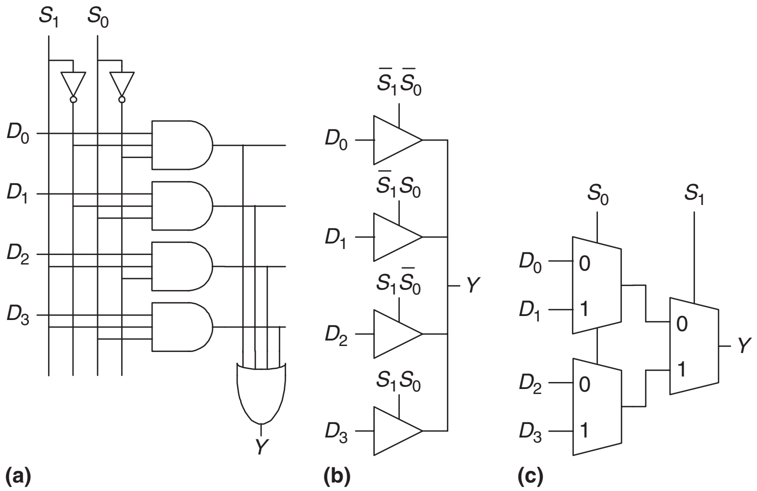
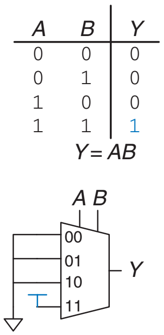


Figure 2.58 4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

AND gate. The inputs, *A* and *B*, serve as select lines. The multiplexer data inputs are connected to 0 or 1, according to the corresponding row of the truth table. In general, a 2^N-input multiplexer can be programmed to perform any *N*-input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1}-input multiplexer to perform any *N*-input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table and then combine pairs of rows to eliminate the rightmost input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of *B*. When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.



A	*B*	*Y*
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

Figure 2.59 4:1 multiplexer implementation of two-input AND function

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table, where each row in the truth table corresponds to a multiplexer input.

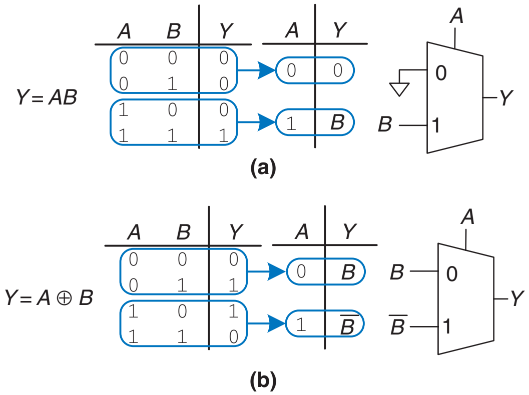


Figure 2.60 Multiplexer logic using variable inputs

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$

(a)

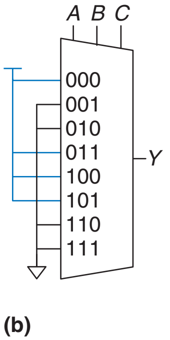


Figure 2.61 Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation

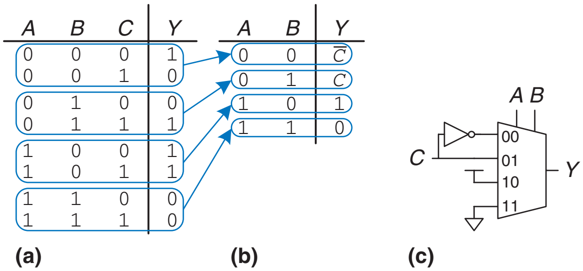
Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidently powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution Alyssa reduces her truth table to four rows by letting the output depend on C. (She could also have chosen to rearrange the columns of the truth table to let the output depend on A or B.) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.



A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

A	B	Y
0	0	$\overline{C}$
0	1	C
1	0	1
1	1	0

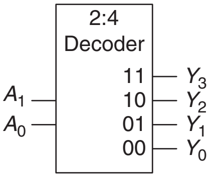
(b)

Figure 2.62 **Alyssa's new circuit**

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND and NOT gates.

Solution Figure 2.64 shows an implementation for the 2:4 decoder, using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an N:2^N decoder can be constructed from 2^N N-input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A_1}\,\overline{A_0}$. This fact will be handy when using decoders with other digital building blocks.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 **2:4 decoder**

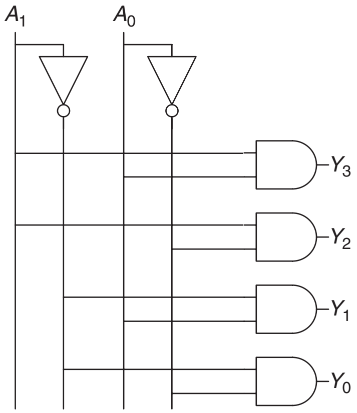


Figure 2.64 **2:4 decoder implementation**

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all of the minterms in the function. In Figure 2.65, $Y = \overline{A}\,\overline{B} + AB = \overline{A \oplus B}$.

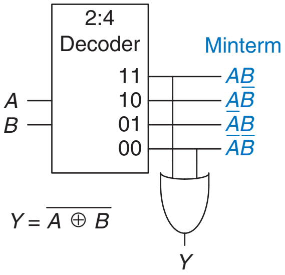


Figure 2.65 **Logic function using decoder**

When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. An N-input function with M 1's in the truth table can be built with an $N{:}2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table. This concept will be applied to the building of read-only memories (ROMs) in Section 5.5.6.

2.9 TIMING

In previous sections, we have been concerned primarily with whether the circuit works—ideally, using the fewest gates. However, as any seasoned circuit designer will attest, one of the most challenging issues in circuit design is *timing*: making a circuit run fast.

An output takes time to change in response to an input change. Figure 2.66 shows the *delay* between an input change and the subsequent output change for a buffer. The figure is called a *timing diagram*; it portrays the *transient response* of the buffer circuit when an input changes. The transition from LOW to HIGH is called the *rising edge*. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the *falling edge*. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A. We measure delay from the *50% point* of the input signal, A, to the 50% point of the output signal, Y. The 50% point is the point at which the signal is halfway (50%) between its LOW and HIGH values as it transitions.

When designers speak of calculating the *delay* of a circuit, they generally are referring to the worst-case value (the propagation delay), unless it is clear otherwise from the context.

2.9.1 Propagation and Contamination Delay

Combinational logic is characterized by its *propagation delay* and *contamination delay*. The propagation delay t_{pd} is the maximum time from when any input changes until the output or outputs reach their final value. The contamination delay t_{cd} is the minimum time from when any input changes until any output starts to change its value.

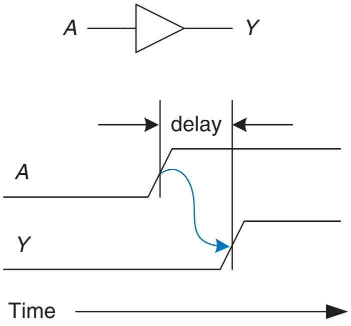


Figure 2.66 Circuit delay

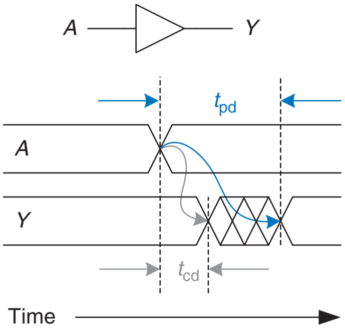


Figure 2.67 Propagation and contamination delay

Figure 2.67 illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps = 10^{-12} seconds) to nanoseconds (1 ns = 10^{-9} seconds). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Calculating t_{pd} and t_{cd} requires delving into the lower levels of abstraction beyond the scope of this book. However, manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.68 shows a four-input logic circuit. The *critical path*, shown in blue, is the path from input A or B to output Y. It is the longest—and, therefore, the slowest—path because the input travels

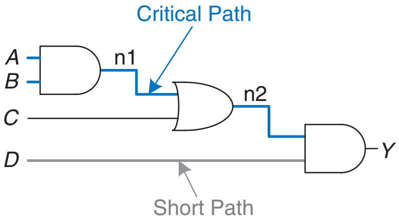


Figure 2.68 Short path and critical path

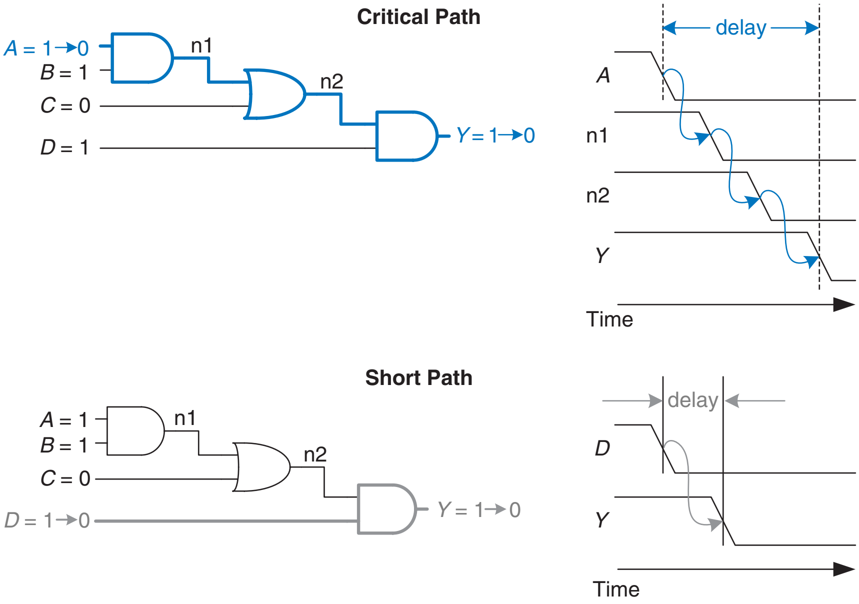


Figure 2.69 **Critical and short path waveforms**

through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The *short path* through the circuit, shown in gray, is from input D to output Y. This is the shortest—and, therefore, the fastest—path through the circuit because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure 2.69 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\text{AND}} + t_{pd\_\text{OR}} \tag{2.8}$$

$$t_{cd} = t_{cd\_\text{AND}} \tag{2.9}$$

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in wires is covered in Appendix A.

Example 2.15 FINDING DELAYS

Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure 2.70. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.

Solution Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue in Figure 2.71, is from input A or B through three gates to the output Y. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray in Figure 2.72, is from input C, D, or E through two gates to the output Y. There are only two gates in the shortest path, so t_{cd} is 120 ps.

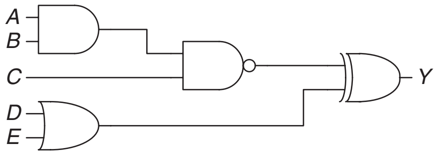


Figure 2.70 Ben's circuit

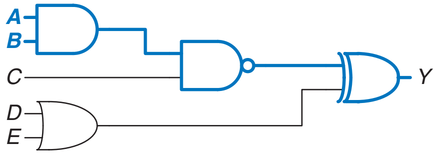


Figure 2.71 Ben's critical path

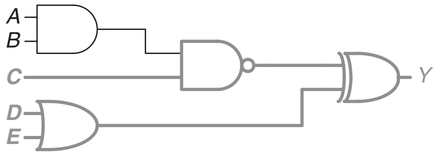


Figure 2.72 Ben's shortest path

Example 2.16 MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL

Compare the worst-case timing of the three four-input multiplexer designs shown in Figure 2.58 on page 83. Table 2.7 lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Solution One of the critical paths for each of the three design options is highlighted in blue in Figures 2.73 and 2.74. t_{pd_sy} indicates the propagation delay from input S to output Y; t_{pd_dy} indicates the propagation delay from input D to output Y; t_{pd} for the circuit is the worst of the two: $\max(t_{pd\_sy}, t_{pd\_dy})$.

For both the two-level logic and tristate implementations in Figure 2.73, the critical path is from one of the control signals S to the output Y: $t_{pd} = t_{pd\_sy}$. These circuits are *control critical*, because the critical path is from the control signals to the output. Any additional delay in the control signals will add directly to the worst-case delay. The delay from D to Y in Figure 2.73(b) is only 50 ps, compared with the delay from S to Y of 125 ps.

Figure 2.74 shows the hierarchical implementation of the 4:1 multiplexer using two stages of 2:1 multiplexers. The critical path is from any of the D inputs to the output. This circuit is *data critical*, because the critical path is from the data input to the output: $t_{pd} = t_{pd\_dy}$.

If data inputs arrive well before the control inputs, we would prefer the design with the shortest control-to-output delay (the hierarchical design in Figure 2.74). Similarly, if the control inputs arrive well before the data inputs, we would prefer the design with the shortest data-to-output delay (the tristate design in Figure 2.73(b)).

The best choice depends not only on the critical path through the circuit and the input arrival times but also on the power, cost, and availability of parts.

Table 2.7 **Timing specifications for multiplexer circuit elements**

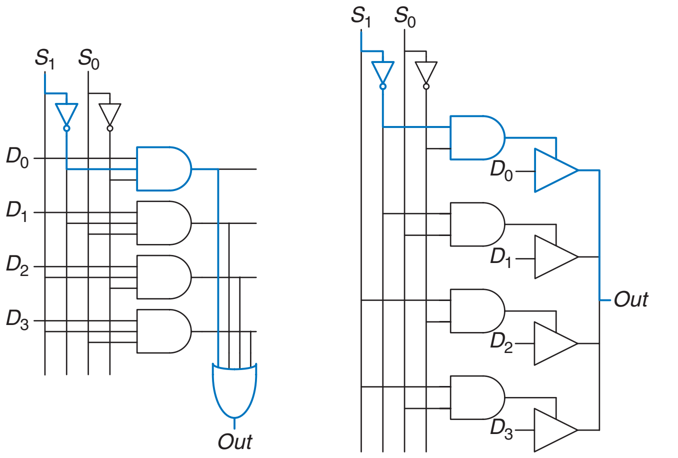
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

2.9.2 Glitches

So far, we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called *glitches* or *hazards*. Although glitches usually don't cause problems, it is important to realize that they exist and recognize them when looking at timing diagrams. Figure 2.75 shows a circuit with a glitch and the Karnaugh map of the circuit.

The Boolean equation is correctly minimized, but let's look at what happens when $A = 0$, $C = 1$, and B transitions from 1 to 0. Figure 2.76 (see page 92) illustrates this scenario. The short path (shown in gray) goes through two gates, the AND and OR gates. The critical path (shown in blue) goes through an inverter and two gates, the AND and OR gates.

Hazards have another meaning related to microarchitecture in Chapter 7, so we will stick with the term *glitches* for multiple output transitions to avoid confusion.



(a)

$$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4} = 30\text{ ps} + 80\text{ ps} + 90\text{ ps} = \mathbf{200\ ps}$$

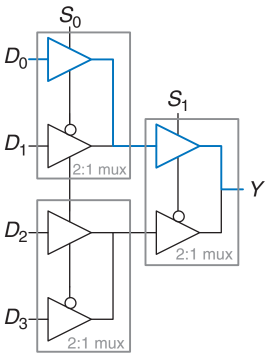
$$t_{pd\_dy} = t_{pd\_AND3} + t_{pd\_OR4} = \mathbf{170\ ps}$$

(b)

$$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_sy} = 30\text{ ps} + 60\text{ ps} + 35\text{ ps} = \mathbf{125\ ps}$$

$$t_{pd\_dy} = t_{pd\_TRI\_ay} = \mathbf{50\ ps}$$

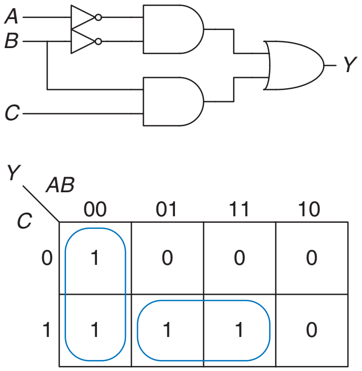
Figure 2.73 4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate



$$t_{pd\_s0y} = t_{pd\_TRI\_sy} + t_{pd\_TRI\_ay} = \mathbf{85\ ps}$$

$$t_{pd\_dy} = 2\ t_{pd\_TRI\_ay} = \mathbf{100\ ps}$$

Figure 2.74 4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers



C \ AB	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$$Y = \overline{A}\,\overline{B} + BC$$

Figure 2.75 Circuit with a glitch

As B transitions from 1 to 0, n2 (on the short path) falls before n1 (on the critical path) can rise. Until n1 rises, the two inputs to the OR gate are 0, and the output Y drops to 0. When n1 eventually rises, Y returns to 1. As shown in the timing diagram of Figure 2.76, Y starts at 1 and ends at 1 but momentarily glitches to 0.

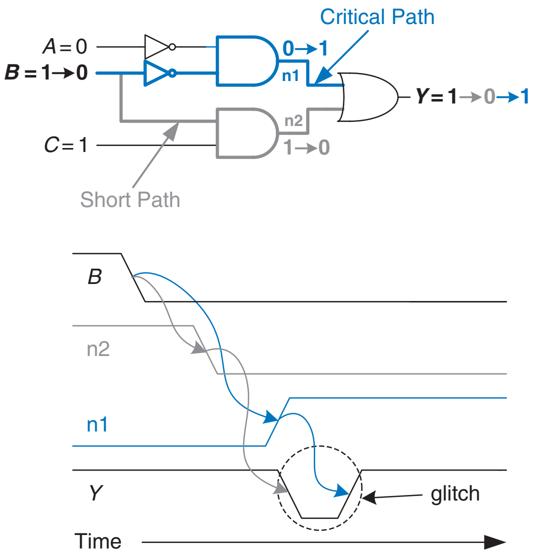


Figure 2.76 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are not a problem, because the output eventually settles to the right answer.

If we choose to, we can avoid this glitch by adding another gate to the implementation. This is easiest to understand in terms of the K-map. Figure 2.77 shows how an input transition on B from ABC = 011 to ABC = 001 moves from one prime implicant circle to another. The transition across the boundary of two prime implicants in the K-map indicates a possible glitch.

As we saw from the timing diagram in Figure 2.76, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.78. You might recognize this as the consensus theorem, where the added term, $\overline{A}C$, is the consensus or redundant term.

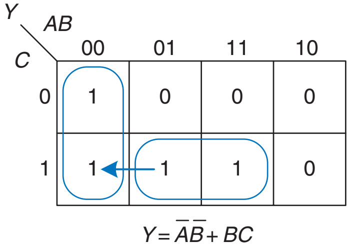
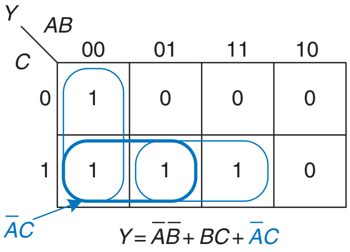


Figure 2.77 Input change crosses implicant boundary



C \ AB	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$Y = \overline{A}\,\overline{B} + BC + \overline{A}C$

Figure 2.78 **K-map without glitch**

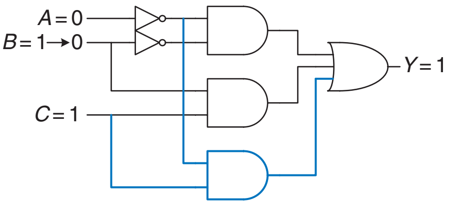


Figure 2.79 **Circuit without glitch**

Figure 2.79 shows the glitch-proof circuit. The added AND gate is highlighted in blue. Now, a transition on *B* when *A* = 0 and *C* = 1 does not cause a glitch on the output because the blue AND gate outputs 1 throughout the transition.

In general, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding redundant implicants to the K-map to cover these boundaries. This, of course, comes at the cost of extra hardware.

However, simultaneous transitions on multiple inputs can also cause glitches. These glitches cannot be fixed by adding hardware. Because the vast majority of interesting systems have simultaneous (or near-simultaneous) transitions on multiple inputs, glitches are a fact of life in most circuits. Although we have shown how to eliminate one kind of glitch, the point of discussing glitches is not to eliminate them but to be aware that they exist. This is especially important when looking at timing diagrams on a simulator or oscilloscope.

2.10 SUMMARY

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module. This chapter has focused on combinational circuits, whose outputs depend only on the current values of the inputs.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products (SOP) or product-of-sums (POS) form. In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equations can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$. Karnaugh maps are a visual tool for minimizing functions of up to four variables. With practice, designers can usually simplify functions of a few variables by inspection. Computer-aided design tools are used for more complicated functions; such methods and tools are discussed in Chapter 4.

Logic gates are connected to create combinational circuits that perform the desired function. Any function in sum-of-products form can be built using two-level logic: NOT gates form the complements of the inputs, AND gates form the products, and OR gates form the sum. Depending on the function and the building blocks available, multilevel logic implementations with various types of gates may be more efficient. For example, CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates. When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits, such as multiplexers, decoders, and priority circuits. A multiplexer chooses one of the data inputs based on the select input. A decoder sets one of the outputs HIGH according to the inputs. A priority circuit produces an output indicating the highest priority input. These circuits are all examples of combinational building blocks. Chapter 5 will introduce more building blocks, including other arithmetic circuits. These building blocks will be used extensively to build a microprocessor in Chapter 7.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit. These indicate the longest and shortest times between an input change and the consequent output change. Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path. There are many different ways to implement complicated combinational circuits; these ways offer trade-offs between speed and cost.

The next chapter will move to sequential circuits, whose outputs depend on current as well as previous values of the inputs. In other words, sequential circuits have *memory* of the past.

Exercises

Exercise 2.1 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.80.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.80 Truth tables for Exercises 2.1, 2.3, and 2.41

Exercise 2.2 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.81.

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 2.81 Truth tables for Exercises 2.2 and 2.4

Exercise 2.3 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.80.

Exercise 2.4 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.81.

Exercise 2.5 Minimize each of the Boolean equations from Exercise 2.1.

Exercise 2.6 Minimize each of the Boolean equations from Exercise 2.2.

Exercise 2.7 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.5. *Reasonably simple* means that you are not wasteful of gates, but you don't waste vast amounts of time checking every possible implementation of the circuit either.

Exercise 2.8 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.6.

Exercise 2.9 Repeat Exercise 2.7 using only NOT gates and AND and OR gates.

Exercise 2.10 Repeat Exercise 2.8 using only NOT gates and AND and OR gates.

Exercise 2.11 Repeat Exercise 2.7 using only NOT gates and NAND and NOR gates.

Exercise 2.12 Repeat Exercise 2.8 using only NOT gates and NAND and NOR gates.

Exercise 2.13 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + (\overline{A + \overline{C}})$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Exercise 2.14 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = \overline{A}BC + \overline{A}B\overline{C}$

(b) $Y = \overline{ABC} + A\overline{B}$

(c) $Y = ABC\overline{D} + A\overline{BCD} + (\overline{A + B + C + D})$

Exercise 2.15 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.13.

Exercise 2.16 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.14.

Exercise 2.17 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = BC + \overline{A}\overline{B}\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\overline{\overline{B}}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\overline{C}D$
$+ \overline{B}\overline{C}E + \overline{B}\overline{D}\overline{E} + \overline{C}\overline{D}\overline{E}$

Exercise 2.18 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$

(b) $Y = (\overline{A + B + C})D + AD + B$

(c) $Y = ABCD + \overline{A}B\overline{C}D + (\overline{\overline{B} + D})E$

Exercise 2.19 Give an example of a truth table requiring between 3 billion and 5 billion rows that can be constructed using fewer than 40 (but at least 1) two-input gates.

Exercise 2.20 Give an example of a circuit with a cyclic path that is nevertheless combinational.

Exercise 2.21 Alyssa P. Hacker says that any Boolean function can be written in minimal sum-of-products form as the sum of all of the prime implicants of the function. Ben Bitdiddle says that there are some functions whose minimal equation does not involve all of the prime implicants. Explain why Alyssa is right or provide a counterexample demonstrating Ben's point.

Exercise 2.22 Prove that the following theorems are true using perfect induction. You need not prove their duals.

(a) The idempotency theorem (T3)

(b) The distributivity theorem (T8)

(c) The combining theorem (T10)

Exercise 2.23 Prove De Morgan's Theorem (T12) for three variables, *A*, *B*, and *C*, using perfect induction.

Exercise 2.24 Write Boolean equations for the circuit in Figure 2.82. You need not minimize the equations.

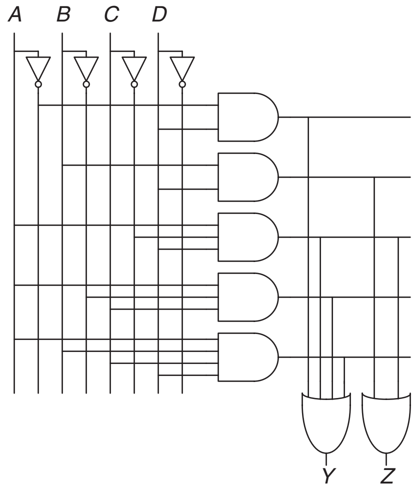


Figure 2.82 Circuit schematic for Exercise 2.24

Exercise 2.25 Minimize the Boolean equations from Exercise 2.24 and sketch an improved circuit with the same function.

Exercise 2.26 Using De Morgan equivalent gates and bubble pushing methods, redraw the circuit in Figure 2.83 so that you can find the Boolean equation by inspection. Write the Boolean equation.

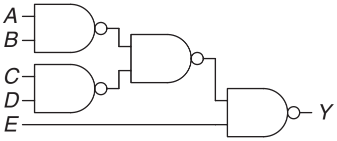


Figure 2.83 Circuit schematic for Exercises 2.26 and 2.43

Exercise 2.27 Repeat Exercise 2.26 for the circuit in Figure 2.84.

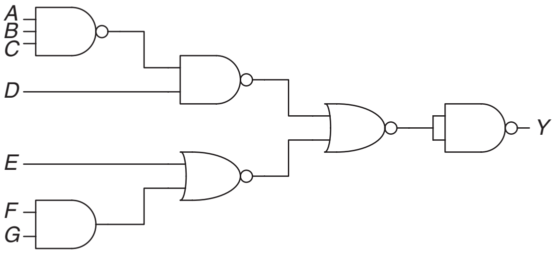


Figure 2.84 **Circuit schematic for Exercises 2.27 and 2.44**

Exercise 2.28 Find a minimal Boolean equation for the function in Figure 2.85. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.85 **Truth table for Exercise 2.28**

Exercise 2.29 Sketch a circuit for the function from Exercise 2.28.

Exercise 2.30 Does your circuit from Exercise 2.29 have any potential glitches when one of the inputs changes? If not, explain why not. If so, show how to modify the circuit to eliminate the glitches.

Exercise 2.31 Find a minimal Boolean equation for the function in Figure 2.86. Remember to take advantage of the don't care entries.

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.86 **Truth table for Exercise 2.31**

Exercise 2.32 Sketch a circuit for the function from Exercise 2.31.

Exercise 2.33 Ben Bitdiddle will enjoy his picnic on sunny days that have no ants. He will also enjoy his picnic any day he sees a hummingbird, as well as on days where there are ants and ladybugs. Write a Boolean equation for his enjoyment (E) in terms of sun (S), ants (A), hummingbirds (H), and ladybugs (L).

Exercise 2.34 Complete the design of the seven-segment decoder segments S_c through S_g (see Example 2.10):

(a) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 must produce blank (0) outputs.

(b) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 are don't cares.

(c) Sketch a reasonably simple gate-level implementation of part (b). Multiple outputs can share gates where appropriate.

Exercise 2.35 A circuit has four inputs and two outputs. The inputs $A_{3:0}$ represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.

Exercise 2.36 A *priority encoder* has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE or 0 if none of the inputs are TRUE. It also produces an output *NONE* that is TRUE if none of the inputs are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2.0}$ and *NONE*. For example, if the input is 00100000, the output *Y* should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.37 Design a modified priority encoder (see Exercise 2.36) that receives an 8-bit input, $A_{7:0}$, and produces two 3-bit outputs, $Y_{2:0}$ and $Z_{2:0}$. *Y* indicates the most significant bit of the input that is TRUE. *Z* indicates the second most significant bit of the input that is TRUE. *Y* should be 0 if none of the inputs are TRUE. *Z* should be 0 if no more than one of the inputs is TRUE. Give a simplified Boolean equation for each output and sketch a schematic.

Exercise 2.38 An *M*-bit *thermometer code* for the number *k* consists of *k* 1's in the least significant bit positions and *M* – *k* 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has *N* inputs and 2^N–1 outputs. It produces a 2^N–1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output and sketch a schematic.

Exercise 2.39 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.87.

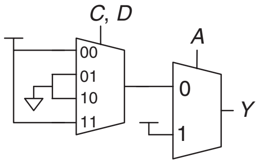


Figure 2.87 Multiplexer circuit for Exercise 2.39

Exercise 2.40 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.88.

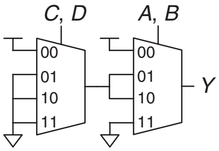


Figure 2.88 Multiplexer circuit for Exercise 2.40

Exercise 2.41 Implement the function from Figure 2.80(b) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and one inverter

(c) a 2:1 multiplexer and two other logic gates

Exercise 2.42 Implement the function from Exercise 2.17(a) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and no other gates

(c) a 2:1 multiplexer, one OR gate, and an inverter

Exercise 2.43 Determine the propagation delay and contamination delay of the circuit in Figure 2.83. Use the gate delays given in Table 2.8.

Exercise 2.44 Determine the propagation delay and contamination delay of the circuit in Figure 2.84. Use the gate delays given in Table 2.8.

Table 2.8 Gate delays for Exercises 2.43–2.45 and 2.47–2.48

Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

Exercise 2.45 Sketch a schematic for a fast 3:8 decoder. Suppose gate delays are given in Table 2.8 (and only the gates in that table are available). Design your decoder to have the shortest possible critical path and indicate what that path is. What are its propagation delay and contamination delay?

Exercise 2.46 Design an 8:1 multiplexer with the shortest possible delay from the data inputs to the output. You may use any of the gates from Table 2.7 on page 90. Sketch a schematic. Using the gate delays from the table, determine this delay.

Exercise 2.47 Redesign the circuit from Exercise 2.35 to be as fast as possible. Use only the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.48 Redesign the priority encoder from Exercise 2.36 to be as fast as possible. You may use any of the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.49 Another way to think about transistor-level design (see Section 1.7) is to use De Morgan's theorem to consider the pull-up and pull-down networks. Design the pull-down network of a transistor-level gate directly from the equations below. Then, apply De Morgan's theorem to the equations and draw the pull-up network using that rewritten equation. Also, state the numbers of transistors used. Do not forget to draw (and count) the inverters needed to complement the inputs, if needed.

(a) $W = \overline{A + BC + \overline{C}D}$

(b) $X = \overline{\overline{A}(B + C + D) + A\overline{D}}$

(c) $Y = \overline{\overline{A}(BC + \overline{B}\,\overline{C}) + A\overline{B}C}$

Exercise 2.50 Repeat Exercise 2.49 for the equations below.

(a) $W = \overline{(A + B)(C + D)}$

(b) $X = \overline{\overline{A}B(C + D) + A\overline{D}}$

(c) $Y = \overline{\overline{A}(B + \overline{C}D) + A\overline{B}CD}$

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 2.1 Sketch a schematic for the two-input XOR function, using only NAND gates. How few can you use?

Question 2.2 Design a circuit that will tell whether a given month has 31 days in it. The month is specified by a 4-bit input $A_{3:0}$. For example, if the inputs are 0001, the month is January, and if the inputs are 1100, the month is December. The circuit output Y should be HIGH only when the month specified by the inputs has 31 days in it. Write the simplified equation, and draw the circuit diagram using a minimum number of gates. (Hint: Remember to take advantage of don't cares.)

Question 2.3 What is a tristate buffer? How and why is it used?

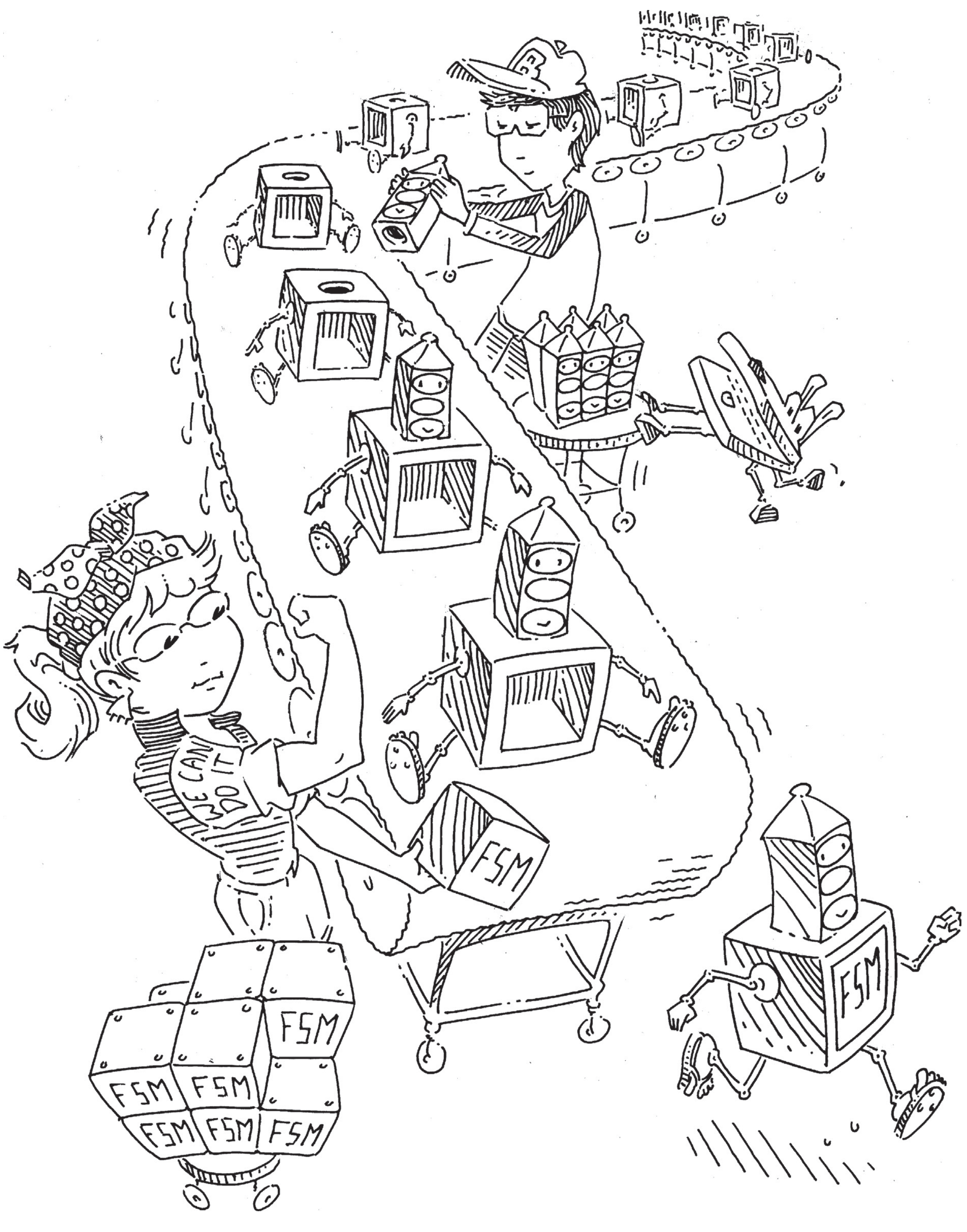
Question 2.4 A gate or set of gates is universal if it can be used to construct any Boolean function. For example, the set {AND, OR, NOT} is universal.

(a) Is an AND gate by itself universal? Why or why not?

(b) Is the set {OR, NOT} universal? Why or why not?

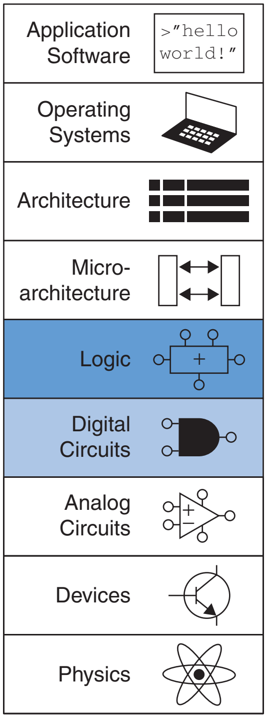
(c) Is a NAND gate by itself universal? Why or why not?

Question 2.5 Explain why a circuit's contamination delay might be less than (instead of equal to) its propagation delay.


WE CAN DO IT
FSM
FSM
FSM
FSM
FSM
FSM
FSM
FSM

Sequential Logic Design

3



3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase speed.

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs,

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00003-9

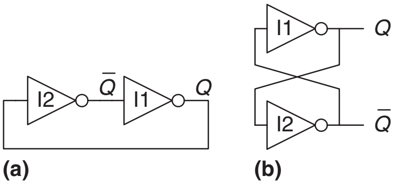


Figure 3.1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Q and $\overline{Q}$. Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on $\overline{Q}$, and $\overline{Q}$ depends on Q.

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- *Case I:* $Q = 0$
  As shown in Figure 3.2(a), I2 receives a FALSE input, Q, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on Q. This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.
- *Case II:* $Q = 1$
  As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input and produces a TRUE output on Q. This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state, which will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is

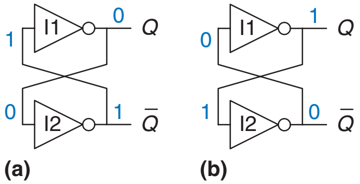


Figure 3.2 Bistable operation of cross-coupled inverters

known, $\overline{Q}$ is also known. On the other hand, $\overline{Q}$ is also an acceptable choice for the state variable.

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.

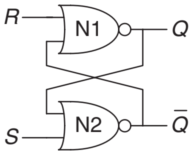


Figure 3.3 **SR latch schematic**

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S.

- *Case I:* $R = 1, S = 0$
  N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.
- *Case II:* $R = 0, S = 1$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.
- *Case III:* $R = 1, S = 1$
  N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence, Q and $\overline{Q}$ are both FALSE.
- *Case IV:* $R = 0, S = 0$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are

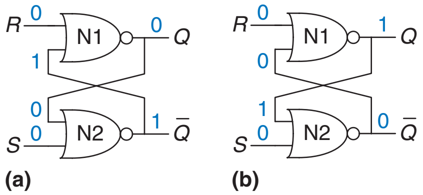


Figure 3.4 **Bistable states of SR latch**

stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1, so we can solve the problem by checking what happens in each of these subcases.

- *Case IVa:* $Q = 0$
  Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$, as shown in Figure 3.4(a). Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE, just as we had assumed.
- *Case IVb:* $Q = 1$
  Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$, as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE, just as we had assumed.

Putting this all together, suppose that Q has some known prior value, which we will call Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1 and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 **SR latch truth table**

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and $\overline{Q}$, are normally complementary. When R is asserted, Q is reset to 0 and $\overline{Q}$ does the opposite. When S is asserted, Q is set to 1 and $\overline{Q}$ does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev}. Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

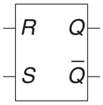


Figure 3.6 **SR latch symbol**

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be accounted for by the single state variable Q. No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

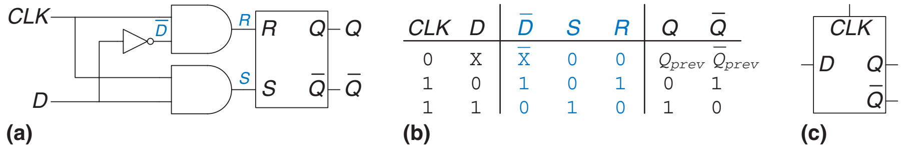
The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes $\overline{D}$, S, and R. If $CLK = 0$, both S and R are FALSE, regardless of the value of D. If $CLK = 1$, one AND gate will produce TRUE and the other FALSE depending on the value of D. Given S and R, Q and $\overline{Q}$ are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev}. When $CLK = 1$, $Q = D$. In all cases, $\overline{Q}$ is the complement of Q, as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.



CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

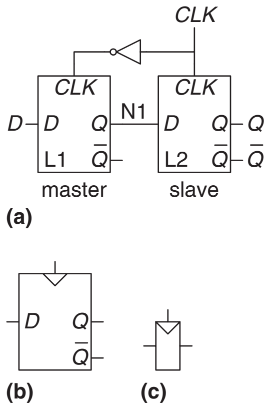


Figure 3.8 **D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol**

3.2.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *leader*. The second latch, L2, is called the *follower*, because it follows whatever L1 does.

The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the $\overline{Q}$ output is not needed, the symbol is often condensed, as in Figure 3.8(c).

When $CLK = 0$, the leader (latch L1) is transparent and the follower (L2) is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the leader (L1) goes opaque and the follower (L2) becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D. Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

In other words, *a D flip-flop copies D to Q on the rising edge of the clock and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as an *edge-triggered flip-flop* or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The $\overline{Q}$ output is often omitted when it is not needed.

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term *flip-flop* or *latch* by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation, using transmission gates.

3.2.4 Register

An *N*-bit register is a bank of *N* flip-flops that share a common *CLK* input so that all bits of the register are updated at the same time. Registers are the key building block of most sequential circuits.

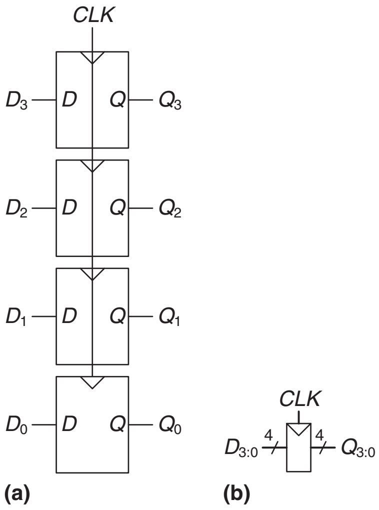


Figure 3.9 A 4-bit register: (a) schematic and (b) symbol

Figure 3.9 shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer passes the value *D* if *EN* is TRUE or recycles the old state from *Q* if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*. If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is FALSE, the *CLK* input is also FALSE and the flip-flop retains its old value. Notice that *EN* must

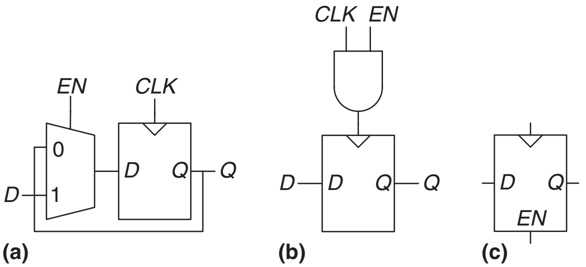


Figure 3.10 Enabled flip-flop: (a, b) schematics, (c) symbol

not change while *CLK* = 1 lest the flip-flop see a clock *glitch* (switch at an incorrect time). Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors, as we will see in Section 3.5.3, so do it only if you are sure you know what you are doing. The symbol for an enabled flip-flop is given in Figure 3.10(c).

3.2.6 Resettable Flip-Flop

A *resettable flip-flop* adds another input, called *RESET*. When *RESET* is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When *RESET* is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as *RESET* becomes TRUE, independent of *CLK*.

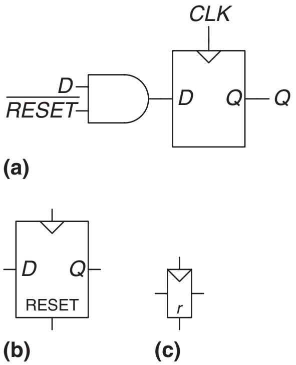


Figure 3.11 Synchronously resettable flip-flop: (a) schematic, (b, c) symbols

Figure 3.11(a) shows how to construct a synchronously resettable flip-flop from an ordinary D flip-flop and an AND gate. When the signal $\overline{RESET}$ is FALSE, the AND gate forces a 0 into the input of the flip-flop. When $\overline{RESET}$ is TRUE, the AND gate passes *D* to the flip-flop. In this example, $\overline{RESET}$ is an *active low* signal, meaning that the reset signal performs its function—in this case, resetting the flip-flop—when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high reset signal instead. Figures 3.11(b) and 3.11(c) show symbols for the resettable flip-flop with active high reset.

Asynchronously resettable flip-flops require modifying the internal structure of the flip-flop and are left to you to design in Exercise 3.13. Both synchronously and asynchronously resettable flip-flops are frequently available to the designer as standard components.

As you might imagine, settable flip-flops are also occasionally used. They load a 1 into the flip-flop when SET is asserted and they, too, come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into *N*-bit registers.

3.2.7 Transistor-Level Latch and Flip-Flop Designs*

Example 3.1 showed that latches and flip-flops require a large number of transistors when built from logic gates. But the fundamental role of a latch is to be transparent or opaque, much like a switch. Recall from Section 1.7.7 that a transmission gate is an efficient way to build a CMOS switch, so we might expect that we could take advantage of transmission gates to reduce the transistor count.

A compact D latch can be constructed from a single transmission gate, as shown in Figure 3.12(a). When $CLK = 1$ and $\overline{CLK} = 0$, the transmission gate is ON, so D flows to Q and the latch is transparent. When $CLK = 0$ and $\overline{CLK} = 1$, the transmission gate is OFF, so Q is isolated from D and the latch is opaque. This latch suffers from two major limitations:

- *Floating output node:* When the latch is opaque, Q is not held at its value by any gates. Thus, Q is called a *floating* or *dynamic* node. After some time, noise and charge leakage may disturb the value of Q.
- *No buffers:* The lack of buffers has caused malfunctions on several commercial chips. A spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when $CLK = 0$. Likewise, a spike on D above V_{DD} can turn on the pMOS transistor even when $CLK = 0$. And the transmission gate is symmetric, so it could be driven backward with noise on Q, affecting the input D. The general rule is that neither the input of a transmission gate nor the state node of a sequential circuit should ever be exposed to the outside world, where noise is likely.

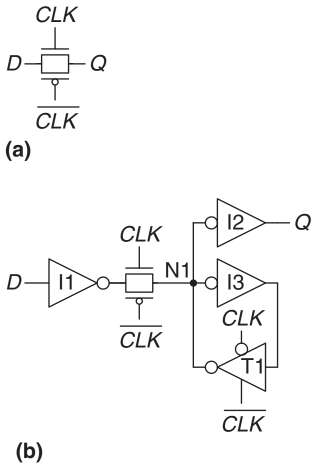


Figure 3.12 D latch schematic

Figure 3.12(b) shows a more robust 12-transistor D latch used on modern commercial chips. It is still built around a clocked transmission gate, but it adds inverters I1 and I2 to buffer the input and output. The state of the latch is held on node N1. Inverter I3 and the tristate buffer, T1, provide feedback to turn N1 into a *static node*. If a small amount of noise occurs on N1 while $CLK = 0$, T1 will drive N1 back to a valid logic value.

Figure 3.13 shows a D flip-flop constructed from two static latches controlled by $\overline{CLK}$ and CLK. Some redundant internal inverters have been removed, so the flip-flop requires only 20 transistors.

This circuit assumes that CLK and $\overline{CLK}$ are both available. If not, two more transistors are needed to invert the clock signal.

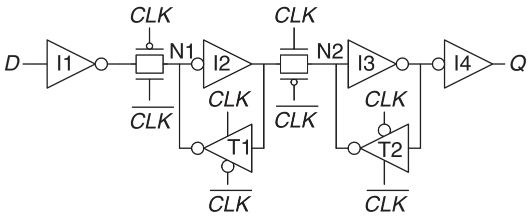


Figure 3.13 D flip-flop schematic

3.2.8 Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits. Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when $CLK = 1$, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state. A register is a bank of several D flip-flops that share a common CLK signal.

Example 3.2 FLIP-FLOP AND LATCH COMPARISON

Ben Bitdiddle applies the D and CLK inputs shown in Figure 3.14 to a D latch and a D flip-flop. Help him determine the output, Q, of each device.

Solution Figure 3.15 shows the output waveforms, assuming a small delay for Q to respond to input changes. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First, consider the latch. On the first rising edge of CLK, $D = 0$, so Q definitely becomes 0. Each time D changes while $CLK = 1$, Q also follows. When D changes while $CLK = 0$, D is ignored. Now, consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.

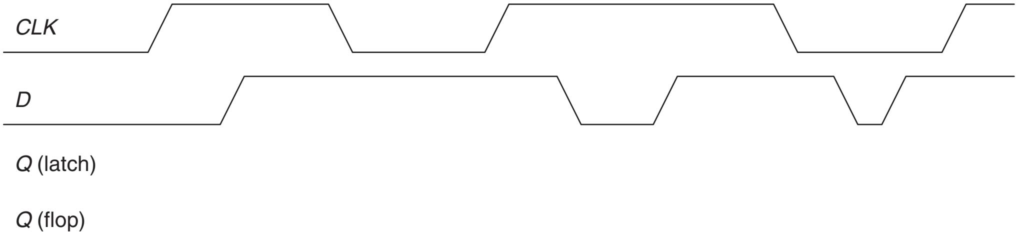


Figure 3.14 Example waveforms

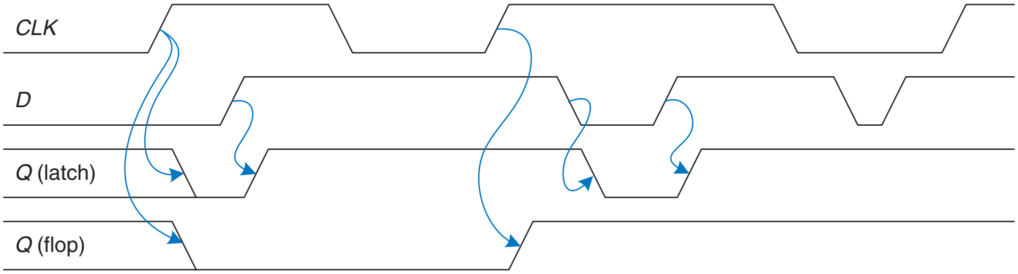


Figure 3.15 Solution waveforms

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational—that is, those whose output cannot be determined simply by looking at the current inputs. Some sequential circuits are just plain kooky. This section begins by examining some of those curious circuits. It then introduces the notion of synchronous sequential circuits and the dynamic discipline. By disciplining ourselves to synchronous sequential circuits, we can develop easy, systematic ways to analyze and design sequential systems.

3.3.1 Some Problematic Circuits

Example 3.3 ASTABLE CIRCUITS

Alyssa P. Hacker encounters three misbegotten inverters who have tied themselves in a loop, as shown in Figure 3.16. The output of the third inverter is *fed back* to the first inverter. Each inverter has a propagation delay of 1 ns. Help Alyssa determine what the circuit does.

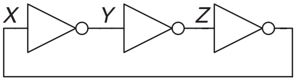


Figure 3.16 Three-inverter loop

Solution Suppose that node X is initially 0. Then, $Y = 1$, $Z = 0$, and, hence, $X = 1$, which is inconsistent with our original assumption. The circuit has no stable states and is said to be *unstable* or *astable*. Figure 3.17 shows the behavior of the circuit. If X rises at time 0, Y will fall at 1 ns, Z will rise at 2 ns, and X will fall again at 3 ns. In turn, Y will rise at 4 ns, Z will fall at 5 ns, and X will rise again at 6 ns; then, the pattern will repeat. Each node oscillates between 0 and 1, with a *period* (repetition time) of 6 ns. This circuit is called a *ring oscillator*.

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

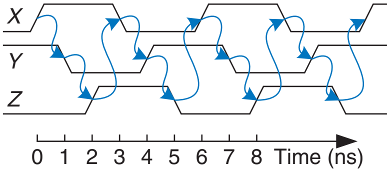


Figure 3.17 Ring oscillator waveforms

Example 3.4 RACE CONDITIONS

Ben Bitdiddle designed a new D latch that he claims is better than the one in Figure 3.7 because it uses fewer gates. He has written the truth table to find the output, Q, given the two inputs, D and CLK, and the old state of the

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

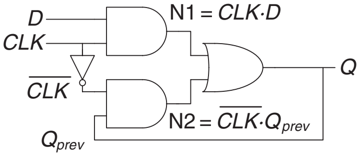


Figure 3.18 **An improved (?) D latch**

latch, Q_{prev}. Based on this truth table, he has derived Boolean equations. He obtains Q_{prev} by feeding back the output, Q. His design is shown in Figure 3.18. Does his latch work correctly, independent of the delays of each gate?

Solution Figure 3.19 shows that the circuit has a *race condition* that causes it to fail when certain gates are slower than others. Suppose that $CLK = D = 1$. The latch is transparent and passes D through to make $Q = 1$. Now, CLK falls. The latch should remember its old value, keeping $Q = 1$. However, suppose that the delay through the inverter from CLK to $\overline{CLK}$ is rather long compared with the delays of the AND and OR gates. Then, nodes N1 and Q may both fall before $\overline{CLK}$ rises. In such a case, N2 will never rise and Q becomes stuck at 0.

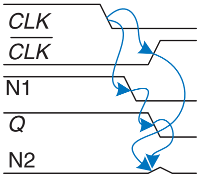


Figure 3.19 **Latch waveforms illustrating race condition**

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. One circuit may work, while a seemingly identical one built from gates with slightly different delays may not work. Or the circuit may work only at certain temperatures or voltages at which the delays are just right. These malfunctions are extremely difficult to track down.

3.3.2 Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. If inputs are applied to combinational logic, the outputs will always settle to the correct value within a propagation delay. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior. Analyzing such circuits for problems is time-consuming, and many bright people have made mistakes.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

Recall that a circuit is defined by its input and output terminals and its functional and timing specifications. A sequential circuit has a finite set of discrete *states* {S0, S1, S2,…}. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The functional specification details the next state and the value of each output for each possible combination of current state and input values. The timing specification consists of an upper bound, t_{pcq}, and a lower bound, t_{ccq}, on the time from the rising edge of the clock until the *output* changes, as well as *setup* and *hold* times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

t_{pcq} stands for the time of propagation from clock to Q, where Q indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to Q. These are analogous to t_{pd} and t_{cd} in combinational logic.

The rules of *synchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements, such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

This definition of a synchronous sequential circuit is sufficient but more restrictive than necessary. For example, in high-performance microprocessors, some registers may receive delayed or gated clocks to squeeze out the last bit of performance or power. Similarly, some microprocessors use latches instead of registers. However, the definition is adequate for all of the synchronous sequential circuits covered in this book and for most commercial digital systems.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the simplest synchronous sequential circuit. It has one input, D, one clock, CLK, one output, Q, and two states, {0, 1}. The functional specification for a flip-flop is that the next state is D and that the output, Q, is the current state, as shown in Figure 3.20.

We often call the current state variable S and the next state variable S'. In this case, the prime after S indicates next state, not inversion. The timing of sequential circuits will be analyzed in Section 3.5.

Two other common types of synchronous sequential circuits are called finite state machines and pipelines. These will be covered later in this chapter.

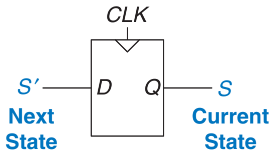


Figure 3.20 Flip-flop current state and next state

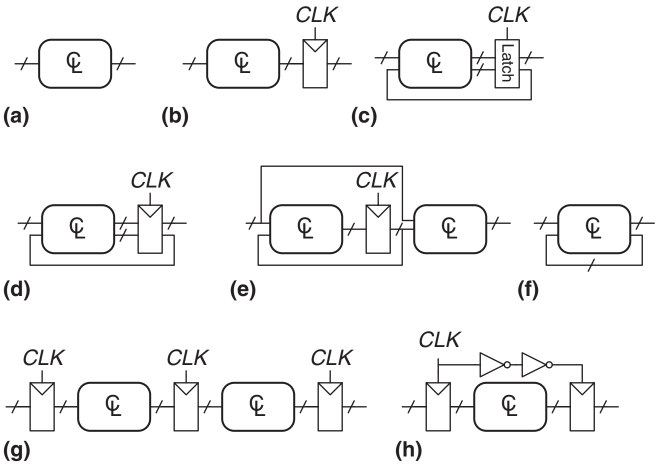


Figure 3.21 Example circuits

Example 3.5 SYNCHRONOUS SEQUENTIAL CIRCUITS

Which of the circuits in Figure 3.21 are synchronous sequential circuits?

Solution Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines, which are discussed in Section 3.4. (f) is neither combinational nor synchronous sequential because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline, which we will study in Section 3.6. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3.3.3 Synchronous and Asynchronous Circuits

Asynchronous design in theory is more general than synchronous design because the timing of the system is not limited by clocked registers. Just as analog circuits are more general than digital circuits because analog circuits can use any voltage, asynchronous circuits are more general than synchronous circuits because they can use any kind of feedback. However, synchronous circuits have proved to be easier to design and use than asynchronous circuits, just as digital are easier than analog circuits. Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous.

Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can also improve synchronous circuits.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, "Gedanken-experiments on Sequential Machines," in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy (1927–2010) published his article, "A Method of Synthesizing Sequential Circuits," in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic

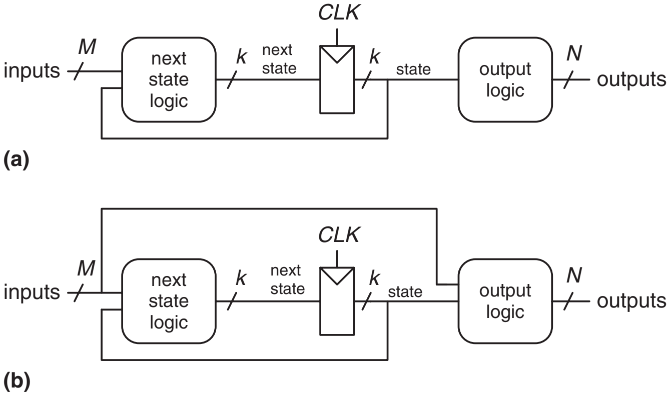


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

Avenue. They are busy reading about FSMs in their favorite textbook and aren't looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long as

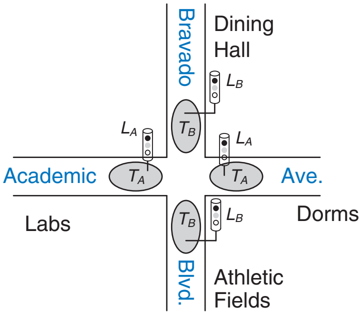


Figure 3.23 Campus map

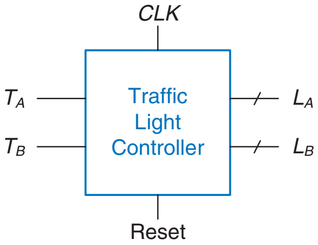


Figure 3.24 Black box view of finite state machine

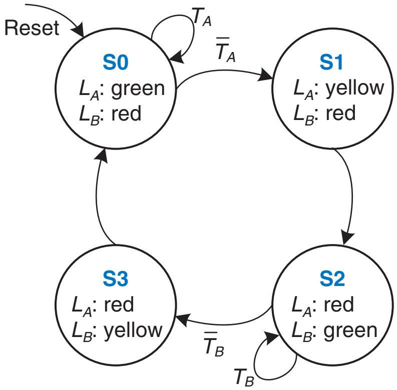


Figure 3.25 State transition diagram

traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset, pointing from outer space into state S0 indicates that the system should enter that state upon reset regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2 at the clock edge. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also, note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The current state is often referred to simply as the *state* of the system.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Notice that states are designated as S0, S1, etc. The subscripted versions, S_0, S_1, etc., refer to the state bits.

Table 3.1 State transition table

Current	Inputs		Next State
State S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines the next state, S', as a function of the current state, S, and the inputs.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B \\ S_0' &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.1}$$

The equations can be simplified, using Karnaugh maps. However, doing it by inspection is often easier. For example, the T_B and $\overline{T}_B$ terms in the S_1' equation are clearly redundant. Thus, S_1' reduces to an XOR operation. Equation 3.2 gives the simplified *next state equations*.

Table 3.4 State transition table with binary encodings

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 Output table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S_1' &= S_1 \oplus S_0 \\ S_0' &= \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B \end{aligned} \tag{3.2}$$

Similarly, Ben writes an *output table* (Table 3.5) that indicates, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state from the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs from the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows *CLK*, *Reset*, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edges of *CLK* when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram, S

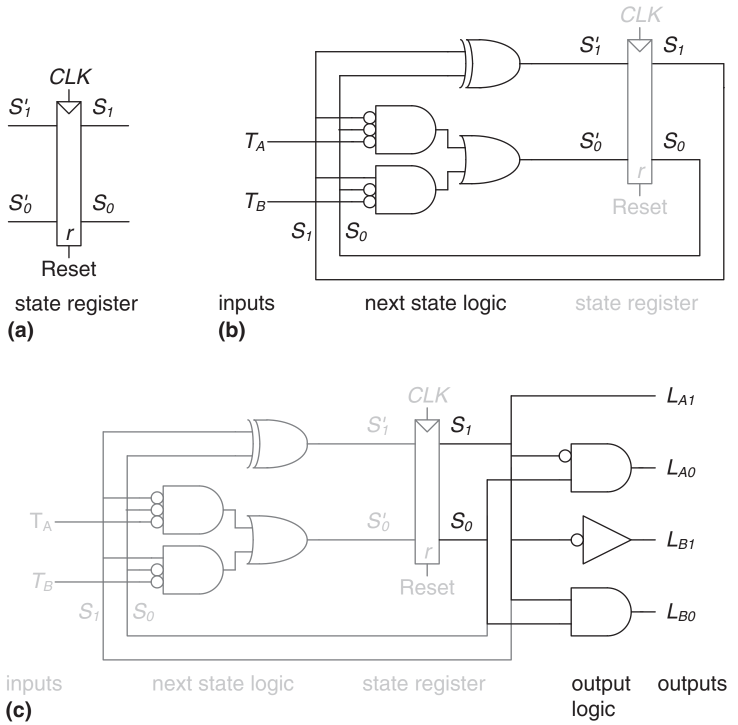


Figure 3.26 State machine circuit for traffic light controller

This schematic uses some AND gates, with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the nonbubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

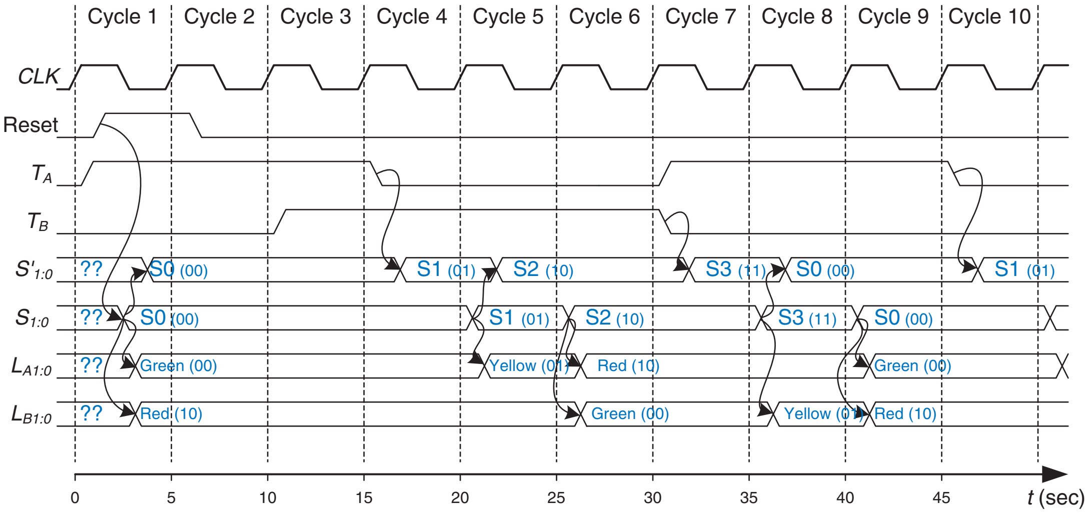
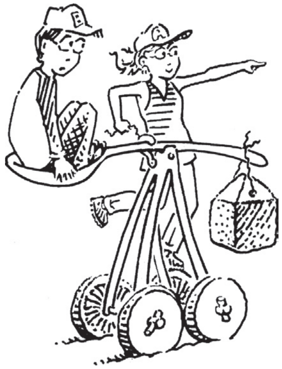


Figure 3.27 Timing diagram for traffic light controller

immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2, in which L_A turns red and L_B turns green. The controller waits in state S2 until all traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. Five seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him and Alyssa to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.



3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding*, as was used in the traffic light controller example, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states needs only $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called *one-hot* because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

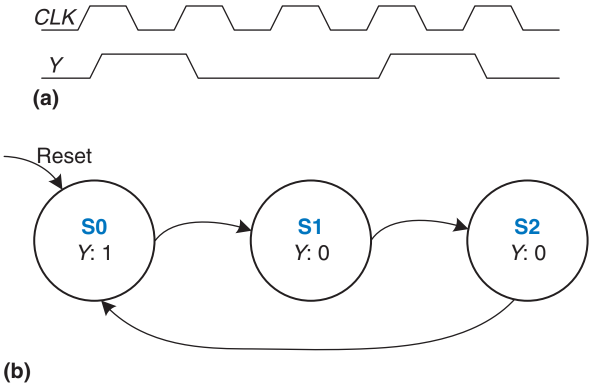


Figure 3.28 Divide-by-3 counter (a) waveform and (b) state transition diagram

Table 3.6 Divide-by-3 counter state transition table

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 Divide-by-3 counter output table

Current State	Output
S0	1
S1	0
S2	0

Solution Tables 3.6 and 3.7 show the abstract state transition and output tables, respectively, before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next state and output equations are:

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next state and output equations are as follows:

$$\begin{aligned} S_2' &= S_1 \\ S_1' &= S_0 \\ S_0' &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S_0'. Also, observe that one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

Table 3.8 One-hot and binary encodings for divide-by-3 counter

	One-Hot Encoding			Binary Encoding	
State	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Table 3.9 State transition table with binary encoding

Current State		Next State	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State			Next State		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

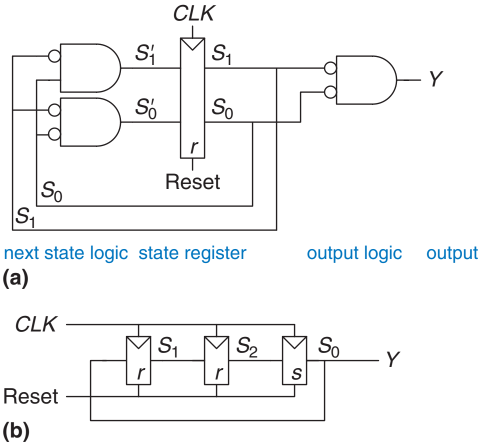


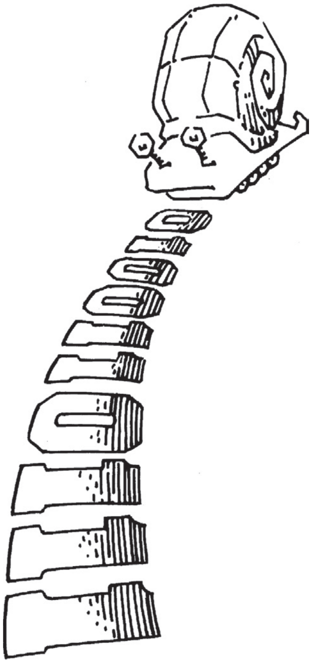
Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).

Example 3.7 MOORE VERSUS MEALY MACHINES



Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are 01. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as Alyssa's snail crawls along the sequence 0100110111.

Solution The Moore machine requires three states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S2 to S1 when the input is 0?

In comparison, the Mealy machine requires only two states, as shown in Figure 3.30(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables, respectively for the Moore machine. The Moore machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, and S2 = 10. Tables 3.13 and 3.14 rewrite the state transition and output tables, respectively, with these encodings.

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that state 11 does not exist. Thus, the corresponding next state and output for the nonexistent state are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$\begin{aligned} S_1' &= S_0 A \\ S_0' &= \overline{A} \end{aligned} \tag{3.8}$$

$$Y = S_1 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires only one bit of state. Consider using a binary state encoding: S0 = 0 and S1 = 1. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$S'_0 = \overline{A} \tag{3.10}$$

$$Y = S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31. The timing diagrams for each machine are shown in Figure 3.32 (see page 133). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

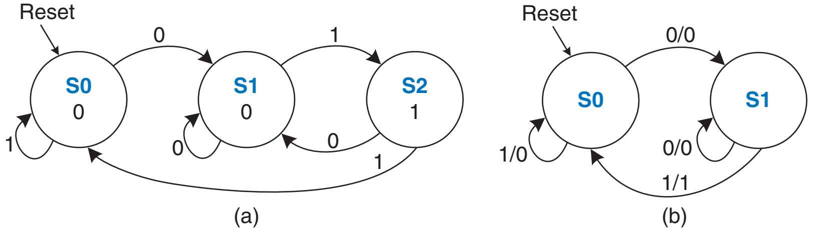


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Table 3.11 Moore state transition table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	1

Table 3.13 Moore state transition table with state encodings

Current State		Input	Next State	
S_1	S_0	A	S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Table 3.14 Moore output table with state encodings

Current State		Output
S_1	S_0	Y
0	0	0
0	1	0
1	0	1

Table 3.15 Mealy state transition and output table

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Table 3.16 Mealy state transition and output table with state encodings

Current State S_0	Input A	Next State S'_0	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

3.4.4 Factoring State Machines

Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines, such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.

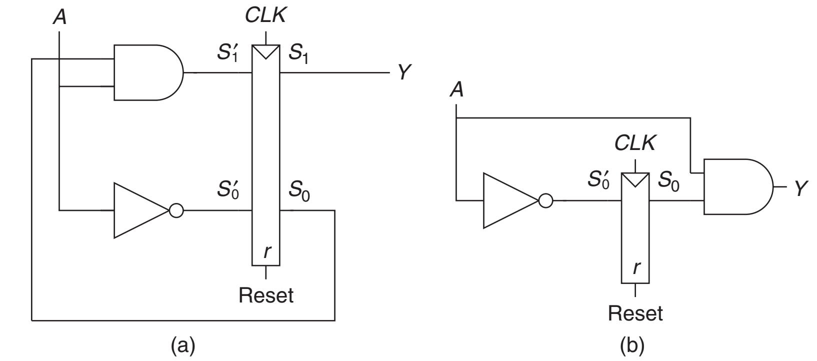


Figure 3.31 FSM schematics for (a) Moore and (b) Mealy machines

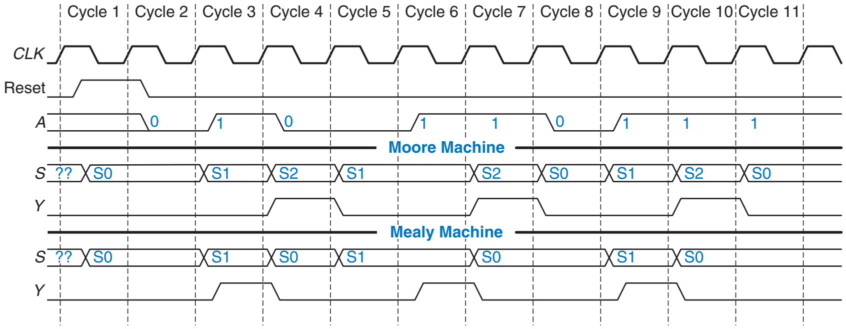


Figure 3.32 Timing diagrams for Moore and Mealy machines

Example 3.8 UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

First, sketch a state transition diagram for a single FSM, as shown in Figure 3.33(a). Then, sketch the state transition diagrams for two interacting FSMs, as shown in Figure 3.33(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors, T_A and T_B.

Solution Figure 3.34(a) shows the single FSM design. States S0 to S3 handle normal mode. States S4 to S7 handle parade mode. The two halves of the diagram are almost identical, but in parade mode, the FSM remains in S6 with a green light on Bravado Blvd. The P and R inputs control movement between these two halves. The FSM is messy and tedious to design. Figure 3.34(b) shows the factored FSM design. The Mode FSM has two states to track whether the lights are in normal or parade mode. The Lights FSM is modified to remain in S2 while M is TRUE.

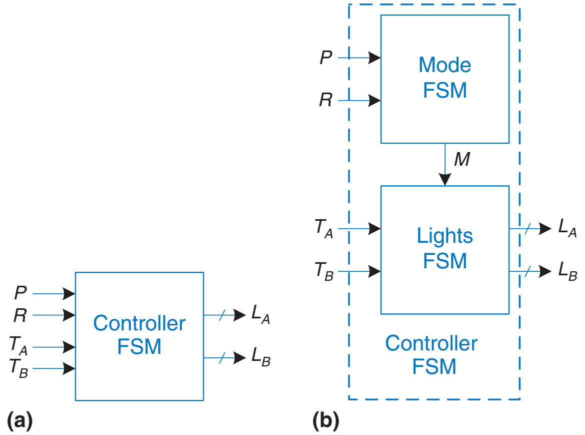


Figure 3.33 (a) Single and (b) factored designs for modified traffic light controller FSM

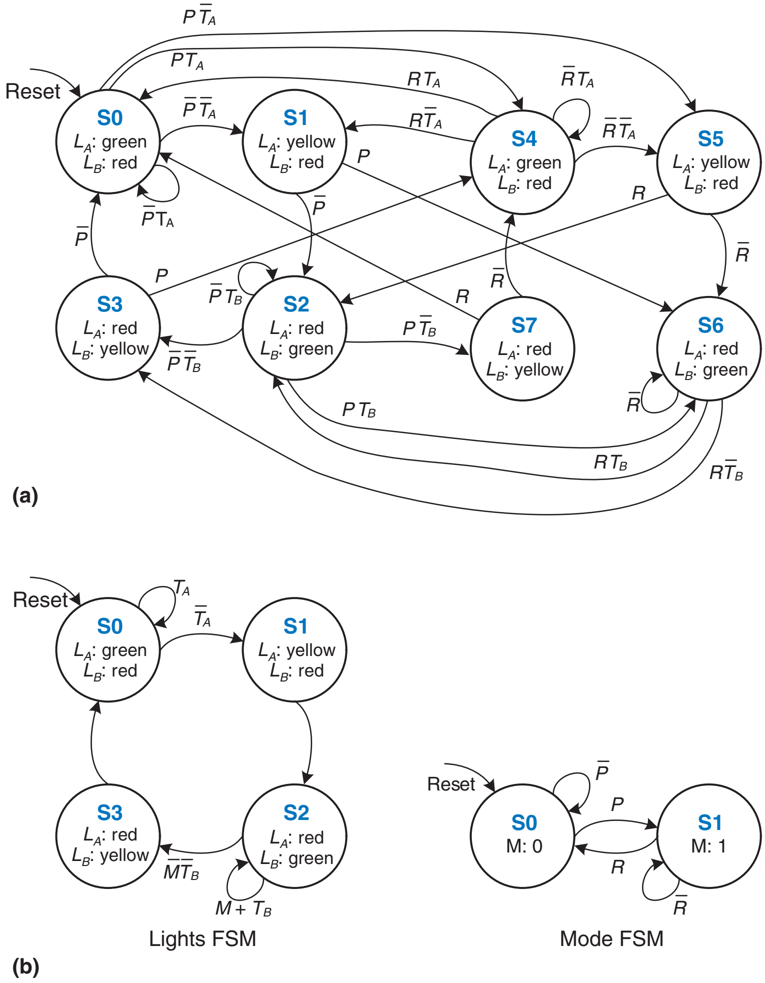


Figure 3.34 State transition diagrams: (a) unfactored, (b) factored

3.4.5 Deriving an FSM from a Schematic

Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design. This process can be necessary, for example, when taking on an incompletely documented project or reverse engineering somebody else's system.

- Examine circuit, stating inputs, outputs, and state bits.
- Write next state and output equations.
- Create next state and output tables.
- Reduce the next state table to eliminate unreachable states.
- Assign each valid state bit combination a name.
- Rewrite next state and output tables with state names.
- Draw state transition diagram.
- State in words what the FSM does.

In the final step, be careful to succinctly describe the overall purpose and function of the FSM—do not simply restate each transition of the state transition diagram.

Example 3.9 DERIVING AN FSM FROM ITS CIRCUIT

Alyssa P. Hacker arrives home, but her keypad lock has been rewired and her old code no longer works. A piece of paper is taped to it showing the circuit diagram in Figure 3.35. Alyssa thinks the circuit could be a finite state machine and decides to derive the state transition diagram to see whether it helps her get in the door.

Solution Alyssa begins by examining the circuit. The input is $A_{1:0}$ and the output is *Unlock*. The state bits are already labeled in Figure 3.35. This is a Moore

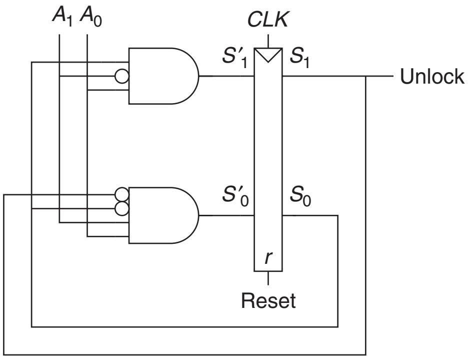


Figure 3.35 Circuit of found FSM for Example 3.9

machine because the output depends only on the state bits. From the circuit, she writes down the next state and output equations directly:

$$\begin{aligned} S'_1 &= S_0 \overline{A_1} A_0 \\ S'_0 &= \overline{S_1}\, \overline{S_0} A_1 A_0 \\ Unlock &= S_1 \end{aligned} \tag{3.12}$$

Next, she writes down the next state and output tables from the equations, as shown in Tables 3.17 and 3.18, respectively, first placing 1's in the tables as indicated by Equation 3.12. She places 0's everywhere else.

Alyssa reduces the table by removing unused states and combining rows using don't cares. The $S_{1:0} = 11$ state is never listed as a possible next state in Table 3.17, so rows with this current state are removed. For current state $S_{1:0} = 10$, the next

Table 3.17 Next state table derived from circuit in Figure 3.35

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	**1**
0	1	0	0	0	0
0	1	0	1	**1**	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	**1**	0
1	1	1	0	0	0
1	1	1	1	0	0

Table 3.18 Output table derived from circuit in Figure 3.35

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	**1**
1	1	**1**

Table 3.19 Reduced next state table

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Table 3.20 Reduced output table

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	1

Table 3.21 Symbolic next state table

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Table 3.22 Symbolic output table

Current State S	Output *Unlock*
S0	0
S1	0
S2	1

state is always $S_{1:0} = 00$, independent of the inputs, so don't cares are inserted for the inputs. The reduced tables are shown in Tables 3.19 and 3.20.

She assigns names to each state bit combination: S0 is $S_{1:0} = 00$, S1 is $S_{1:0} = 01$, and S2 is $S_{1:0} = 10$. Tables 3.21 and 3.22 show the next state and output tables, respectively, with state names.

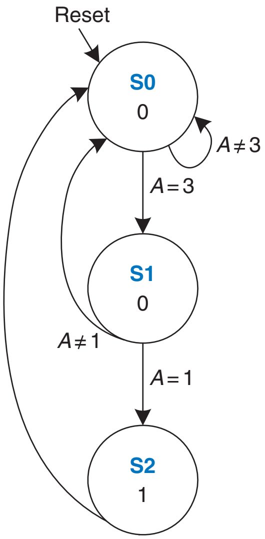


Figure 3.36 State transition diagram of found FSM from Example 3.9

Alyssa writes down the state transition diagram shown in Figure 3.36 using Tables 3.21 and 3.22. By inspection, she can see that the finite state machine unlocks the door only after detecting an input value, $A_{1:0}$, of three followed by an input value of one. The door is then locked again. Alyssa tries this code on the door keypad and the door opens!

3.4.6 FSM Review

Finite state machines are a powerful way to systematically design sequential circuits from a written specification. Use the following procedure to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.
- For a Moore machine:
  - Write a state transition table.
  - Write an output table.
- For a Mealy machine:
  - Write a combined state transition and output table.
- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

We will repeatedly use FSMs to design complex digital systems throughout this book.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?

This problem is similar to that faced by a camera when snapping a picture. Imagine photographing a frog jumping from a lily pad into the lake. If you take the picture before the jump, you will see a frog on a lily pad. If you take the picture after the jump, you will see ripples in the water. But if you take it just as the frog jumps, you may see a blurred image of the frog stretching from the lily pad into the water. A camera is characterized by its *aperture time*, during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a *setup* time and a *hold* time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time. By taking advantage of the dynamic discipline, we can think of time in discrete units called clock cycles, just as we think of signal levels as discrete 1's and 0's. A signal may glitch and oscillate wildly for some bounded amount of time. Under the dynamic discipline, we are concerned only about its final value at the end of the clock cycle, after it has settled to a stable value. Hence, we can simply write $A[n]$, the value of signal A at the end of the nth clock cycle, where n is an integer, rather than $A(t)$, the value of A at some instant t, where t is any real number.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not reach all flip-flops at precisely the same time. This variation in time, called clock skew, further increases the necessary clock period.

Sometimes it is impossible to satisfy the dynamic discipline, especially when interfacing with the real world. For example, consider a circuit with an input coming from a button. A monkey might press the button just as the clock rises. This can result in a phenomenon called *metastability*, where the flip-flop captures a value partway between 0 and 1 that can take an unlimited amount of time to resolve into a good logic value. The solution to such asynchronous inputs is to use a synchronizer, which has a very small (but nonzero) probability of producing an illegal logic value.

We expand on all of these ideas in the rest of this section.

3.5.1 The Dynamic Discipline

So far, we have focused on the functional specification of sequential circuits. Recall that a synchronous sequential circuit, such as a flip-flop or FSM, also has a timing specification, as illustrated in Figure 3.37. When the clock rises, the output (or outputs) may start to change after the clock-to-Q *contamination delay*, t_{ccq}, and must definitely settle to the final value within the clock-to-Q *propagation delay*, t_{pcq}. These represent the fastest and slowest delays through the circuit, respectively. For the circuit to sample its input correctly, the input (or inputs) must have stabilized at least some *setup time*, t_{setup}, before the rising edge of the clock and must remain stable for at least some *hold time*, t_{hold}, after the rising edge of the clock. The sum of the setup and hold times is called the *aperture time* of the circuit, because it is the total time for which the input must remain stable.

The *dynamic discipline* states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge. By imposing this requirement, we guarantee that the flip-flops sample signals while they are not changing. Because we are concerned only about the final values of the inputs at the time they are sampled, we can treat signals as discrete in time as well as in logic levels.

3.5.2 System Timing

In the three decades from when one of the authors' families bought an Apple II+ computer to the present time of writing, microprocessor clock frequencies have increased from 1 MHz to several GHz, a factor of more than 1000. This speedup partially explains the revolutionary changes computers have made in society.

The *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*. All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time. Frequency is measured in units of Hertz (Hz), or cycles per second: 1 megahertz (MHz) = 10^6 Hz, and 1 gigahertz (GHz) = 10^9 Hz.

Figure 3.38(a) illustrates a generic path in a synchronous sequential circuit whose clock period we wish to calculate. On the rising edge of the clock, register R1 produces output (or outputs) $Q1$. These signals enter a block of combinational logic, producing $D2$, the input (or inputs) to register R2. The timing diagram in Figure 3.38(b) shows that each output signal may start to change a contamination delay after its

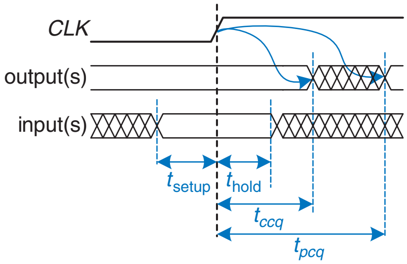


Figure 3.37 Timing specification for synchronous sequential circuit

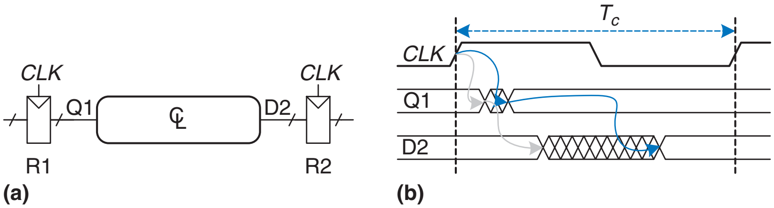


Figure 3.38 Path between registers and timing diagram

input changes and settles to the final value within a propagation delay after its input settles. The gray arrows represent the contamination delay through R1 and the combinational logic, and the blue arrows represent the propagation delay through R1 and the combinational logic. We analyze the timing constraints with respect to the setup and hold time of the second register, R2.

Setup Time Constraint

Figure 3.39 is the timing diagram showing only the maximum delay through the path, indicated by the blue arrows. To satisfy the setup time of R2, $D2$ must settle no later than the setup time before the next clock edge. Hence, we find an equation for the minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \tag{3.13}$$

In commercial designs, the clock period is often dictated by the Director of Engineering or by the marketing department (to ensure a competitive product). Moreover, the flip-flop clock-to-Q propagation delay and setup time, t_{pcq} and t_{setup}, are specified by the manufacturer. Hence, we rearrange Equation 3.13 to solve for the maximum propagation delay through the combinational logic, which is usually the only variable under the control of the individual designer.

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}) \tag{3.14}$$

The term in parentheses, $t_{pcq} + t_{\text{setup}}$, is called the *sequencing overhead*. Ideally, the entire cycle time T_c would be available for useful

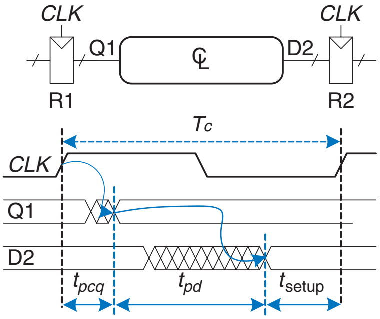


Figure 3.39 Maximum delay for setup time constraint

computation in the combinational logic, t_{pd}. However, the sequencing overhead of the flip-flop cuts into this time. Equation 3.14 is called the *setup time constraint* or *max-delay constraint* because it depends on the setup time and limits the maximum delay through combinational logic.

If the propagation delay through the combinational logic is too great, $D2$ may not have settled to its final value by the time R2 needs it to be stable and samples it. Hence, R2 may sample an incorrect result or even an illegal logic level, a level in the forbidden region. In such a case, the circuit will malfunction. The problem can be solved by increasing the clock period or by redesigning the combinational logic to have a shorter propagation delay.

Hold Time Constraint

The register R2 in Figure 3.38(a) also has a *hold time constraint*. Its input, $D2$, must not change until some time, t_{hold}, after the rising edge of the clock. According to Figure 3.40, $D2$ might change as soon as $t_{ccq} + t_{cd}$ after the rising edge of the clock. Hence, we find

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} \tag{3.15}$$

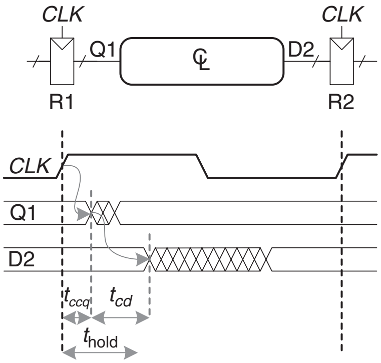
Again, t_{ccq} and t_{hold} are characteristics of the flip-flop that are usually outside the designer's control. Rearranging, we can solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq} \tag{3.16}$$

Equation 3.16 is called the *hold time constraint* or *min-delay constraint* because it limits the minimum delay through combinational logic.

We have assumed that any logic elements can be connected to each other without introducing timing problems. In particular, we would expect that two flip-flops may be directly cascaded as in Figure 3.41 without causing hold time problems.

Figure 3.40 Minimum delay for hold time constraint



In such a case, $t_{cd} = 0$ because there is no combinational logic between flip-flops. Substituting into Equation 3.16 yields the requirement that

$$t_{\text{hold}} \leq t_{ccq} \tag{3.17}$$

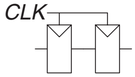


Figure 3.41 Back-to-back flip-flops

In other words, a reliable flip-flop must have a hold time shorter than its contamination delay. Often, flip-flops are designed with $t_{\text{hold}} = 0$ so that Equation 3.17 is always satisfied. Unless noted otherwise, we will usually make that assumption and ignore the hold time constraint in this book.

Nevertheless, hold time constraints are critically important. If they are violated, the only solution is to increase the contamination delay through the logic, which requires redesigning the circuit. Unlike setup time constraints, they cannot be fixed by adjusting the clock period. Redesigning an integrated circuit and manufacturing the corrected design takes months and millions of dollars in today's advanced technologies, so *hold time violations* must be taken extremely seriously.

Putting It All Together

Sequential circuits have setup and hold time constraints that dictate the maximum and minimum delays of the combinational logic between flip-flops. Modern flip-flops are usually designed so that the minimum delay through the combinational logic can be 0—that is, flip-flops can be placed back-to-back. The maximum delay constraint limits the number of consecutive gates on the critical path of a high-speed circuit because a high clock frequency means a short clock period.

Example 3.10 TIMING ANALYSIS

Ben Bitdiddle designed the circuit in Figure 3.42. According to the data sheets for the components he is using, flip-flops have a clock-to-*Q* contamination delay of 30 ps and a propagation delay of 80 ps. They have a setup time of 50 ps and a hold time of 60 ps. Each logic gate has a propagation delay of 40 ps

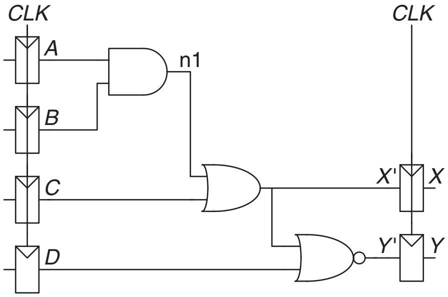


Figure 3.42 Sample circuit for timing analysis

and a contamination delay of 25 ps. Help Ben determine the maximum clock frequency and whether any hold time violations could occur. This process is called *timing analysis*.

Solution Figure 3.43(a) shows waveforms illustrating when the signals might change. The inputs, *A* to *D*, are registered, so they only change shortly after *CLK* rises.

The critical path occurs when $B = 1$, $C = 0$, $D = 0$, and *A* rises from 0 to 1, triggering n1 to rise, X' to rise, and Y' to fall, as shown in Figure 3.43(b). This path involves three gate delays. For the critical path, we assume that each gate requires its full propagation delay. Y' must set up before the next rising edge of the *CLK*. Hence, the minimum cycle time is

$$T_c \geq t_{pcq} + 3t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250\,\text{ps} \tag{3.18}$$

The maximum clock frequency is $f_c = 1/T_c = 4$ GHz.

A short path occurs when $A = 0$ and *C* rises, causing X' to rise, as shown in Figure 3.43(c). For the short path, we assume that each gate switches after only a contamination delay. This path involves only one gate delay, so it may occur after $t_{ccq} + t_{cd} = 30 + 25 = 55\,\text{ps}$. But recall that the flip-flop has a hold time of

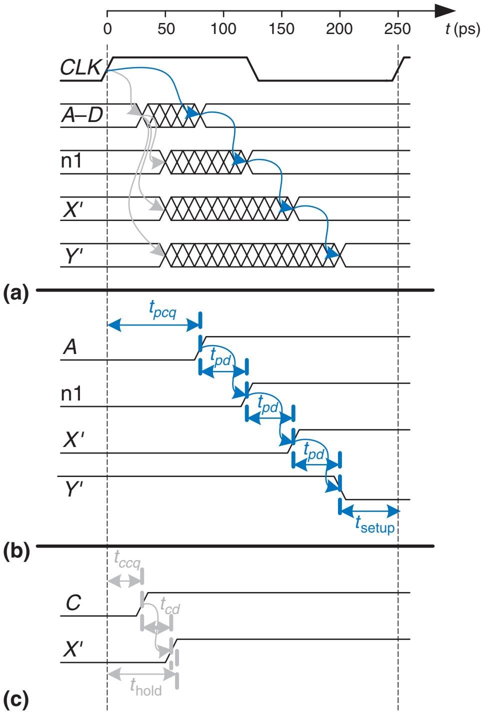


Figure 3.43 Timing diagram: (a) general case, (b) critical path, (c) short path

60 ps, meaning that X' must remain stable for 60 ps after the rising edge of CLK for the flip-flop to reliably sample its value. In this case, $X' = 0$ at the first rising edge of CLK, so we want the flip-flop to capture $X = 0$. Because X' did not hold stable long enough, the actual value of X is unpredictable. The circuit has a hold time violation and may behave erratically at any clock frequency.

Example 3.11 FIXING HOLD TIME VIOLATIONS

Alyssa P. Hacker proposes to fix Ben's circuit by adding buffers to slow down the short paths, as shown in Figure 3.44. The buffers have the same delays as other gates. Help her determine the maximum clock frequency and whether any hold time problems could occur.

Solution Figure 3.45 shows waveforms illustrating when the signals might change. The critical path from A to Y is unaffected because it does not pass through any buffers. Therefore, the maximum clock frequency is still 4 GHz. However, the short paths are slowed by the contamination delay of the buffer. Now, X' will not change until $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ ps. This is after the 60 ps hold time has elapsed, so the circuit now operates correctly.

This example had an unusually long hold time to illustrate the point of hold time problems. Most flip-flops are designed with $t_{\text{hold}} < t_{ccq}$ to avoid such

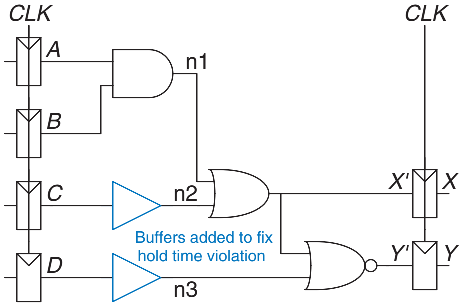


Figure 3.44 Corrected circuit to fix hold time problem

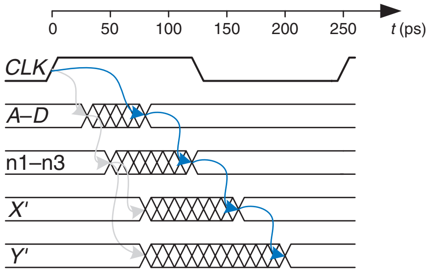


Figure 3.45 Timing diagram with buffers to fix hold time problem

problems. However, some high-performance microprocessors, including the Pentium 4, use an element called a *pulsed latch* in place of a flip-flop. The pulsed latch behaves like a flip-flop but has a short clock-to-Q delay and a long hold time. In general, adding buffers can usually, but not always, solve hold time problems without slowing the critical path.

3.5.3 Clock Skew*

In the previous analysis, we assumed that the clock reaches all registers at exactly the same time. In reality, there is some variation in this time. This variation in clock edges is called *clock skew*. For example, the wires from the clock source to different registers may be of different lengths, resulting in slightly different delays, as shown in Figure 3.46. Noise also results in different delays. Clock gating, described in Section 3.2.5, further delays the clock. If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks. In Figure 3.46, *CLK*2 is *early* with respect to *CLK*1, because the clock wire between the two registers follows a scenic route. If the clock had been routed differently, *CLK*1 might have been early instead. When doing timing analysis, we consider the worst-case scenario so that we can guarantee that the circuit will work under all circumstances.

Figure 3.47 adds skew to the timing diagram from Figure 3.38. The heavy clock line indicates the latest time at which the clock signal might reach any register; the hashed lines show that the clock might arrive up to t_{skew} earlier.

First, consider the setup time constraint shown in Figure 3.48. In the worst case, R1 receives the latest skewed clock and R2 receives the earliest skewed clock, leaving as little time as possible for data to propagate between the registers.

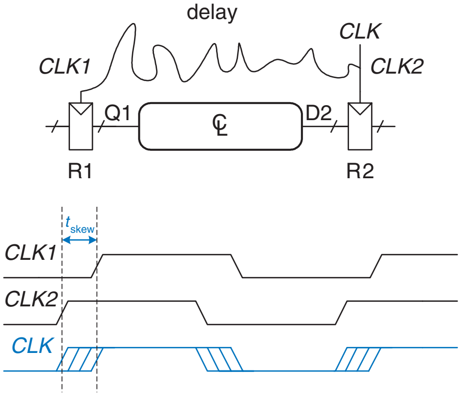


Figure 3.46 Clock skew caused by wire delay

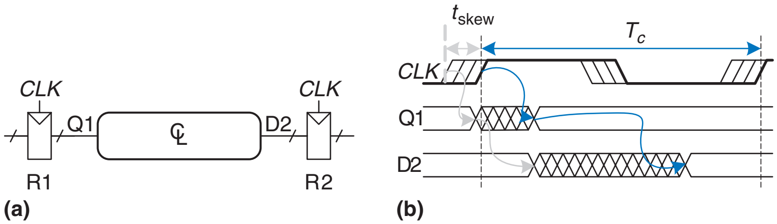


Figure 3.47 Timing diagram with clock skew

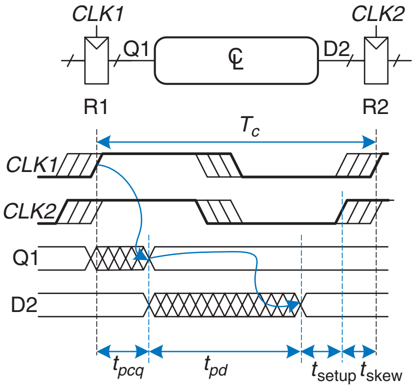


Figure 3.48 Setup time constraint with clock skew

The data propagates through the register and combinational logic and must set up before R2 samples it. Hence, we conclude that

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \tag{3.19}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}} + t_{\text{skew}}) \tag{3.20}$$

Next, consider the hold time constraint shown in Figure 3.49. In the worst case, R1 receives an early skewed clock, *CLK*1, and R2 receives a late skewed clock, *CLK*2. The data zips through the register and combinational logic, but must not arrive until a hold time after the late clock. Thus, we find that

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} \tag{3.21}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \tag{3.22}$$

In summary, clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic. Even if $t_{\text{hold}} = 0$, a pair of back-to-back flip-flops will violate Equation 3.22 if $t_{\text{skew}} > t_{ccq}$. To prevent serious hold time failures, designers must not permit too much

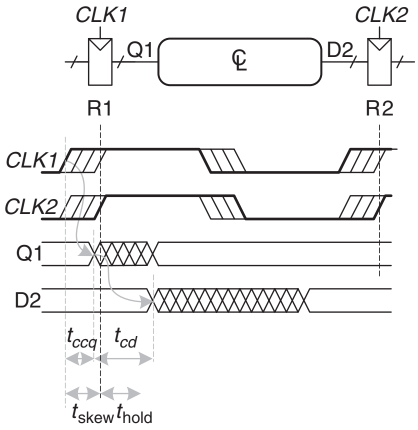


Figure 3.49 **Hold time constraint with clock skew**

clock skew. Sometimes, flip-flops are intentionally designed to be particularly slow (i.e., large t_{ccq}), to prevent hold time problems even when the clock skew is substantial.

Example 3.12 TIMING ANALYSIS WITH CLOCK SKEW

Revisit Example 3.10 and assume that the system has 50 ps of clock skew.

Solution The critical path remains the same, but the setup time is effectively increased by the skew. Hence, the minimum cycle time is

$$\begin{aligned} T_c &\geq t_{pcq} + 3t_{pd} + t_{\text{setup}} + t_{\text{skew}} \\ &= 80 + 3 \times 40 + 50 + 50 = 300\,\text{ps} \end{aligned} \tag{3.23}$$

The maximum clock frequency is $f_c = 1/T_c = 3.33$ GHz.

The short path also remains the same at 55 ps. The hold time is effectively increased by the skew to 60 + 50 = 110 ps, which is much greater than 55 ps. Hence, the circuit will violate the hold time and malfunction at any frequency. The circuit violated the hold time constraint even without skew. Skew in the system just makes the violation worse.

Example 3.13 FIXING HOLD TIME VIOLATIONS

Revisit Example 3.11 and assume that the system has 50 ps of clock skew.

Solution The critical path is unaffected, so the maximum clock frequency remains 3.33 GHz.

The short path increases to 80 ps. This is still less than $t_{\text{hold}} + t_{\text{skew}} = 110$ ps, so the circuit still violates its hold time constraint.

To fix the problem, even more buffers could be inserted. Buffers would need to be added on the critical path as well, reducing the clock frequency. Alternatively, a better flip-flop with a shorter hold time might be used.

3.5.4 Metastability

As noted earlier, it is not always possible to guarantee that the input to a sequential circuit is stable during the aperture time, especially when the input arrives from the external world. Consider a button connected to the input of a flip-flop, as shown in Figure 3.50. When the button is not pressed, $D = 0$. When the button is pressed, $D = 1$. A monkey presses the button at some random time relative to the rising edge of *CLK*. We want to know the output *Q* after the rising edge of *CLK*. In Case I, when the button is pressed much before *CLK*, $Q = 1$. In Case II, when the button is not pressed until long after *CLK*, $Q = 0$. But in Case III, when the button is pressed sometime between t_{setup} before *CLK* and t_{hold} after *CLK*, the input violates the dynamic discipline and the output is undefined.

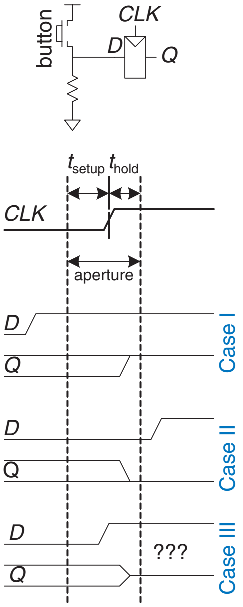


Figure 3.50 Input changing before, after, or during aperture

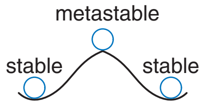


Figure 3.51 Stable and metastable states

Metastable State

When a flip-flop samples an input that is changing during its aperture, the output *Q* may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. This is called a *metastable* state. Eventually, the flip-flop will resolve the output to a *stable state* of either 0 or 1. However, the *resolution time* required to reach the stable state is unbounded.

The metastable state of a flip-flop is analogous to a ball on the summit of a hill between two valleys, as shown in Figure 3.51. The two valleys are stable states, because a ball in the valley will remain there as long as it is not disturbed. The top of the hill is called metastable because the ball would remain there if it were perfectly balanced. But because nothing is perfect, the ball will eventually roll to one side or the other. The time required for this change to occur depends on how nearly well balanced the ball originally was. Every bistable device has a metastable state between the two stable states.

Resolution Time

If a flip-flop input changes at a random time during the clock cycle, the resolution time, t_{res}, required to resolve to a stable state is also a random variable. If the input changes outside the aperture, then $t_{res} = t_{pcq}$. But if the input happens to change within the aperture, t_{res} can be substantially longer. Theoretical and experimental analyses (see Section 3.5.6) have shown that the probability that the resolution time, t_{res}, exceeds some arbitrary time, t, decreases exponentially with t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3.24}$$

where T_c is the clock period, and T_0 and τ are characteristic of the flip-flop. The equation is valid only for t substantially longer than t_{pcq}.

Intuitively, T_0/T_c describes the probability that the input changes at a bad time (i.e., during the aperture time); this probability decreases with the cycle time, T_c. τ is a time constant indicating how fast the flip-flop moves away from the metastable state; it is related to the delay through the cross-coupled gates in the flip-flop.

In summary, if the input to a bistable device such as a flip-flop changes during the aperture time, the output may take on a metastable value for some time before resolving to a stable 0 or 1. The amount of time required to resolve is unbounded because for any finite time, t, the probability that the flip-flop is still metastable is nonzero. However, this probability drops off exponentially as t increases. Therefore, if we wait long enough, much longer than t_{pcq}, we can expect with exceedingly high probability that the flip-flop will reach a valid logic level.

3.5.5 Synchronizers

Asynchronous inputs to digital systems from the real world are inevitable. Human input is asynchronous, for example. If handled carelessly, these asynchronous inputs can lead to metastable voltages within the system, causing erratic system failures that are extremely difficult to track down and correct. The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small. "Sufficiently" depends on the context. For a cell phone, perhaps one failure in 10 years is acceptable, because the user can always turn the phone off and back on if it locks up. For a medical device, one failure in the expected life of the universe (10^{10} years) is a better target. To guarantee good logic levels, all asynchronous inputs should be passed through *synchronizers*.

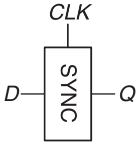


Figure 3.52 Synchronizer symbol

A synchronizer, shown in Figure 3.52, is a device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time; the output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value, but must not be metastable.

Figure 3.53 shows a simple way to build a synchronizer out of two flip-flops. F1 samples D on the rising edge of CLK. If D is changing at that time, the output D2 may be momentarily metastable. If the clock

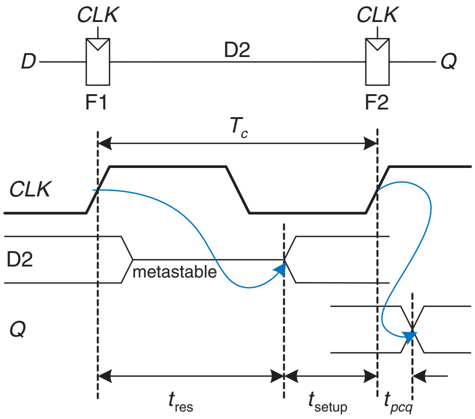


Figure 3.53 Simple synchronizer

period is long enough, D2 will, with high probability, resolve to a valid logic level before the end of the period. F2 then samples D2, which is now stable, producing a good output Q.

We say that a synchronizer *fails* if Q, the output of the synchronizer, becomes metastable. This may happen if $D2$ has not resolved to a valid level by the time it must set up at F2—that is, if $t_{res} > T_c - t_{\text{setup}}$. According to Equation 3.24, the probability of failure for a single input change at a random time is

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.25}$$

The probability of failure, P(failure), is the probability that the output Q will be metastable upon a single change in D. If D changes once per second, the probability of failure per second is just P(failure). However, if D changes N times per second, the probability of failure per second is N times as great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.26}$$

System reliability is usually measured in *mean time between failures* (*MTBF*). As the name suggests, MTBF is the average amount of time between failures of the system. It is the reciprocal of the probability that the system will fail in any given second

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{\frac{T_c - t_{\text{setup}}}{\tau}}}{NT_0} \tag{3.27}$$

Equation 3.27 shows that the MTBF improves exponentially as the synchronizer waits for a longer time, T_c. For most systems, a synchronizer that waits for one clock cycle provides a safe MTBF. In exceptionally high-speed systems, waiting for more cycles may be necessary.

Example 3.14 SYNCHRONIZER FOR FSM INPUT

The traffic light controller FSM from Section 3.4.1 receives asynchronous inputs from the traffic sensors. Suppose that a synchronizer is used to guarantee stable inputs to the controller. Traffic arrives on average 0.2 times per second. The flip-flops in the synchronizer have the following characteristics: $\tau = 200$ ps, $T_0 = 150$ ps, and $t_{\text{setup}} = 500$ ps. How long must the synchronizer clock period be for the MTBF to exceed 1 year?

Solution 1 year $\approx \pi \times 10^7$ seconds. Solve Equation 3.27.

$$\pi \times 10^7 = \frac{T_c e^{\frac{T_c - 500\times10^{-12}}{200\times10^{-12}}}}{(0.2)(150 \times 10^{-12})} \tag{3.28}$$

This equation has no closed form solution. However, it is easy enough to solve by guess and check. In a spreadsheet, try a few values of T_c and calculate the MTBF until discovering the value of T_c that gives an MTBF of 1 year: $T_c = 3.036$ ns.

3.5.6 Derivation of Resolution Time*

Equation 3.24 can be derived using a basic knowledge of circuit theory, differential equations, and probability. This section can be skipped if you are not interested in the derivation or if you are unfamiliar with the mathematics.

A flip-flop output will be metastable after some time, t, if the flip-flop samples a changing input (causing a metastable condition) and the output does not resolve to a valid level within that time after the clock edge. Symbolically, this can be expressed as

$$P(t_{res} > t) = P(\text{samples changing input}) \times P(\text{unresolved}) \tag{3.29}$$

We consider each probability term individually. The asynchronous input signal switches between 0 and 1 in some time, t_{switch}, as shown in Figure 3.54. The probability that the input changes during the aperture around the clock edge is

$$P(\text{samples changing input}) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{T_c} \tag{3.30}$$

If the flip-flop does enter metastability—that is, with probability P(samples changing input)—the time to resolve from metastability depends on the inner workings of the circuit. This resolution time determines P(unresolved), the probability that the flip-flop has not yet

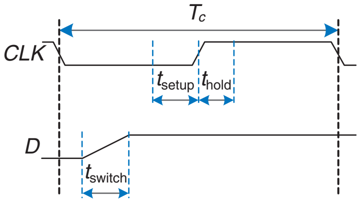


Figure 3.54 Input timing

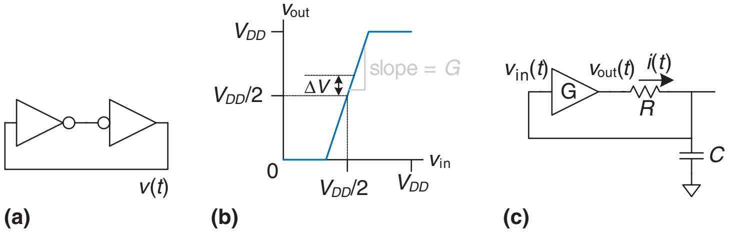


Figure 3.55 Circuit model of bistable device

resolved to a valid logic level after a time *t*. The remainder of this section analyzes a simple model of a bistable device to estimate this probability.

A bistable device uses storage with positive feedback. Figure 3.55(a) shows this feedback implemented with a pair of inverters; this circuit's behavior is representative of most bistable elements. A pair of inverters behaves like a buffer. Let us model the buffer as having the symmetric DC transfer characteristics shown in Figure 3.55(b), with a slope of *G*. The buffer can deliver only a finite amount of output current; we can model this as an output resistance, *R*. All real circuits also have some capacitance C that must be charged up. Charging the capacitor through the resistor causes an RC delay, preventing the buffer from switching instantaneously. Hence, the complete circuit model is shown in Figure 3.55(c), where $v_{out}(t)$ is the voltage of interest conveying the state of the bistable device.

The metastable point for this circuit is $v_{out}(t) = v_{in}(t) = V_{DD}/2$; if the circuit began at exactly that point, it would remain there indefinitely in the absence of noise. Because voltages are continuous variables, the chance that the circuit will begin at exactly the metastable point is vanishingly small. However, the circuit might begin at time 0 near metastability at $v_{out}(0) = V_{DD}/2 + \Delta V$ for some small offset ΔV. In such a case, the positive feedback will eventually drive $v_{out}(t)$ to V_{DD} if $\Delta V > 0$ and to 0 if $\Delta V < 0$. The time required to reach V_{DD} or 0 is the resolution time of the bistable device.

The DC transfer characteristic is nonlinear, but it appears linear near the metastable point, which is the region of interest to us. Specifically, if $v_{in}(t) = V_{DD}/2 + \Delta V/G$, then $v_{out}(t) = V_{DD}/2 + \Delta V$ for small ΔV. The current through the resistor is $i(t) = (v_{out}(t) - v_{in}(t))/R$.

The capacitor charges at a rate $dv_{\text{in}}(t)/dt = i(t)/C$. Putting these facts together, we find the governing equation for the output voltage.

$$\frac{dv_{\text{out}}(t)}{dt} = \frac{(G-1)}{RC}\left[v_{\text{out}}(t) - \frac{V_{DD}}{2}\right] \tag{3.31}$$

This is a linear first-order differential equation. Solving it with the initial condition $v_{\text{out}}(0) = V_{DD}/2 + \Delta V$ gives

$$v_{\text{out}}(t) = \frac{V_{DD}}{2} + \Delta V e^{\frac{(G-1)t}{RC}} \tag{3.32}$$

Figure 3.56 plots trajectories for $v_{\text{out}}(t)$, given various starting points. $v_{\text{out}}(t)$ moves exponentially away from the metastable point $V_{DD}/2$ until it saturates at V_{DD} or 0. The output eventually resolves to 1 or 0. The amount of time this takes depends on the initial voltage offset (ΔV) from the metastable point ($V_{DD}/2$).

Solving Equation 3.32 for the resolution time t_{res}, such that $v_{\text{out}}(t_{res}) = V_{DD}$ or 0, gives

$$|\Delta V| e^{\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2} \tag{3.33}$$

$$t_{res} = \frac{RC}{G-1}\ln\frac{V_{DD}}{2|\Delta V|} \tag{3.34}$$

In summary, the resolution time increases if the bistable device has high resistance or capacitance that causes the output to change slowly. It decreases if the bistable device has high *gain*, G. The resolution time also increases logarithmically as the circuit starts closer to the metastable point ($\Delta V \rightarrow 0$).

Define τ as $\frac{RC}{G-1}$. Solving Equation 3.34 for ΔV finds the initial offset, ΔV_{res}, that gives a particular resolution time, t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-t_{res}/\tau} \tag{3.35}$$

Suppose that the bistable device samples the input while it is changing. It measures a voltage, $v_{\text{in}}(0)$, which we will assume is uniformly distributed between 0 and V_{DD}. The probability that the output has not

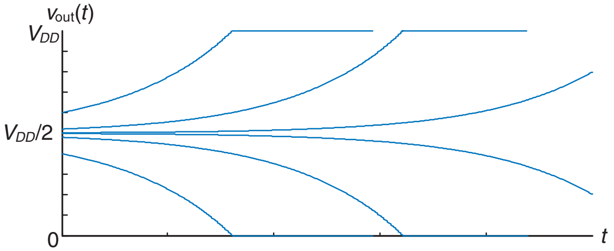


Figure 3.56 Resolution trajectories

resolved to a legal value after time t_{res} depends on the probability that the initial offset is sufficiently small. Specifically, the initial offset on v_{out} must be less than ΔV_{res}, so the initial offset on v_{in} must be less than $\Delta V_{res}/G$. Then, the probability that the bistable device samples the input at a time to obtain a sufficiently small initial offset is

$$P(\text{unresolved}) = P\left(\left|v_{in}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{GV_{DD}} \tag{3.36}$$

Putting this all together, the probability that the resolution time exceeds some time t is given by the following equation:

$$P(t_{res} > t) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{GT_c} e^{-\frac{t}{\tau}} \tag{3.37}$$

Observe that Equation 3.37 is in the form of Equation 3.24, where $T_0 = (t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}})/G$ and $\tau = RC/(G - 1)$. In summary, we have derived Equation 3.24 and shown how T_0 and τ depend on physical properties of the bistable device.

3.6 PARALLELISM

The speed of a system is characterized by the latency and throughput of information moving through it. We define a *token* to be a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualize data moving through the circuit. The *latency* of a system is the time required for one token to pass through the system from start to end. The *throughput* is the number of tokens that can be produced per unit time.

Example 3.15 COOKIE THROUGHPUT AND LATENCY

Ben Bitdiddle is throwing a milk and cookies party to celebrate the installation of his traffic light controller. It takes him 5 minutes to roll cookies and place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

Solution In this example, a tray of cookies is a token. The latency is 1/3 hour per tray. The throughput is 3 trays/hour.

As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism*, which comes in two forms: spatial and temporal. With *spatial parallelism*, multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With *temporal parallelism*, a task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time, so multiple tasks can overlap. Temporal parallelism is commonly called *pipelining*. Spatial parallelism is sometimes just called *parallelism*, but we will avoid that naming convention because it is ambiguous.

Example 3.16 COOKIE PARALLELISM

Ben Bitdiddle has hundreds of friends coming to his party and needs to bake cookies faster. He is considering using spatial and/or temporal parallelism.

Spatial Parallelism: Ben asks Alyssa P. Hacker to help out. She has her own cookie tray and oven.

Temporal Parallelism: Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

Solution The latency is the time required to complete one task from start to finish. In all cases, the latency is 1/3 hour. If Ben starts with no cookies, the latency is the time he has to wait until he gets to eat the first cookie.

The throughput is the number of cookie trays per hour. With spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes. Hence, the throughput doubles, to 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, for a throughput of 4 trays/hour. These are illustrated in Figure 3.57.

If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

Consider a task with latency L. In a system with no parallelism, the throughput is $1/L$. In a spatially parallel system with N copies of the hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken into N steps, or stages, of equal length. In such a case, the throughput is also N/L, and only one copy of the hardware is required. However, as the cookie example showed, finding N steps of equal length is often impractical. If the longest step has a latency L_1, the pipelined throughput is $1/L_1$.

Pipelining (temporal parallelism) is particularly attractive because it speeds up a circuit without duplicating the hardware. Instead, registers are placed between blocks of combinational logic to divide the logic into

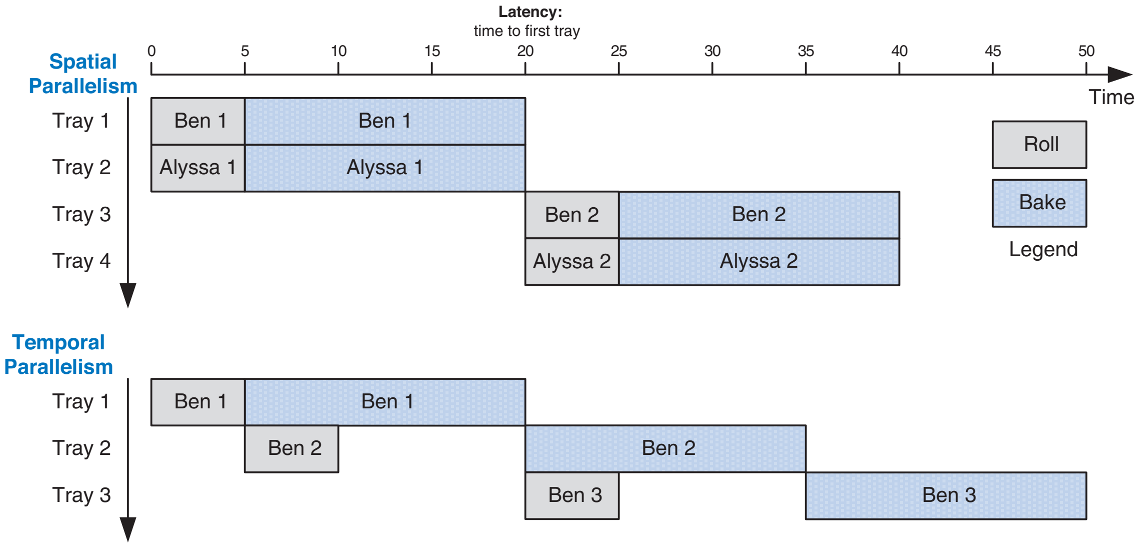


Figure 3.57 Spatial and temporal parallelism in the cookie kitchen

shorter stages that can run with a faster clock. The registers prevent a token in one pipeline stage from catching up with and corrupting the token in the next stage.

Figure 3.58 shows an example of a circuit with no pipelining. It contains four blocks of logic between the registers. The critical path passes through blocks 2, 3, and 4. Assume that the register has a clock-to-Q propagation delay of 0.3 ns and a setup time of 0.2 ns. Then, the cycle time is $T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5$ ns. The circuit has a latency of 9.5 ns and a throughput of 1/9.5 ns = 105 MHz.

Figure 3.59 shows the same circuit partitioned into a two-stage pipeline by adding a register between blocks 3 and 4. The first stage has a minimum clock period of $0.3 + 3 + 2 + 0.2 = 5.5$ ns. The second stage has a minimum clock period of $0.3 + 4 + 0.2 = 4.5$ ns. The clock must be slow enough for all stages to work. Hence, $T_c = 5.5$ ns. The latency is

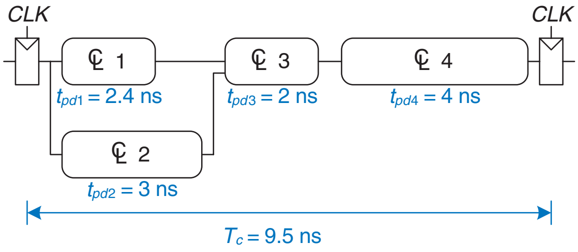


Figure 3.58 Circuit with no pipelining

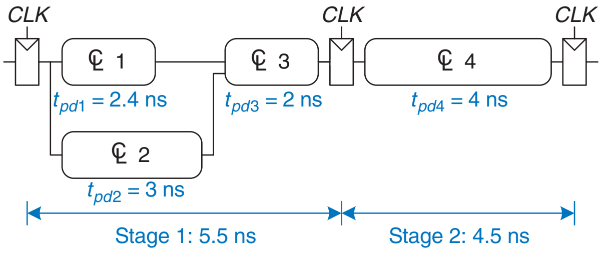


Figure 3.59 Circuit with two-stage pipeline

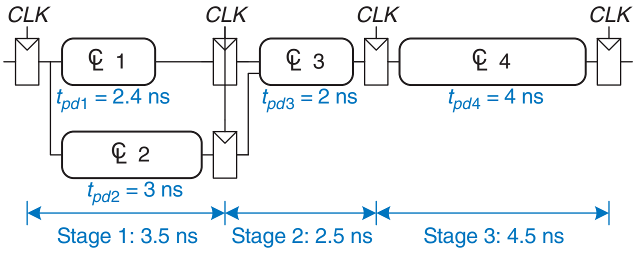


Figure 3.60 Circuit with three-stage pipeline

two clock cycles, or 11 ns. The throughput is 1/5.5 ns = 182 MHz. This example shows that, in a real circuit, pipelining with two stages almost doubles the throughput and slightly increases the latency. In comparison, ideal pipelining would exactly double the throughput at no penalty in latency. The discrepancy comes about because the circuit cannot be divided into two exactly equal halves and because the registers introduce more sequencing overhead.

Figure 3.60 shows the same circuit partitioned into a three-stage pipeline. Note that two more registers are needed to store the results of blocks 1 and 2 at the end of the first pipeline stage. The cycle time is now limited by the third stage to 4.5 ns. The latency is three cycles, or 13.5 ns. The throughput is 1/4.5 ns = 222 MHz. Again, adding a pipeline stage improves throughput at the expense of some latency.

Although these techniques are powerful, they do not apply to all situations. The bane of parallelism is *dependencies*. If a current task is dependent on the result of a prior task, rather than just prior steps in the current task, the task cannot start until the prior task has completed. For example, if Ben wants to check that the first tray of cookies tastes good before he starts preparing the second, he has a dependency that prevents pipelining or parallel operation. Parallelism is one of the most important techniques for designing high-performance digital systems. Chapter 7 discusses pipelining further and shows examples of handling dependencies.

3.7 SUMMARY

This chapter has described the analysis and design of sequential logic. In contrast to combinational logic, whose outputs depend only on the current inputs, sequential logic outputs depend on both current and prior inputs. In other words, sequential logic remembers information about prior inputs. This memory is called the *state of the logic*.

Anyone who could invent logic whose outputs depend on future inputs would be fabulously wealthy!

Sequential circuits can be difficult to analyze and are easy to design incorrectly, so we limit ourselves to a small set of carefully designed building blocks. The most important element for our purposes is the flip-flop, which receives a clock and an input D and produces an output Q. The flip-flop copies D to Q on the rising edge of the clock and otherwise remembers the old state of Q. A group of flip-flops sharing a common clock is called a *register*. Flip-flops may also receive reset or enable control signals.

Although many forms of sequential logic exist, we discipline ourselves to use synchronous sequential circuits because they are easy to design. Synchronous sequential circuits consist of blocks of combinational logic separated by clocked registers. The state of the circuit is stored in the registers and updated only on clock edges.

Finite state machines are a powerful technique for designing sequential circuits. To design an FSM, first identify the inputs and outputs of the machine and sketch a state transition diagram, indicating the states and the transitions between them. Select an encoding for the states, and rewrite the diagram as a state transition table and output table, indicating the next state and output, given the current state and input. From these tables, design the combinational logic to compute the next state and output, and sketch the circuit.

Synchronous sequential circuits have a timing specification, including the clock-to-Q propagation and contamination delays, t_{pcq} and t_{ccq}, and the setup and hold times, t_{setup} and t_{hold}. For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock. The minimum cycle time T_c of the system is equal to the propagation delay t_{pd} through the combinational logic plus $t_{pcq} + t_{\text{setup}}$ of the register. For correct operation, the contamination delay through the register and combinational logic must be greater than t_{hold}. Despite the common misconception to the contrary, hold time does not affect the cycle time.

Overall system performance is measured in latency and throughput. The latency is the time required for a token to pass from start to end. The throughput is the number of tokens that the system can process per unit time. Parallelism improves system throughput.

Exercises

Exercise 3.1 Given the input waveforms shown in Figure 3.61, sketch the output, *Q*, of an SR latch.

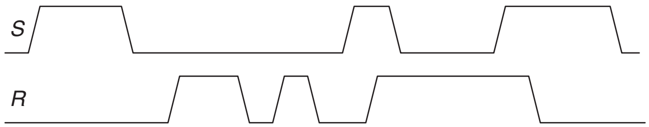


Figure 3.61 Input waveforms of SR latch for Exercise 3.1

Exercise 3.2 Given the input waveforms shown in Figure 3.62, sketch the output, *Q*, of an SR latch.

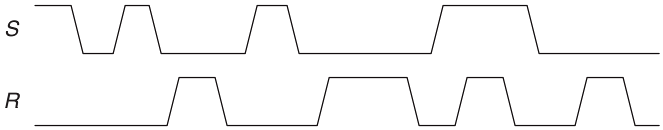


Figure 3.62 Input waveforms of SR latch for Exercise 3.2

Exercise 3.3 Given the input waveforms shown in Figure 3.63, sketch the output, *Q*, of a D latch.

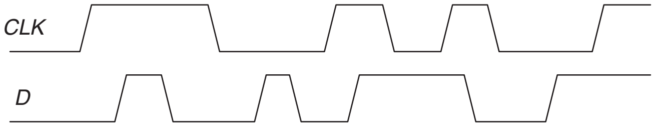


Figure 3.63 Input waveforms of D latch or flip-flop for Exercises 3.3 and 3.5

Exercise 3.4 Given the input waveforms shown in Figure 3.64, sketch the output, *Q*, of a D latch.

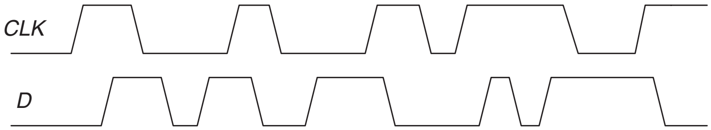


Figure 3.64 Input waveforms of D latch or flip-flop for Exercises 3.4 and 3.6

Exercise 3.5 Given the input waveforms shown in Figure 3.63, sketch the output, Q, of a D flip-flop.

Exercise 3.6 Given the input waveforms shown in Figure 3.64, sketch the output, Q, of a D flip-flop.

Exercise 3.7 Is the circuit in Figure 3.65 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

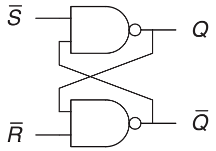


Figure 3.65 Mystery circuit

Exercise 3.8 Is the circuit in Figure 3.66 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

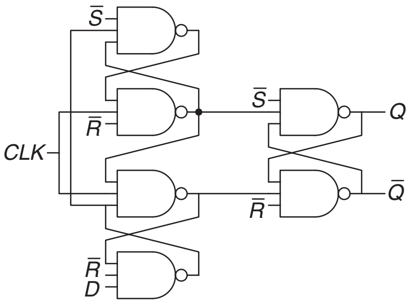


Figure 3.66 Mystery circuit

Exercise 3.9 The *toggle (T) flip-flop* has one input, *CLK*, and one output, Q. On each rising edge of *CLK*, Q toggles to the complement of its previous value. Draw a schematic for a T flip-flop using a D flip-flop and an inverter.

Exercise 3.10 *A JK flip-flop* receives a clock and two inputs, J and K. On the rising edge of the clock, it updates the output, Q. If J and K are both 0, Q retains its old value. If only J is 1, Q becomes 1. If only K is 1, Q becomes 0. If both J and K are 1, Q becomes the opposite of its present state.

(a) Construct a JK flip-flop, using a D flip-flop and some combinational logic.

(b) Construct a D flip-flop, using a JK flip-flop and some combinational logic.

(c) Construct a T flip-flop (see Exercise 3.9), using a JK flip-flop.

Exercise 3.11 The circuit in Figure 3.67 is called a *Muller C-element*. Explain in a simple fashion what the relationship is between the inputs and output.

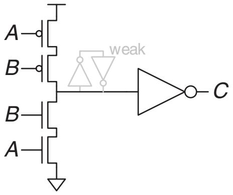


Figure 3.67 Muller C-element

Exercise 3.12 Design an asynchronously resettable D latch, using logic gates.

Exercise 3.13 Design an asynchronously resettable D flip-flop, using logic gates.

Exercise 3.14 Design a synchronously settable D flip-flop, using logic gates.

Exercise 3.15 Design an asynchronously settable D flip-flop, using logic gates.

Exercise 3.16 Suppose that a ring oscillator is built from N inverters connected in a loop. Each inverter has a minimum delay of t_{cd} and a maximum delay of t_{pd}. If N is odd, determine the range of frequencies at which the oscillator might operate.

Exercise 3.17 Why must N be odd in Exercise 3.16?

Exercise 3.18 Which of the circuits in Figure 3.68 are synchronous sequential circuits? Explain.

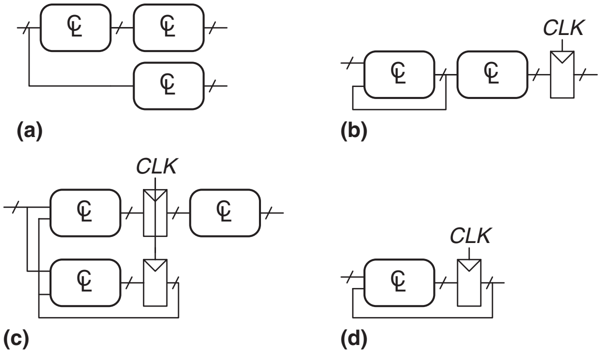


Figure 3.68 Circuits

Exercise 3.19 You are designing an elevator controller for a building with 25 floors. The controller has two inputs: *UP* and *DOWN*. It produces an output indicating the floor that the elevator is on. There is no floor 13. What is the minimum number of bits of state in the controller?

Exercise 3.20 You are designing an FSM to keep track of the mood of four students working in the digital design lab. Each student's mood is either HAPPY (the circuit works), SAD (the circuit blew up), BUSY (working on the circuit), CLUELESS (confused about the circuit), or ASLEEP (face down on the circuit board). How many states does the FSM have? What is the minimum number of bits necessary to represent these states?

Exercise 3.21 How would you factor the FSM from Exercise 3.20 into multiple simpler machines? How many states does each simpler machine have? What is the minimum total number of bits necessary in this factored design?

Exercise 3.22 Describe in words what the state machine in Figure 3.69 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

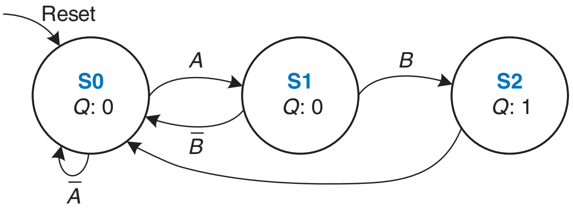


Figure 3.69 State transition diagram for Exercise 3.22

Exercise 3.23 Describe in words what the state machine in Figure 3.70 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

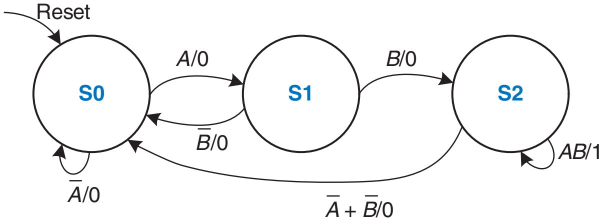


Figure 3.70 State transition diagram for Exercise 3.23

Exercise 3.24 Accidents are still occurring at the intersection of Academic Avenue and Bravado Boulevard. The football team is rushing into the intersection the moment light *B* turns green. They are colliding with sleep-deprived CS majors who stagger into the intersection just before light *A* turns red. Extend the traffic light controller from Section 3.4.1 so that both lights are red for 5 seconds before either light turns green again. Sketch your improved Moore machine state transition diagram, state encodings, state transition table, output table, next state and output equations, and your FSM schematic.

Exercise 3.25 Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 or the pattern 1110. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a combined state transition and output table, using your encodings. Write the next state and output equations and sketch your FSM schematic.

Exercise 3.26 You have been enlisted to design a soda machine dispenser for your department lounge. Sodas are partially subsidized by the student chapter of the IEEE, so they cost only 25 cents. The machine accepts nickels, dimes, and quarters. When enough coins have been inserted, it dispenses the soda and returns any necessary change. Design an FSM controller for the soda machine. The FSM inputs are *Nickel*, *Dime*, and *Quarter*, indicating which coin was inserted. Assume that exactly one coin is inserted on each cycle. The outputs are *Dispense*, *ReturnNickel*, *ReturnDime*, and *ReturnTwoDimes*. When the FSM reaches 25 cents, it asserts *Dispense* and the necessary *Return* outputs required to deliver the appropriate change. Then, it should be ready to start accepting coins for another soda.

Exercise 3.27 Gray codes have a useful property in that consecutive numbers differ in only a single bit position. Table 3.23 lists a 3-bit Gray code representing the numbers 0 to 7. Design a 3-bit modulo 8 Gray code counter FSM with no inputs and three outputs. (A modulo N counter counts from 0 to $N - 1$,

Table 3.23 3-bit Gray code

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

then repeats. For example, a watch uses a modulo 60 counter for the minutes and seconds that counts from 0 to 59.) When reset, the output should be 000. On each clock edge, the output should advance to the next Gray code. After reaching 100, it should repeat with 000.

Exercise 3.28 Extend your modulo 8 Gray code counter from Exercise 3.27 to be an UP/DOWN counter by adding an *UP* input. If *UP* = 1, the counter advances to the next number. If *UP* = 0, the counter retreats to the previous number.

Exercise 3.29 Your company, Detect-o-rama, would like to design an FSM that takes two inputs, *A* and *B*, and generates one output, *Z*. The output in cycle *n*, Z_n, is either the Boolean AND or OR of the corresponding input A_n and the previous input A_{n-1}, depending on the other input, B_n:

$$\begin{aligned} Z_n &= A_n A_{n-1} \quad &\text{if} \quad B_n &= 0 \\ Z_n &= A_n + A_{n-1} \quad &\text{if} \quad B_n &= 1 \end{aligned}$$

(a) Sketch the waveform for *Z*, given the inputs shown in Figure 3.71.

(b) Is this FSM a Moore or a Mealy machine?

(c) Design the FSM. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

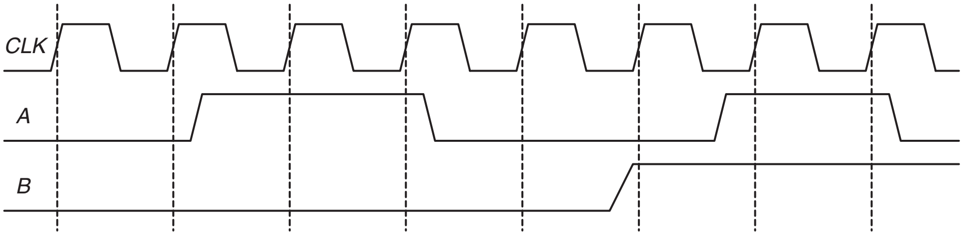


Figure 3.71 FSM input waveforms for Exercise 3.29

Exercise 3.30 Design an FSM with one input, *A*, and two outputs, *X* and *Y*. *X* should be 1 if *A* has been 1 for at least three cycles altogether (not necessarily consecutively). *Y* should be 1 if *A* has been 1 for at least two consecutive cycles. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

Exercise 3.31 Analyze the FSM shown in Figure 3.72. Write the state transition and output tables and sketch the state transition diagram.

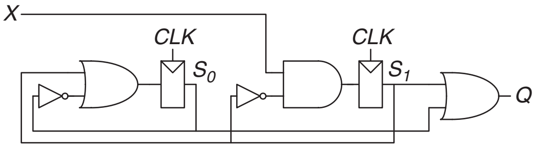


Figure 3.72 **FSM schematic for Exercise 3.31**

Exercise 3.32 Repeat Exercise 3.31 for the FSM shown in Figure 3.73. Recall that the *s* and *r* register inputs indicate set and reset, respectively.

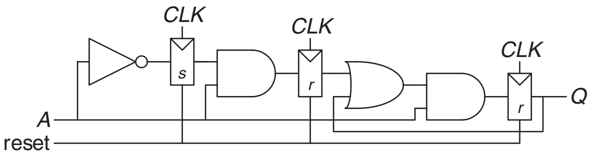


Figure 3.73 **FSM schematic for Exercise 3.32**

Exercise 3.33 Ben Bitdiddle has designed the circuit in Figure 3.74 to compute a registered four-input XOR function. Each two-input XOR gate has a propagation delay of 100 ps and a contamination delay of 55 ps. Each flip-flop has a setup time of 60 ps, a hold time of 20 ps, a clock-to-*Q* maximum delay of 70 ps, and a clock-to-*Q* minimum delay of 50 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 2 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

(d) Alyssa P. Hacker points out that she can redesign the combinational logic between the registers to be faster *and* tolerate more clock skew. Her improved circuit also uses three two-input XORs, but they are arranged differently. What is her circuit? What is its maximum frequency if there is no clock skew? How much clock skew can the circuit tolerate before it might experience a hold time violation?

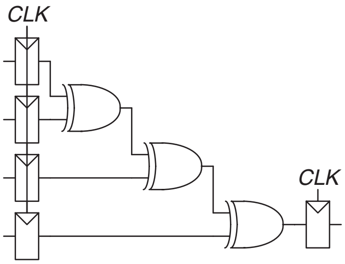


Figure 3.74 **Registered four-input XOR circuit for Exercise 3.33**

Exercise 3.34 You are designing an adder for the blindingly fast 2-bit RePentium Processor. The adder is built from two full adders, such that the carry out of the first adder is the carry in to the second adder, as shown in Figure 3.75. Your adder has input and output registers and must complete the addition in one clock cycle. Each full adder has the following propagation delays: 20 ps from C_{in} to C_{out} or to *Sum* (S), 25 ps from A or B to C_{out}, and 30 ps from A or B to S. The adder has a contamination delay of 15 ps from C_{in} to either output and 22 ps from A or B to either output. Each flip-flop has a setup time of 30 ps, a hold time of 10 ps, a clock-to-Q propagation delay of 35 ps, and a clock-to-Q contamination delay of 21 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 8 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

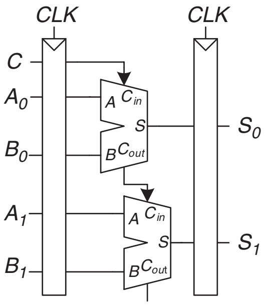


Figure 3.75 **2-bit adder schematic for Exercise 3.34**

Exercise 3.35 A *field programmable gate array* (*FPGA*) uses *configurable logic blocks* (*CLBs*) rather than logic gates to implement combinational logic. The Xilinx Spartan 3 FPGA has propagation and contamination delays of 0.61 and 0.30 ns, respectively, for each CLB. It also contains flip-flops with propagation and contamination delays of 0.72 and 0.50 ns, and setup and hold times of 0.53 and 0 ns, respectively.

(a) If you are building a system that needs to run at 40 MHz, how many consecutive CLBs can you use between two flip-flops? Assume that there is no clock skew and no delay through wires between CLBs.

(b) Suppose that all paths between flip-flops pass through at least one CLB. How much clock skew can the FPGA have without violating the hold time?

Exercise 3.36 A synchronizer is built from a pair of flip-flops with $t_{setup} = 50$ ps, $T_0 = 20$ ps, and $\tau = 30$ ps. It samples an asynchronous input that changes 10^8 times per second. What is the minimum clock period of the synchronizer to achieve a mean time between failures (MTBF) of 100 years?

Exercise 3.37 You would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 50 years. Your system is running at 1 GHz, and you use sampling flip-flops with $\tau = 100$ ps, $T_0 = 110$ ps, and $t_{\text{setup}} = 70$ ps. The synchronizer receives a new asynchronous input on average 0.5 times per second (i.e., once every 2 seconds). What is the required probability of failure to satisfy this MTBF? How many clock cycles would you have to wait before reading the sampled input signal to give that probability of error?

Exercise 3.38 You are walking down the hallway when you run into your lab partner walking in the other direction. The two of you first step one way and are still in each other's way. Then, you both step the other way and are still in each other's way. Then you both wait a bit, hoping the other person will step aside. You can model this situation as a metastable point and apply the same theory that has been applied to synchronizers and flip-flops. Suppose you create a mathematical model for yourself and your lab partner. You start the unfortunate encounter in the metastable state. The probability that you remain in this state after t seconds is $e^{-\frac{t}{\tau}}$. τ indicates your response rate; today, your brain has been blurred by lack of sleep and has $\tau = 20$ seconds.

(a) How long will it be until you have 99% certainty that you will have resolved from metastability (i.e., figured out how to pass one another)?

(b) You are not only sleepy, but also ravenously hungry. In fact, you will starve to death if you don't get going to the cafeteria within 3 minutes. What is the probability that your lab partner will have to drag you to the morgue?

Exercise 3.39 You have built a synchronizer using flip-flops with $T_0 = 20$ ps and $\tau = 30$ ps. Your boss tells you that you need to increase the MTBF by a factor of 10. By how much do you need to increase the clock period?

Exercise 3.40 Ben Bitdiddle invents a new and improved synchronizer in Figure 3.76 that he claims eliminates metastability in a single cycle. He explains that the circuit in box M is an analog "metastability detector" that produces a HIGH output if the input voltage is in the forbidden zone between V_{IL} and V_{IH}. The metastability detector checks to determine whether the first flip-flop has produced a metastable output on $D2$. If so, it asynchronously resets the flip-flop to produce a good 0 at $D2$. The second flip-flop then samples $D2$, always producing a valid logic level on Q. Alyssa P. Hacker tells Ben that there must be a bug in the circuit, because eliminating metastability is just as impossible as building a perpetual motion machine. Who is right? Explain, showing Ben's error or showing why Alyssa is wrong.

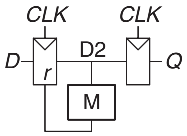


Figure 3.76 "New and improved" synchronizer for Exercise 3.40

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 3.1 Draw a state machine that can detect when it has received the serial input sequence 01010.

Question 3.2 Design a serial (one bit at a time) two's complementer FSM with two inputs, *Start* and *A*, and one output, *Q*. A binary number of arbitrary length is provided to input *A*, starting with the least significant bit. The corresponding bit of the output appears at *Q* on the same cycle. *Start* is asserted for one cycle to initialize the FSM before the least significant bit is provided.

Question 3.3 What is the difference between a latch and a flip-flop? Under what circumstances is each one preferable?

Question 3.4 Design a 5-bit counter finite state machine.

Question 3.5 Design an edge detector circuit. The output should go HIGH for one cycle after the input makes a 0 → 1 transition.

Question 3.6 Describe the concept of pipelining and why it is used.

Question 3.7 Describe what it means for a flip-flop to have a negative hold time.

Question 3.8 Given signal *A*, shown in Figure 3.77, design a circuit that produces signal *B*.

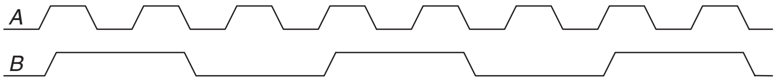


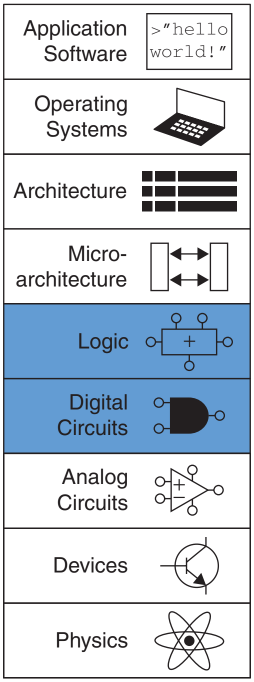
Figure 3.77 Signal waveforms for Question 3.8

Question 3.9 Consider a block of logic between two registers. Explain the timing constraints. If you add a buffer on the clock input of the receiver (the second flip-flop), does the setup time constraint get better or worse?

HDL Dictionary
verilog
VHDL
a & b
a ^ b
~(a+b)
a and b
a xor b
a nor b
Hardware
FSM
B

Hardware Description Languages

4



4.1 INTRODUCTION

Thus far, we have focused on designing combinational and sequential digital circuits at the schematic level. The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations and manual translation of finite state machines (FSMs) into gates. In the 1990's, designers discovered that they were far more productive if they worked at a higher level of abstraction, specifying just the logical function and allowing a *computer-aided design* (*CAD*) tool to produce the optimized gates. The specifications are generally given in a *hardware description language* (*HDL*). The two leading hardware description languages are *SystemVerilog* and *VHDL*.

SystemVerilog and VHDL are built on similar principles but have different syntax. Discussion of these languages in this chapter is divided into two columns for literal side-by-side comparison, with SystemVerilog on the left and VHDL on the right. When you read the chapter for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it.

Subsequent chapters show hardware in both schematic and HDL form. If you choose to skip this chapter and not learn one of the HDLs, you will still be able to master the principles of computer organization from the schematics. However, the vast majority of commercial systems are now built using HDLs rather than schematics. If you expect to do digital design at any point in your professional life, we urge you to learn one of the HDLs.

4.1.1 Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00004-0

Some people still use Verilog, the original incarnation of SystemVerilog. So, we provide a version of this chapter describing Verilog on the companion website (see the Preface).

behavioral and *structural*. Behavioral models describe what a module does. Structural models describe how a module is built from simpler pieces; it is an application of hierarchy. The SystemVerilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6, $y = \bar{a}\,\bar{b}\,\bar{c} + a\bar{b}\,\bar{c} + a\bar{b}c$. In both languages, the module is named `sillyfunction` and has three inputs, `a`, `b`, and `c`, and one output, `y`.

HDL Example 4.1 COMBINATIONAL LOGIC

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. ~ indicates NOT, & indicates AND, and | indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);
end;
```

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library` `use` clause will be discussed in Section 4.7.2. The `entity` declaration lists the module name and its inputs and outputs. The `architecture` body defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of `'0'` or `'1'` as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations between AND and OR, so Boolean equations should be parenthesized.

A module, as you might expect, is a good application of modularity. It has a well-defined interface, consisting of its inputs and outputs, and it performs a specific function. The particular way in which it is coded is unimportant to others that might use the module, as long as it performs its function.

4.1.2 Language Origins

Universities are almost evenly split on which of these languages is taught in a first course. Industry is trending toward SystemVerilog, but many companies still use VHDL, so many designers need to be fluent in both.

SystemVerilog	**VHDL**
Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990, under the control of Open Verilog International. The language became an IEEE standard[1] in 1995. The language was extended in 2005 to streamline idiosyncrasies and to better support modeling and verification of systems. These extensions have been merged into a single language standard, which is now called SystemVerilog (IEEE STD 1800-2009). SystemVerilog file names normally end in `.sv`.	VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is, in turn, an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense. VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis. The IEEE standardized it in 1987 and has updated the standard several times since. This chapter is based on the 2008 revision of the VHDL standard (IEEE STD 1076-2008), which streamlines the language in a variety of ways. To use VHDL 2008 in ModelSim, you may need to set VHDL93 = 2008 in the modelsim.ini configuration file. VHDL file names normally end in `.vhd`.

Compared to SystemVerilog, VHDL is more verbose and cumbersome, as you might expect of a language developed by committee.

Both languages are fully capable of describing any hardware system, and both have their quirks. The best language to use is the one that is already being used at your site or the one that your customers demand. Most CAD tools today allow the two languages to be mixed so that different modules can be described in different languages.

4.1.3 Simulation and Synthesis

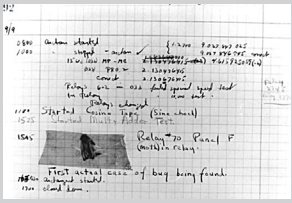
The two major purposes of HDLs are logic *simulation* and *synthesis*. During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Testing a system in the laboratory is time-consuming. Discovering the cause of errors in the lab can be extremely difficult because only signals routed to the chip pins can be observed.

The term "bug" predates the invention of the computer. Thomas Edison called the "little faults and difficulties" with his inventions "bugs" in 1878.

The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electromechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment "first actual case of bug being found."



92

9/9

0800 Anctan started
1000 stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 995 const
13" UC (032) MP - MC ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

Source: Notebook entry courtesy Naval Historical Center, US Navy; photo No. NII 96566-KN

[1] The Institute of Electrical and Electronics Engineers (IEEE) is a professional society responsible for many computing standards, including Wi-Fi (802.11), Ethernet (802.3), and floating-point numbers (754).

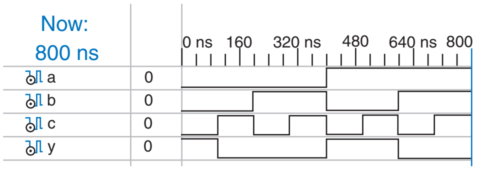


Figure 4.1 Simulation waveforms

There is no way to directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example, correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped at a total cost of $475 million in 1984. Logic simulation is essential to test a system before it is built.

Figure 4.1 shows waveforms from a simulation[2] of the previous `sillyfunction` module, demonstrating that the module works correctly. `y` is TRUE when `a`, `b`, and `c` are 000, 100, or 101, as specified by the Boolean equation.

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit. Figure 4.2 shows the results of synthesizing the `sillyfunction` module.[3] Notice how the three three-input AND gates are simplified into two two-input AND gates, as we discovered in Example 2.6 using Boolean algebra.

The synthesis tool labels each of the synthesized gates. In Figure 4.2, they are un5_y, un8_y, and y.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. SystemVerilog and VHDL are rich languages with many commands.

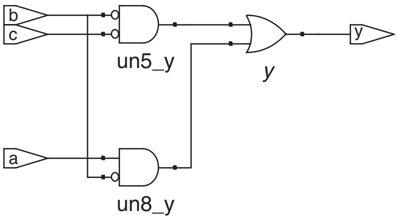


Figure 4.2 Synthesized circuit

[2] The simulations in this chapter were performed with the ModelSim PE Student Edition Version 10.3c. ModelSim was selected because it is used commercially, yet a student version with a capacity of 10,000 lines of code is freely available.

[3] Throughout this chapter, synthesis was performed using Synplify Premier from Synopsys. However, many synthesis tools exist, such as those included in Vivado and Quartus, the freely available Xilinx and Intel design tools for synthesizing HDL to field programmable gate arrays (see Section 5.6.2).

Not all of these commands can be synthesized into hardware. For example, a command to print results on the screen during simulation does not translate into hardware. Because our primary interest is to build hardware, we will emphasize a *synthesizable subset* of the languages. Specifically, we will divide HDL code into *synthesizable* modules and a *testbench*. The synthesizable modules describe the hardware. The testbench contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies between expected and actual outputs. Testbench code is intended only for simulation and cannot be synthesized.

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

In our experience, the best way to learn an HDL is by example. HDLs have specific ways of describing various classes of logic; these ways are called *idioms*. This chapter will teach you how to write the proper HDL idioms for each type of block and then how to put the blocks together to produce a working system. When you need to describe a particular kind of hardware, look for a similar example and adapt it to your purpose. We do not attempt to rigorously define all the syntax of the HDLs, because that is deathly boring and it tends to encourage thinking of HDLs as programming languages, not shorthand for hardware. The IEEE SystemVerilog and VHDL specifications, and numerous dry but exhaustive textbooks, contain all of the details, should you find yourself needing more information on a particular topic. (See the Further Readings section at the back of the book.)

4.2 COMBINATIONAL LOGIC

Recall that we are disciplining ourselves to design synchronous sequential circuits, which consist of combinational logic and registers. The outputs of combinational logic depend only on the current inputs. This section describes how to write behavioral models of combinational logic with HDLs.

4.2.1 Bitwise Operators

Bitwise operators act on single-bit signals or on multibit busses. For example, the `inv` module in HDL Example 4.2 describes four inverters connected to 4-bit busses.

HDL Example 4.2 INVERTERS

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

a[3:0] represents a 4-bit bus. The bits, from most significant to least significant, are a[3], a[2], a[1], and a[0]. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus a[4:1], in which case a[4] would have been the most significant. Or we could have used a[0:3], in which case the bits, from most significant to least significant, would be a[0], a[1], a[2], and a[3]. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses STD_LOGIC_VECTOR to indicate busses of STD_LOGIC. STD_LOGIC_VECTOR(3 downto 0) represents a 4-bit bus. The bits, from most significant to least significant, are a(3), a(2), a(1), and a(0). This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be STD_LOGIC_VECTOR(4 downto 1), in which case bit 4 would have been the most significant. Or we could have written STD_LOGIC_VECTOR(0 to 3), in which case the bits, from most significant to least significant, would be a(0), a(1), a(2), and a(3). This is called *big-endian* order.

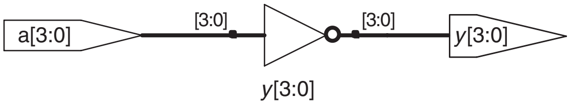


Figure 4.3 inv **synthesized circuit**

The endianness of a bus is purely arbitrary. (See the sidebar in Section 6.6.1 for the origin of the term.) Indeed, endianness is also irrelevant to this example because a bank of inverters doesn't care what the order of the bits are. Endianness matters only for operators, such as addition, where the sum of one column carries over into the next. Either ordering is acceptable as long as it is used consistently. We will consistently use the little-endian order, [N−1:0] in SystemVerilog and (N−1 downto 0) in VHDL, for an *N*-bit bus.

After each code example in this chapter is a schematic produced from the SystemVerilog code by the Synplify Premier synthesis tool. Figure 4.3 shows that the inv module synthesizes to a bank of four inverters, indicated by the inverter symbol labeled y[3:0]. The bank of inverters connects to 4-bit input and output busses. Similar hardware is produced from the synthesized VHDL code.

The gates module in HDL Example 4.3 demonstrates bitwise operations acting on 4-bit busses for other basic logic functions.

HDL Example 4.3 LOGIC GATES

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;      // AND
  assign y2 = a | b;      // OR
  assign y3 = a ^ b;      // XOR
  assign y4 = ~(a & b);   // NAND
  assign y5 = ~(a | b);   // NOR
endmodule
```

~, ^, and | are examples of SystemVerilog *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a & b, or ~(a | b), is called an *expression*. A complete command such as assign y4 = ~(a & b); is called a *statement*.

assign out = in1 op in2; is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Whenever the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- five different two-input logic gates
  -- acting on 4-bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

not, xor, and or are examples of VHDL *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a and b, or a nor b, is called an *expression*. A complete command such as y4 <= a nand b; is called a *statement*.

out <= in1 op in2; is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Whenever the inputs on the right side of the <= in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.

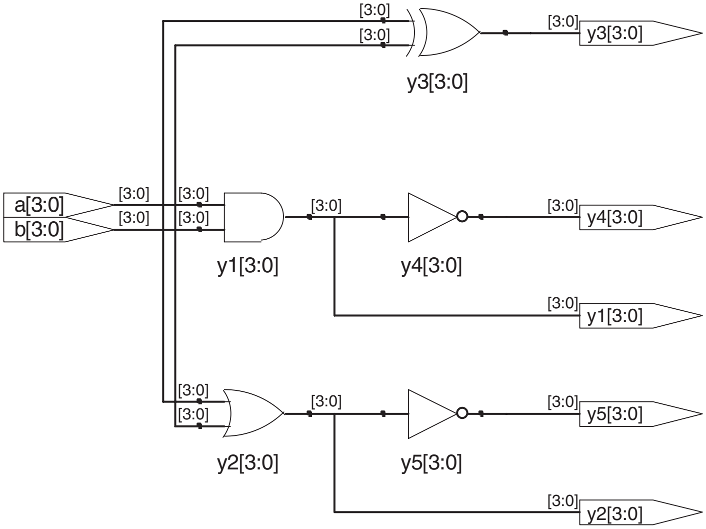


Figure 4.4 gates **synthesized circuit**

4.2.2 Comments and White Space

The `gates` example showed how to format comments. SystemVerilog and VHDL are not picky about the use of white space (i.e., spaces, tabs, and line breaks). Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable. Be consistent in your use of capitalization and underscores in signal and module names. This text uses all lower case. Module and signal names must not begin with a digit.

SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line.

SystemVerilog is case-sensitive. `y1` and `Y1` are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

VHDL

Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `--` continue to the end of the line.

VHDL is not case-sensitive. `y1` and `Y1` are the same signal in VHDL. However, other tools that may read your file might be case-sensitive, leading to nasty bugs if you blithely mix upper and lower case.

4.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. HDL Example 4.4 describes an eight-input AND gate with inputs a_7, $a_6, \ldots, a_0$. Analogous reduction operators exist for OR, XOR, NAND, NOR, and XNOR gates. Recall that a multiple-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

HDL Example 4.4 EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --      a(3) and a(2) and a(1) and a(0);
end;
```

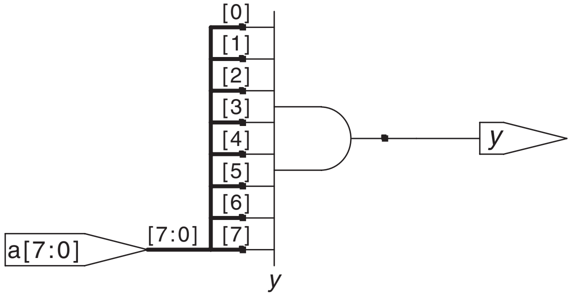


Figure 4.5 `and8` **synthesized circuit**

4.2.4 Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the *condition*. HDL Example 4.5 illustrates a 2:1 multiplexer using conditional assignment.

HDL Example 4.5 2:1 MULTIPLEXER

SystemVerilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y = d1`. If `s` is 0, then `y = d0`.

`?:` is also called a *ternary operator* because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations, depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
   port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
        s:      in  STD_LOGIC;
        y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The conditional signal assignment sets `y` to `d1` if `s` is 1. Otherwise, it sets `y` to `d0`. Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.

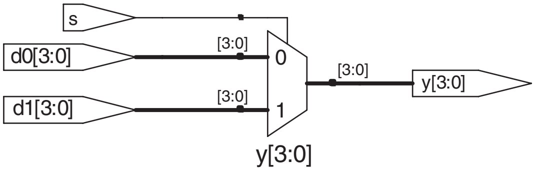


Figure 4.6 `mux2` **synthesized circuit**

HDL Example 4.6 shows a 4:1 multiplexer based on the same principle as the 2:1 multiplexer in HDL Example 4.5. Figure 4.7 shows the schematic for the 4:1 multiplexer produced by the synthesis tool. The software uses a different multiplexer symbol than this text has shown so far. The multiplexer has multiple data (d) and one-hot enable (e) inputs. When one of the enables is asserted, the associated data is passed to the output. For example, when s[1] = s[0] = 0, the bottom AND gate, un1_s_5, produces a 1, enabling the bottom input of the multiplexer and causing it to select d0[3:0].

HDL Example 4.6 4:1 MULTIPLEXER

SystemVerilog

A 4:1 multiplexer can select one of four inputs, using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If s[1] is 1, then the multiplexer chooses the first expression, (s[0] ? d3 : d2). This expression, in turn, chooses either d3 or d2 based on s[0] (y = d3 if s[0] is 1 and d2 if s[0] is 0). If s[1] is 0, then the multiplexer similarly chooses the second expression, which gives either d1 or d0 based on s[0].

VHDL

A 4:1 multiplexer can select one of four inputs, using multiple else clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
      d1 when s = "01" else
      d2 when s = "10" else
      d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a switch/case statement in place of multiple if/else statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

4.2.5 Internal Variables

Often it is convenient to break a complex function into intermediate steps. For example, a full adder, which will be described in Section 5.2.1,

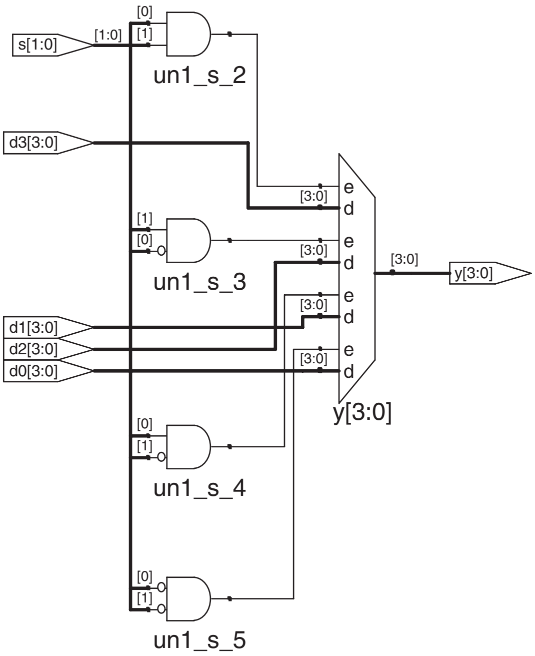


Figure 4.7 mux4 **synthesized circuit**

is a circuit with three inputs and two outputs defined by the following equations:

$$\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= AB + AC_{in} + BC_{in} \end{aligned} \qquad (4.1)$$

If we define intermediate signals, P and G,

$$\begin{aligned} P &= A \oplus B \\ G &= AB \end{aligned} \qquad (4.2)$$

we can rewrite the full adder as follows:

$$\begin{aligned} S &= P \oplus C_{in} \\ C_{out} &= G + PC_{in} \end{aligned} \qquad (4.3)$$

Check this by filling out the truth table to convince yourself it is correct.

P and G are called *internal variables* because they are neither inputs nor outputs; rather, they are used only internal to the module. They are similar to local variables in programming languages. HDL Example 4.7 shows how they are used in HDLs.

HDL assignment statements (`assign` in SystemVerilog and <= in VHDL) take place concurrently. This is different from conventional programming languages, such as C or Java, in which statements are evaluated in the order in which they are written. In a conventional language, it is

HDL Example 4.7 FULL ADDER

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements*, such as `p <= a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

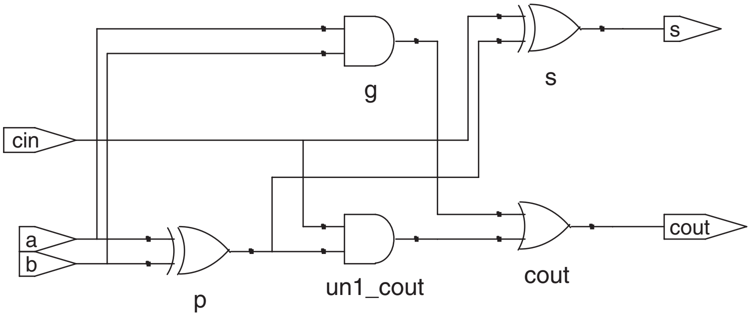


Figure 4.8 `fulladder` **synthesized circuit**

important that $S = P \oplus C_{in}$ comes after $P = A \oplus B$ because statements are executed sequentially. In an HDL, the order does not matter. Like hardware, HDL assignment statements are evaluated whenever the inputs, signals on the right-hand side, change their value regardless of the order in which the assignment statements appear in a module.

4.2.6 Precedence

Notice that we parenthesized the `cout` computation in HDL Example 4.7 to define the order of operations as $C_{out} = G + (P \cdot C_{in})$ rather than $C_{out} = (G + P) \cdot C_{in}$. If we had not used parentheses, the default operation order would be defined by the language. HDL Example 4.8

HDL Example 4.8 OPERATOR PRECEDENCE

SystemVerilog

Table 4.1 SystemVerilog operator precedence

	Op	Meaning
Highest	`~`	NOT
	`*, /, %`	MUL, DIV, MOD
	`+, -`	PLUS, MINUS
	`<<, >>`	Logical Left/Right Shift
	`<<<, >>>`	Arithmetic Left/Right Shift
	`<, <=, >, >=`	Relative Comparison
	`==, !=`	Equality Comparison
	`&, ~&`	AND, NAND
	`^, ~^`	XOR, XNOR
	`|, ~|`	OR, NOR
Lowest	`?:`	Conditional

The operator precedence for SystemVerilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses:

```
assign cout = g | p & cin;
```

VHDL

Table 4.2 VHDL operator precedence

	Op	Meaning
Highest	`not`	NOT
	`*, /, mod, rem`	MUL, DIV, MOD, REM
	`+, -`	PLUS, MINUS
	`rol, ror, srl, sll`	Rotate, Shift logical
	`<, <=, >, >=`	Relative Comparison
	`=, /=`	Equality Comparison
Lowest	`and, or, nand, nor, xor, xnor`	Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike SystemVerilog, all of the logical operations (`and`, `or`, etc.) have equal precedence, unlike what one might expect in Boolean algebra. Thus, parentheses are necessary. Otherwise, `cout <= g or p and cin` would be interpreted from left to right as `cout <= (g or p) and cin`.

specifies operator precedence from highest to lowest for each language. The tables include arithmetic, shift, and comparison operators that will be defined in Chapter 5.

4.2.7 Numbers

Numbers can be specified in binary, octal, decimal, or hexadecimal (bases 2, 8, 10, and 16, respectively). The size, that is, the number of bits, may be given optionally, and leading zeros are inserted to reach this size. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. HDL Example 4.9 explains how numbers are written in each language.

HDL Example 4.9 NUMBERS

SystemVerilog

The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal.

If the base is omitted, it defaults to decimal. If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms to fill all of the bits with 0 and 1, respectively.

Table 4.3 SystemVerilog numbers

Numbers	Bits	Base	Val	Stored
`3'b101`	3	2	5	101
`'b11`	?	2	3	000 … 0011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	00 … 0101010

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes: `'0'` and `'1'` indicate logic 0 and 1. The format for declaring STD_LOGIC_VECTOR constants is `NB"value"`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9X"25"` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 supports `B` for binary, `O` for octal, `D` for decimal, and `X` for hexadecimal.

If the base is omitted, it defaults to binary. If the size is not given, the number is assumed to have a size matching the number of bits specified in the value. For example, `y <= X"7B"` requires that `y` be an 8-bit signal. If `y` has more bits, VHDL does *not* pad the number with 0's on the left. Instead, an error occurs during compilation. `others => '0'` and `others => '1'` are VHDL idioms to fill all of the bits with 0 and 1, respectively.

Table 4.4 VHDL numbers

Numbers	Bits	Base	Val	Stored
`3B"101"`	3	2	5	101
`B"11"`	2	2	3	11
`8B"11"`	8	2	3	00000011
`8B"1010_1011"`	8	2	171	10101011
`3D"6"`	3	10	6	110
`6O"42"`	6	8	34	100010
`8X"AB"`	8	16	171	10101011
`"101"`	3	2	5	101
`B"101"`	3	2	5	101
`X"AB"`	8	16	171	10101011

4.2.8 Z's and X's

HDLs use `z` to indicate a floating value. `z` is particularly useful for describing a tristate buffer, whose output floats when the enable is 0. Recall from Section 2.6.2 that a bus can be driven by several tristate buffers, exactly one of which should be enabled. HDL Example 4.10 shows the idiom for a tristate buffer. If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (`z`).

Similarly, HDLs use `x` to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is `x`, indicating contention. If all of the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by `z`.

At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (`x` in SystemVerilog and `u` in VHDL). This is helpful to track errors caused by forgetting to reset a flip-flop before its output is used.

HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that `y` is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also, note that a `tri` output from a module can be used as a `logic` input to another module. Section 4.7 further discusses nets with multiple drivers.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= a when en else "ZZZZ";
end;
```

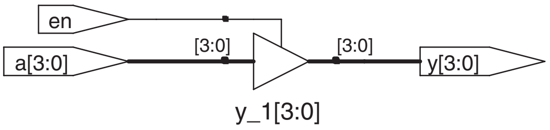


Figure 4.9 `tristate` **synthesized circuit**

If a gate receives a floating input, it may produce an `x` output when it can't determine the correct output value. Similarly, if it receives an illegal or uninitialized input, it may produce an `x` output. HDL Example 4.11

HDL Example 4.11 TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS

SystemVerilog

SystemVerilog signal values are `0`, `1`, `z`, and `x`. SystemVerilog constants starting with `z` or `x` are padded with leading `z`'s or `x`'s (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example, `0 & z` returns `0` because the output of an AND gate is always `0` if either input is `0`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in SystemVerilog.

VHDL

VHDL STD_LOGIC signals are `'0'`, `'1'`, `'z'`, `'x'`, and `'u'`.

Table 4.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, `'0' and 'z'` returns`'0'` because the output of an AND gate is always `'0'` if either input is`'0'`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `'x'` in VHDL. Uninitialized inputs cause uninitialized outputs, displayed as `'u'` in VHDL.

Table 4.5 SystemVerilog AND gate truth table with z and x

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

Table 4.6 VHDL AND gate truth table with z, x, and u

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

HDL Example 4.12 BIT SWIZZLING

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The {} operator is used to concatenate busses. {3{d[0]}} indicates three copies of d[0].

Don't confuse the 3-bit binary constant 3'b101 with a bus named b. Note that it was critical to specify the length of 3 bits in the constant. Otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y.

If y were wider than 9 bits, zeros would be placed in the most significant bits.

VHDL

```
y <= (c(2 downto 1), d(0), d(0), d(0), c(0), 3B"101");
```

The () aggregate operator is used to concatenate busses. y must be a 9-bit STD_LOGIC_VECTOR.

Another example demonstrates the power of VHDL aggregations. Assuming that z is an 8-bit STD_LOGIC_VECTOR, z is given the value 10010110 using the following command aggregation.

```
z <= ("10", 4 => '1', 2 downto 1 =>'1', others =>'0')
```

The "10" goes in the leading pair of bits. 1's are also placed into bit 4 and bits 2 and 1. The other bits are 0.

shows how SystemVerilog and VHDL combine these different signal values in logic gates.

Seeing x or u values in simulation is almost always an indication of a bug or bad coding practice. In the synthesized circuit, this corresponds to a floating gate input, uninitialized state, or contention. The x or u may be interpreted randomly by the circuit as 0 or 1, leading to unpredictable behavior.

4.2.9 Bit Swizzling

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as *bit swizzling*. In HDL Example 4.12, y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations.

4.2.10 Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results). These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13 adds delays to the original function from HDL Example 4.1, $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. It assumes that inverters have a delay of 1 ns, three-input AND gates have a delay of 2 ns, and three-input OR gates have a delay of 4 ns. Figure 4.10 shows the simulation waveforms, with y lagging 7 ns after the inputs. Note that y is initially unknown at the beginning of the simulation.

HDL Example 4.13 LOGIC GATES WITH DELAYS

SystemVerilog

```
'timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4  y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `'timescale unit/precision`. In this file, each unit is 1 ns and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as nonblocking (<=) and blocking (=) assignments, which will be discussed in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

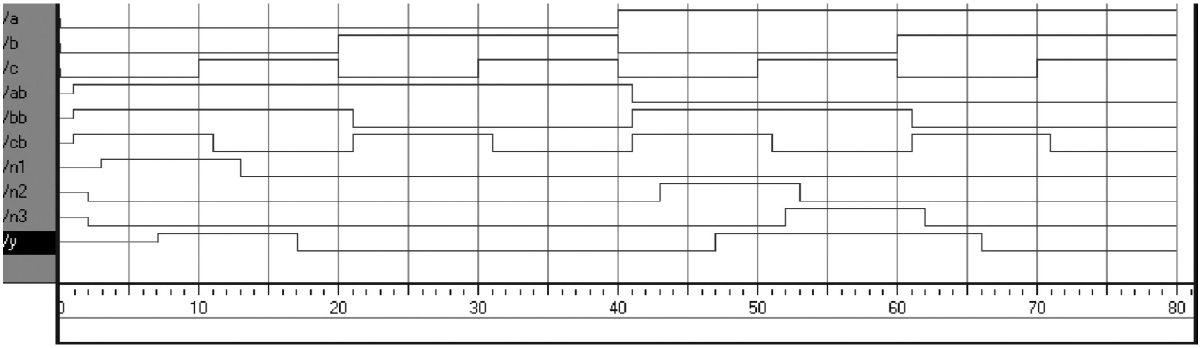


Figure 4.10 Example simulation waveforms with delays (from the ModelSim simulator)

4.3 STRUCTURAL MODELING

The previous section discussed *behavioral* modeling, describing a module in terms of the relationships between inputs and outputs. This section examines *structural* modeling, which describes a module in terms of how it is composed of simpler modules.

For example, HDL Example 4.14 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers. Each copy of the 2:1 multiplexer is called

HDL Example 4.14 STRUCTURAL MODEL OF 4:1 MULTIPLEXER

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three mux2 instances are called lowmux, highmux, and finalmux. The mux2 module must be defined elsewhere in the SystemVerilog code—see HDL Example 4.5, 4.15, or 4.34.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the mux2 ports using the component declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of mux4 was named struct, whereas architectures of modules with behavioral descriptions from Section 4.2 were named synth. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but struct and synth are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

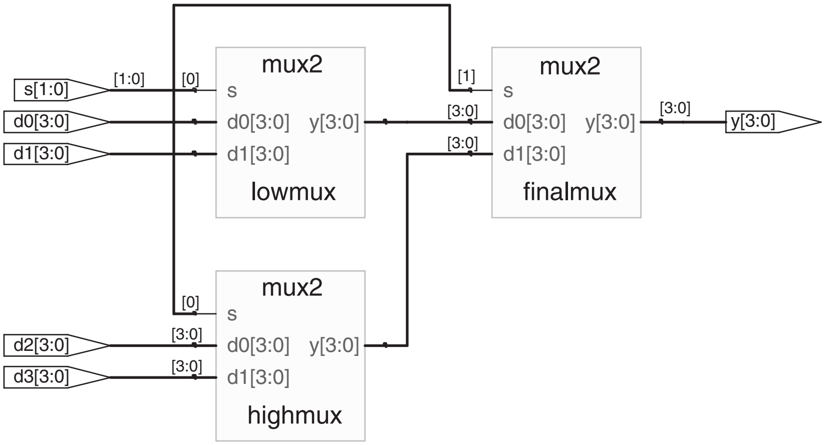


Figure 4.11 `mux4` **synthesized circuit**

an *instance*. Multiple instances of the same module are distinguished by distinct names, in this case `lowmux`, `highmux`, and `finalmux`. This is an example of regularity, in which the 2:1 multiplexer is reused many times.

HDL Example 4.15 uses structural modeling to construct a 2:1 multiplexer from a pair of tristate buffers. Building logic out of tristates is not recommended, however.

HDL Example 4.15 STRUCTURAL MODEL OF 2:1 MULTIPLEXER

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.

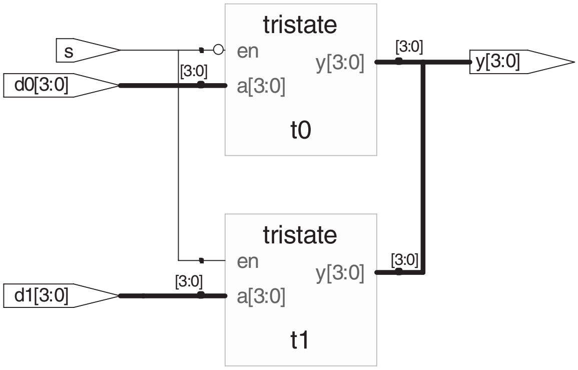


Figure 4.12 `mux2` **synthesized circuit**

HDL Example 4.16 shows how modules can access part of a bus. An 8-bit-wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.

In general, complex systems are designed *hierarchically*. The overall system is described structurally by instantiating its major components. Each of these components is described structurally from its building blocks and so forth recursively until the pieces are simple enough to describe behaviorally. It is good style to avoid (or at least to minimize) mixing structural and behavioral descriptions within a single module.

HDL Example 4.16 ACCESSING PARTS OF BUSSES

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

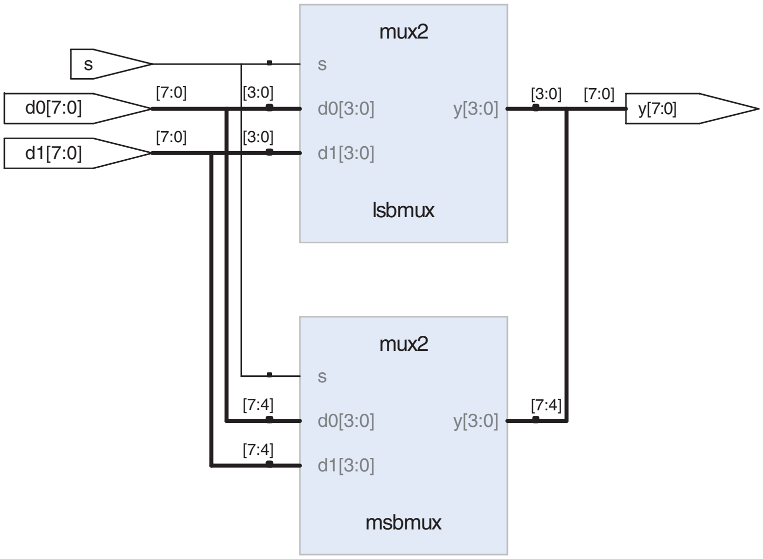


Figure 4.13 `mux2_8` **synthesized circuit**

4.4 SEQUENTIAL LOGIC

HDL synthesizers recognize certain idioms and turn them into specific sequential circuits. Other coding styles may simulate correctly but synthesize into circuits with blatant or subtle errors. This section presents the proper idioms to describe registers and latches.

4.4.1 Registers

The vast majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops. HDL Example 4.17 shows the idiom for such flip-flops.

In SystemVerilog `always` statements and VHDL `process` statements, signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory. For example, the flip-flop includes only `clk` in the sensitive list. It remembers its old value of `q` until the next rising edge of the `clk`, even if `d` changes in the interim.

In contrast, SystemVerilog continuous assignment statements (`assign`) and VHDL concurrent assignment statements (<=) are reevaluated whenever any of the inputs on the right-hand side change. Therefore, such code necessarily describes combinational logic.

HDL Example 4.17 REGISTER

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

In general, a SystemVerilog `always` statement is written in the form

```
always @(sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "`q` gets `d`"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

<= is called a *nonblocking assignment*. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that <= is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

```
process(sensitivity list) begin
  statement;
end process;
```

The `statement` is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement checks whether the change was a rising edge on `clk`. If so, then `q <= d` (pronounced "`q` gets `d`"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

`rising_edge(clk)` is synonymous with `clk'event and clk = '1'`.

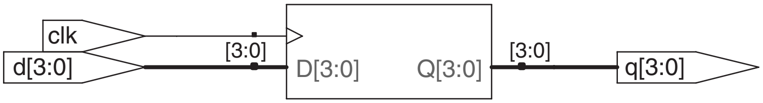


Figure 4.14 `flop` **synthesized circuit**

4.4.2 Resettable Registers

When simulation begins or power is first applied to a circuit, the output of a flop or register is unknown. This is indicated with `x` in SystemVerilog and `u` in VHDL. Generally, it is good practice to use resettable registers so that on powerup you can put your system in a known state. The reset may be either asynchronous or synchronous. Recall that asynchronous reset occurs immediately, whereas synchronous reset clears the output only on the next rising edge of the clock. HDL Example 4.18 demonstrates the idioms for flip-flops with asynchronous and synchronous resets. Note that distinguishing synchronous and asynchronous reset in a schematic can be difficult.

HDL Example 4.18 RESETTABLE REGISTER

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
   if (reset)  q <= 4'b0;
   else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design. If you wanted to use both, you would need to rename one of the modules.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `reset` is in the sensitivity list on the asynchronously resettable flop but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design. If you wanted to use both, you would need to rename one of the modules.

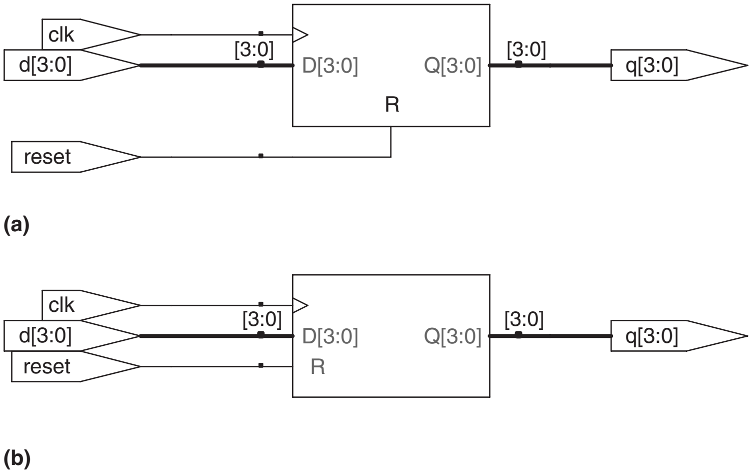


Figure 4.15 `flopr` **synthesized circuit (a) asynchronous reset, (b) synchronous reset**

4.4.3 Enabled Registers

Enabled registers respond to the clock only when the enable is asserted. HDL Example 4.19 shows an asynchronously resettable enabled register that retains its old value if both `reset` and `en` are FALSE.

HDL Example 4.19 RESETTABLE ENABLED REGISTER

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

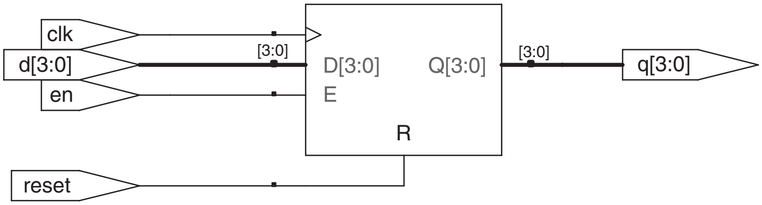


Figure 4.16 `flopenr` **synthesized circuit**

4.4.4 Multiple Registers

A single `always/process` statement can be used to describe multiple pieces of hardware. For example, consider the synchronizer from Section 3.5.5 made of two back-to-back flip-flops, as shown in Figure 4.17. HDL Example 4.20 describes the synchronizer. On the rising edge of `clk`, `d` is copied to `n1`. At the same time, `n1` is copied to `q`.

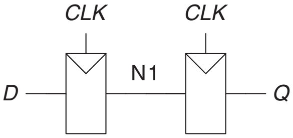


Figure 4.17 Synchronizer circuit

HDL Example 4.20 SYNCHRONIZER

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
     n1 <= d; // nonblocking
     q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to `{}` in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
       n1 <= d;
       q <= n1;
    end if;
  end process;
end;
```

`n1` must be declared as a `signal` because it is an internal signal used in the module.

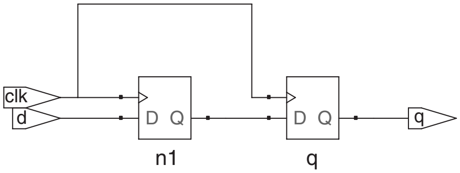


Figure 4.18 `sync` **synthesized circuit**

4.4.5 Latches

Recall from Section 3.2.2 that a D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state. HDL Example 4.21 shows the idiom for a D latch.

Not all synthesis tools support latches well. Unless you know that your tool does support latches and you have a good reason to use them, avoid them and use edge-triggered flip-flops instead. Furthermore, take care that your HDL does not imply any unintended latches, something that is easy to do if you aren't attentive. Many synthesis tools warn you when a latch is created; if you didn't expect one, track down the bug in your HDL. And if you don't know whether you intended to have a latch or not, you are probably approaching HDLs like a programming language and have bigger problems lurking.

4.5 MORE COMBINATIONAL LOGIC

In Section 4.2, we used assignment statements to describe combinational logic behaviorally. SystemVerilog `always` statements and VHDL `process`

HDL Example 4.21 D LATCH

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` is equivalent to `always @(clk, d)` and is the preferred idiom for describing a latch in SystemVerilog. It evaluates whenever `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`, so this code describes a positive level-sensitive latch. Otherwise, `q` keeps its old value. SystemVerilog can generate a warning if the `always_latch` block doesn't imply a latch.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both `clk` and `d`, so the `process` evaluates whenever `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

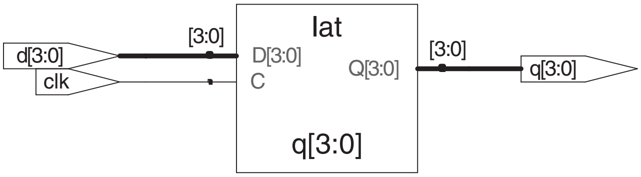


Figure 4.19 `latch` **synthesized circuit**

HDL Example 4.22 INVERTER USING always/process

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

always_comb reevaluates the statements inside the always statement whenever any of the signals on the right-hand side of <= or = in the always statement change. In this case, it is equivalent to always @(a) but is better because it avoids mistakes if signals in the always statement are renamed or added. If the code inside the always block is not combinational logic, SystemVerilog will report a warning. always_comb is equivalent to always @(*) but is preferred in SystemVerilog.

The = in the always statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In SystemVerilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

process(all) reevaluates the statements inside the process whenever any of the signals in the process change. It is equivalent to process(a) but is better because it avoids mistakes if signals in the process are renamed or added.

The begin and end process statements are required in VHDL even though the process contains only one assignment.

statements are used to describe sequential circuits because they remember the old state when no new state is prescribed. However, always/process statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL Example 4.22 uses always/process statements to describe a bank of four inverters (see Figure 4.3 for the synthesized circuit).

HDLs support *blocking* and *nonblocking assignments* in an always/process statement. A group of blocking assignments are evaluated in the order in which they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left-hand sides are updated.

HDL Example 4.23 defines a full adder using intermediate signals p and g to compute s and cout. It produces the same circuit from Figure 4.8, but uses always/process statements in place of assignment statements.

HDL Examples 4.22 and 4.23 are poor applications of always/process statements for modeling combinational logic because they require more lines than the equivalent approach with assignment statements from HDL Examples 4.2 and 4.7. However, case and if statements are convenient for modeling more complicated combinational logic. case and if statements must appear within always/process statements and are examined in the next sections.

SystemVerilog

In a SystemVerilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment, i.e., the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, := indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where := is introduced.

Nonblocking assignments are made to outputs and signals. Blocking assignments are made to variables, which are declared in `process` statements (see HDL Example 4.23). <= can also appear outside `process` statements, where it is also evaluated concurrently.

HDL Example 4.23 FULL ADDER USING `always/process`

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;             // blocking
      g = a & b;             // blocking
      s = p ^ cin;           // blocking
      cout = g | (p & cin);  // blocking
    end
endmodule
```

In this case, `always @(a, b, cin)` would have been equivalent to `always_comb`. However, `always_comb` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and, finally, `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

In this case, `process(a, b, cin)` would have been equivalent to `process(all)`. However, `process(all)` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for intermediate variables in combinational logic. This example uses blocking assignments for `p` and `g` so that they get their new values before being used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left-hand side of a blocking assignment (:=) in a `process` statement, they must be declared to be `variable` rather than `signal`. The `variable` declaration appears before the `begin` in the process where the variable is used.

4.5.1 Case Statements

A better application of using the `always/process` statement for combinational logic is a seven-segment display decoder that takes advantage of the `case` statement that must appear inside an `always/process` statement.

As you might have noticed in the seven-segment display decoder of Example 2.10, the design process for large blocks of combinational logic is tedious and prone to error. HDLs offer a great improvement, allowing you to specify the function at a higher level of abstraction and then automatically synthesize the function into gates. HDL Example 4.24 uses `case` statements to describe a seven-segment display decoder based on its truth table. The `case` statement performs different actions depending on the value of its input. A `case` statement implies combinational logic if all

HDL Example 4.24 SEVEN-SEGMENT DISPLAY DECODER

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                     abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
  endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The case statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                           abcdefg
      when X"0"   => segments <= "1111110";
      when X"1"   => segments <= "0110000";
      when X"2"   => segments <= "1101101";
      when X"3"   => segments <= "1111001";
      when X"4"   => segments <= "0110011";
      when X"5"   => segments <= "1011011";
      when X"6"   => segments <= "1011111";
      when X"7"   => segments <= "1110000";
      when X"8"   => segments <= "1111111";
      when X"9"   => segments <= "1110011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

The `case` statement checks the value of `data`. When `data` is `0`, the statement performs the action after the =>, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of `X` for hexadecimal numbers). The `others` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike SystemVerilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like `case` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

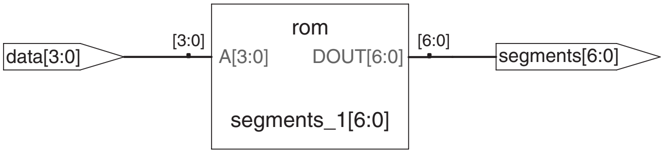


Figure 4.20 `sevenseg` **synthesized circuit**

possible input combinations are defined. Otherwise, it implies sequential logic, because the output will keep its old value in the undefined cases.

The HDL for the seven-segment display decoder synthesizes into a *read-only memory* (*ROM*) containing the 7 outputs for each of the 16 possible inputs. ROMs are discussed further in Section 5.5.6.

If the `default` or `others` clause were left out of the `case` statement, the decoder would have remembered its previous output any time data were in the range of 10–15. This is strange behavior for hardware.

Ordinary decoders are also commonly written with `case` statements. HDL Example 4.25 describes a 3:8 decoder.

4.5.2 If Statements

`always/process` statements may also contain `if` statements. The `if` statement may be followed by an `else` statement. If all possible input

HDL Example 4.25 3:8 DECODER

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);

  always_comb
    case(a)
      3'b000:  y=8'b00000001;
      3'b001:  y=8'b00000010;
      3'b010:  y=8'b00000100;
      3'b011:  y=8'b00001000;
      3'b100:  y=8'b00010000;
      3'b101:  y=8'b00100000;
      3'b110:  y=8'b01000000;
      3'b111:  y=8'b10000000;
      default: y=8'bxxxxxxxx;
    endcase
endmodule
```

The `default` statement isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an x or z.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000"  => y <= "00000001";
      when "001"  => y <= "00000010";
      when "010"  => y <= "00000100";
      when "011"  => y <= "00001000";
      when "100"  => y <= "00010000";
      when "101"  => y <= "00100000";
      when "110"  => y <= "01000000";
      when "111"  => y <= "10000000";
      when others => y <= "XXXXXXXX";
    end case;
  end process;
end;
```

The `others` clause isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an x, z, or u.

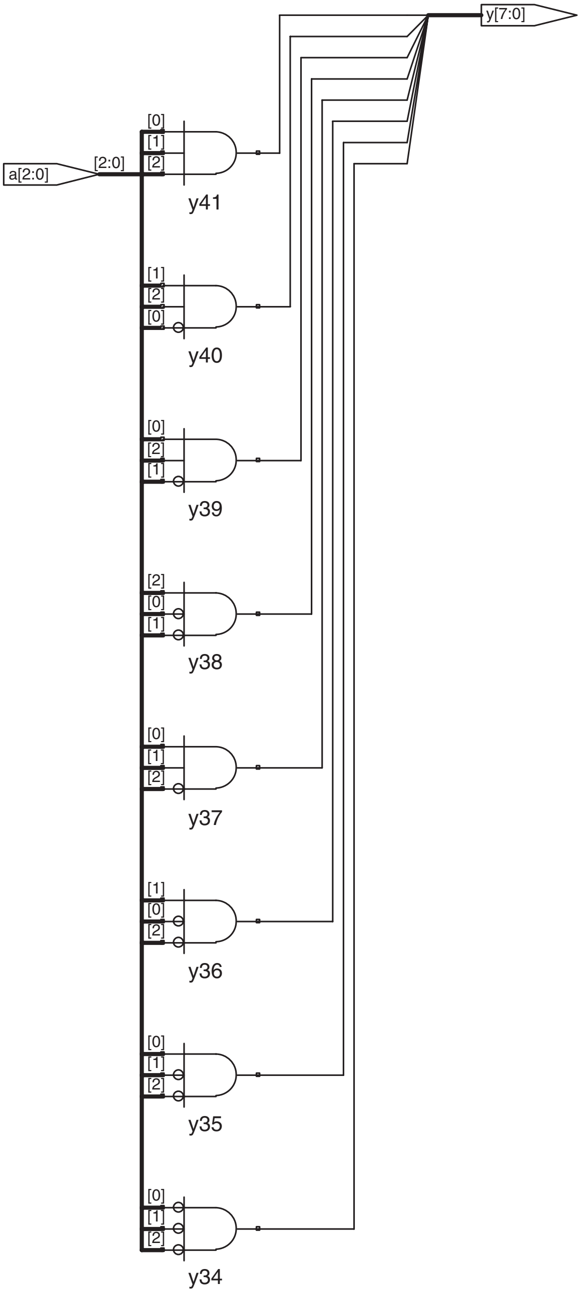


Figure 4.21 `decoder3_8` **synthesized circuit**

HDL Example 4.26 PRIORITY CIRCUIT

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);
  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside of `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    else            y <= "0000";
    end if;
  end process;
end;
```

Unlike SystemVerilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

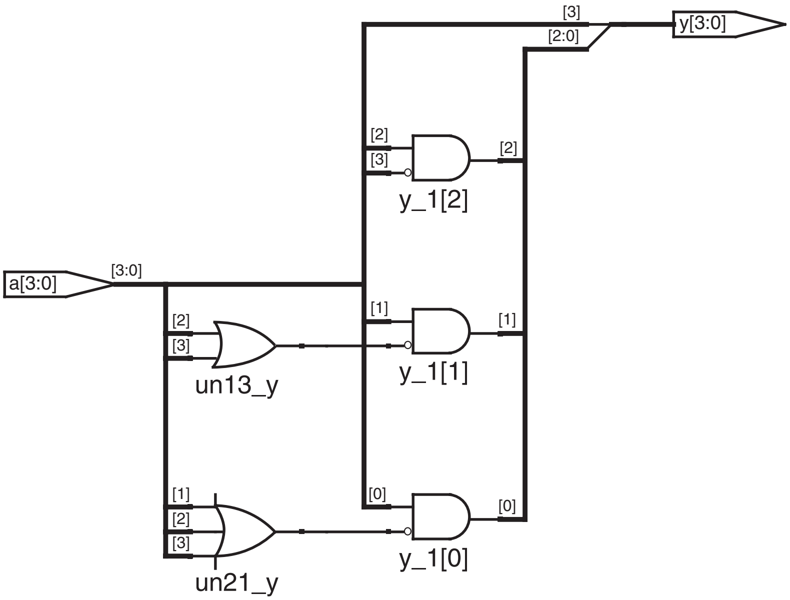


Figure 4.22 `priorityckt` synthesized circuit

combinations are handled, the statement implies combinational logic. Otherwise, it produces sequential logic (like the latch in Section 4.4.5).

HDL Example 4.26 uses `if` statements to describe a priority circuit, defined in Section 2.4. Recall that an *N*-input priority circuit sets the output TRUE that corresponds to the most significant input that is TRUE.

4.5.3 Truth Tables with Don't Cares

As examined in Section 2.7.3, truth tables may include don't cares to allow more logic simplification. HDL Example 4.27 shows how to describe a priority circuit with don't cares.

The synthesis tool generates a slightly different circuit for this module, shown in Figure 4.23, than it did for the priority circuit in Figure 4.22. However, the circuits are logically equivalent.

4.5.4 Blocking and Nonblocking Assignments

The guidelines on page 204 explain when and how to use each type of assignment. If these guidelines are not followed, it is possible to write code that appears to work in simulation but synthesizes to incorrect hardware. The optional remainder of this section explains the principles behind the guidelines.

HDL Example 4.27 PRIORITY CIRCUIT USING DON'T CARES

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

The `casez` statement acts like a `case` statement except that it also recognizes `?` as don't care.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture dontcare of priority_casez is
begin
  process(all) begin
    case? a is
      when "1---" => y <= "1000";
      when "01--" => y <= "0100";
      when "001-" => y <= "0010";
      when "0001" => y <= "0001";
      when others => y <= "0000";
    end case?;
  end process;
end;
```

The `case?` statement acts like a `case` statement except that it also recognizes `-` as don't care.

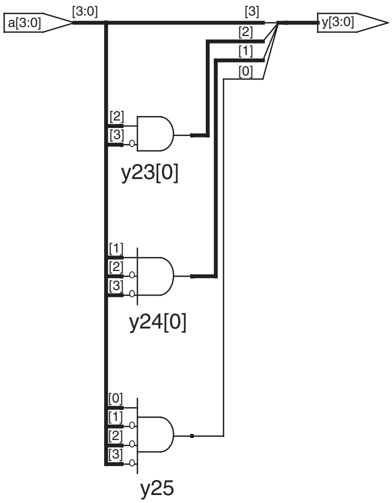


Figure 4.23 `priority_casez` **synthesized circuit**

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  if      (a[3]) y = 4'b1000;
  else if (a[2]) y = 4'b0100;
  else if (a[1]) y = 4'b0010;
  else if (a[0]) y = 4'b0001;
  else           y = 4'b0000;
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

VHDL

1. Use `process(clk)` and nonblocking assignments to model synchronous sequential logic.

```
process(clk) begin
  if rising_edge(clk) then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

```
y <= d0 when s = '0' else d1;
```

3. Use `process(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process(all)
  variable p, g: STD_LOGIC;
begin
   p := a xor b; -- blocking
   g := a and b; -- blocking
   s <= p xor cin;
   cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

Combinational Logic*

The full adder from HDL Example 4.23 is correctly modeled using blocking assignments. This section explores how it operates and how it would differ if nonblocking assignments had been used.

Imagine that `a`, `b`, and `cin` are all initially 0. `p`, `g`, `s`, and `cout` are thus 0 as well. At some time, `a` changes to 1, triggering the `always/process` statement. The four blocking assignments evaluate in the order shown here. (In the VHDL code, `s` and `cout` are assigned concurrently.) Note that `p` and `g` get their new values before `s` and `cout` are computed because of the blocking assignments. This is important because we want to compute `s` and `cout` using the new values of `p` and `g`.

1. $\texttt{p} \leftarrow 1 \oplus 0 = 1$
2. $\texttt{g} \leftarrow 1 \cdot 0 = 0$
3. $\texttt{s} \leftarrow 1 \oplus 0 = 1$
4. $\texttt{cout} \leftarrow 0 + 1 \cdot 0 = 0$

In contrast, HDL Example 4.28 illustrates the use of nonblocking assignments.

Now, consider the same case of `a` rising from 0 to 1 while `b` and `cin` are 0. The four nonblocking assignments evaluate concurrently:

$$\texttt{p} \leftarrow 1 \oplus 0 = 1 \quad \texttt{g} \leftarrow 1 \cdot 0 = 0 \quad \texttt{s} \leftarrow 0 \oplus 0 = 0 \quad \texttt{cout} \leftarrow 0 + 0 \cdot 0 = 0$$

HDL Example 4.28 FULL ADDER USING NONBLOCKING ASSIGNMENTS

SystemVerilog

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
     cout <= g | (p & cin);
  end
endmodule
```

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process(all) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Because `p` and `g` appear on the left-hand side of a nonblocking assignment in a `process` statement, they must be declared to be `signal` rather than `variable`. The `signal` declaration appears before the `begin` in the `architecture`, not the `process`.

Observe that `s` is computed concurrently with `p`. Hence, it uses the old value of `p`, not the new value. Therefore, `s` remains 0 rather than becoming 1. However, `p` does change from 0 to 1. This change triggers the `always/process` statement to evaluate a second time, as follows:

$$\texttt{p} \leftarrow 1 \oplus 0 = 1 \quad \texttt{g} \leftarrow 1 \cdot 0 = 0 \quad \texttt{s} \leftarrow 1 \oplus 0 = 1 \quad \texttt{cout} \leftarrow 0 + 1 \cdot 0 = 0$$

This time, `p` is already 1, so `s` correctly changes to 1. The nonblocking assignments eventually reach the right answer, but the `always/process` statement had to evaluate twice. This makes simulation slower, though it synthesizes to the same hardware.

Another drawback of nonblocking assignments in modeling combinational logic is that the HDL will produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

Worse yet, some synthesis tools will synthesize the correct hardware even when a faulty sensitivity list causes incorrect simulation. This leads to a mismatch between the simulation results and what the hardware actually does.

SystemVerilog

If the sensitivity list of the `always` statement in HDL Example 4.28 were written as `always @(a, b, cin)` rather than `always_comb`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

VHDL

If the sensitivity list of the process statement in HDL Example 4.28 were written as `process(a, b, cin)` rather than `process(all)`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

Sequential Logic*

The synchronizer from HDL Example 4.20 is correctly modeled using nonblocking assignments. On the rising edge of the clock, `d` is copied to `n1` at the same time that `n1` is copied to `q`, so the code properly describes two registers. For example, suppose initially that `d` = 0, `n1` = 1, and `q` = 0. On the rising edge of the clock, the following two assignments occur concurrently, so that after the clock edge, `n1` = 0 and `q` = 1.

$$\texttt{n1} \leftarrow \texttt{d} = 0 \quad \texttt{q} \leftarrow \texttt{n1} = 1$$

HDL Example 4.29 tries to describe the same module using blocking assignments. On the rising edge of `clk`, `d` is copied to `n1`. Then, this new value of `n1` is copied to `q`, resulting in `d` improperly appearing at both `n1` and `q`. The assignments occur one after the other so that after the clock edge, `q` = `n1` = 0.

1. $\texttt{n1} \leftarrow \texttt{d} = 0$
2. $\texttt{q} \leftarrow \texttt{n1} = 0$

HDL Example 4.29 BAD SYNCHRONIZER WITH BLOCKING ASSIGNMENTS

SystemVerilog

```
// Bad implementation of a synchronizer using blocking
// assignments

module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
     n1 = d; // blocking
     q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation of a synchronizer using blocking
-- assignment

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk:in  STD_LOGIC;
       d:  in  STD_LOGIC;
       q:  out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

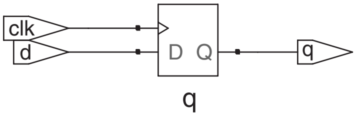


Figure 4.24 `syncbad` **synthesized circuit**

Because `n1` is invisible to the outside world and does not influence the behavior of `q`, the synthesizer optimizes it away entirely, as shown in Figure 4.24.

The moral of this illustration is to exclusively use nonblocking assignment in `always/process` statements when modeling sequential logic. With sufficient cleverness, such as reversing the orders of the assignments, you could make blocking assignments work correctly, but blocking assignments offer no advantages and only introduce the risk of unintended behavior. Certain sequential circuits will not work with blocking assignments no matter what the order.

4.6 FINITE STATE MACHINES

Recall that a finite state machine (FSM) consists of a state register and two blocks of combinational logic to compute the next state and the output given the current state and the input, as was shown in Figure 3.22. HDL descriptions of state machines are correspondingly divided into three parts to model the state register, the next state logic, and the output logic.

HDL Example 4.30 DIVIDE-BY-3 FINITE STATE MACHINE

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate = S1;
      S1:      nextstate = S2;
      S2:      nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `typedef` statement defines `statetype` to be a two-bit logic value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100}
statetype;
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary, even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
        y:         out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin

  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

In the HDL above, the output, `y`, is 1 when the `state` is `S0`. The inequality comparison uses `/=`. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

HDL Example 4.30 describes the divide-by-3 FSM from Section 3.4.2. It provides an asynchronous reset to initialize the FSM. The state register uses the ordinary idiom for flip-flops. The next state and output logic blocks are combinational.

Synthesis tools produce just a block diagram and state transition diagram for state machines; they do not show the logic gates or the inputs

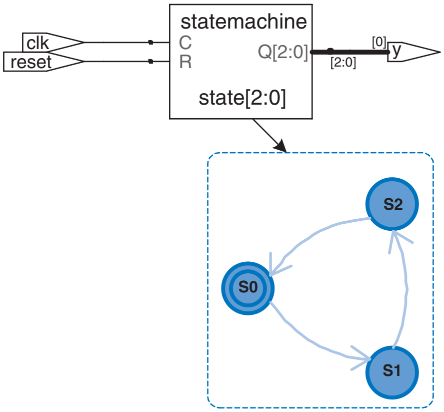


Figure 4.25 `divideby3fsm` **synthesized circuit**

and outputs on the arcs and states. Therefore, be careful that you have specified the FSM correctly in your HDL code. The state transition diagram in Figure 4.25 for the divide-by-3 FSM is analogous to the diagram in Figure 3.28(b). The double circle indicates that S0 is the reset state. Gate-level implementations of the divide-by-3 FSM were shown in Section 3.4.2.

Notice that the states are named with an enumeration data type rather than by referring to them as binary values. This makes the code more readable and easier to change.

If, for some reason, we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows.

Notice that the synthesis tool uses a 3-bit encoding (`Q[2:0]`) instead of the 2-bit encoding suggested in the SystemVerilog code.

SystemVerilog

```
// output logic
assign y = (state == S0 | state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

The next two examples describe the snail pattern recognizer FSM from Section 3.4.3. The code shows how to use `case` and `if` statements to handle next state and output logic that depend on the inputs as well as the current state. We show both Moore and Mealy modules. In the Moore machine (HDL Example 4.31), the output depends only on the current state, whereas in the Mealy machine (HDL Example 4.32), the output logic depends on both the current state and inputs.

HDL Example 4.31 PATTERN RECOGNIZER MOORE FSM

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

Note how nonblocking assignments (<=) are used in the state register to describe sequential logic, whereas blocking assignments (=) are used in the next state logic to describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  --output logic
  y <= '1' when state = S2 else '0';
end;
```

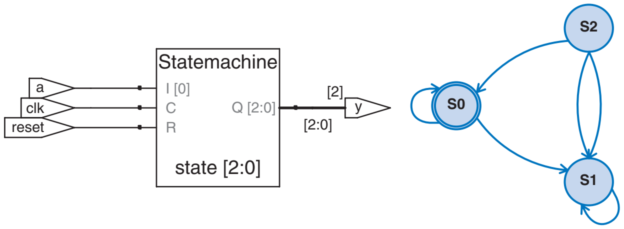


Figure 4.26 `patternMoore` **synthesized circuit**

HDL Example 4.32 PATTERN RECOGNIZER MEALY FSM

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then              state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

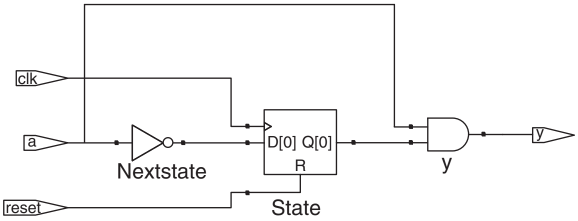


Figure 4.27 `patternMealy` **synthesized circuit**

4.7 DATA TYPES*

This section explains some subtleties about SystemVerilog and VHDL types in more depth.

4.7.1 SystemVerilog

Prior to SystemVerilog, Verilog primarily used two types: reg and wire. Despite its name, a reg signal might or might not be associated with a register. This was a great source of confusion for those learning the language. SystemVerilog introduced the logic type to eliminate the confusion; hence, this book emphasizes the logic type. This section explains the reg and wire types in more detail for those who need to read old Verilog code.

In Verilog, if a signal appears on the left-hand side of <= or = in an always block, it must be declared as reg. Otherwise, it should be declared as wire. Hence, a reg signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an always block.

Input and output ports default to the wire type unless their type is explicitly defined as reg. The following example shows how a flip-flop is described in conventional Verilog. Note that clk and d default to wire, while q is explicitly defined as reg because it appears on the left-hand side of <= in the always block.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog introduces the logic type. logic is a synonym for reg and avoids misleading users about whether it is actually a flip-flop. Moreover, SystemVerilog relaxes the rules on assign statements and hierarchical port instantiations so that logic can be used outside always blocks where a wire traditionally would have been required. Thus, nearly all SystemVerilog signals can be logic. The exception is that signals with multiple drivers (e.g., a tristate bus) must be declared as a net, as described in HDL Example 4.10. This rule allows SystemVerilog to generate an error message rather than an x value when a logic signal is accidentally connected to multiple drivers.

The most common type of net is called a wire or tri. These two types are synonymous, but wire is conventionally used when a single driver is present and tri is used when multiple drivers are present. Thus, wire is obsolete in SystemVerilog because logic is preferred for signals with a single driver.

When a tri net is driven to a single value by one or more drivers, it takes on that value. When it is undriven, it floats (z). When it is driven to a different value (0, 1, or x) by multiple drivers, it is in contention (x).

There are other net types that resolve differently when undriven or driven by multiple sources. These other types are rarely used but may be substituted anywhere a tri net would normally appear (e.g., for signals with multiple drivers). Each is described in Table 4.7.

Table 4.7 Net Resolution

Net Type	No Driver	Conflicting Drivers
tri	z	x
trireg	previous value	x
triand	z	0 if there are any 0
trior	z	1 if there are any 1
tri0	0	x
tri1	1	x

4.7.2 VHDL

Unlike SystemVerilog, VHDL enforces a strict data typing system that can protect the user from some errors but that is also clumsy at times.

Despite its fundamental importance, the STD_LOGIC type is not built into VHDL. Instead, it is part of the IEEE.STD_LOGIC_1164 library. Thus, every file must contain the library statements shown in the previous examples.

Moreover, IEEE.STD_LOGIC_1164 lacks basic operations such as addition, comparison, shifts, and conversion to integers for the STD_LOGIC_VECTOR data. These were finally added to the VHDL 2008 standard in the IEEE.NUMERIC_STD_UNSIGNED library.

VHDL also has a BOOLEAN type with two values: true and false. BOOLEAN values are returned by comparisons (such as the equality comparison, s='0') and are used in conditional statements, such as when and if. Despite the temptation to believe a BOOLEAN true value should be equivalent to a STD_LOGIC '1' and BOOLEAN false should mean STD_LOGIC '0', these types were not interchangeable prior to VHDL 2008. For example, in old VHDL code, one must write

```
y <= d1 when (s='1') else d0;
```

while in VHDL 2008, the when statement automatically converts s from STD_LOGIC to BOOLEAN so one can simply write

```
y <= d1 when s else d0;
```

Even in VHDL 2008, it is still necessary to write

```
q <= '1' when (state=S2) else '0';
```

instead of

```
q <= (state=S2);
```

because (state=S2) returns a BOOLEAN result, which cannot be directly assigned to the STD_LOGIC signal y.

Although we do not declare any signals to be BOOLEAN, they are automatically implied by comparisons and used by conditional statements. Similarly, VHDL has an INTEGER type that represents both positive and negative integers. Signals of type INTEGER span at least the values $-(2^{31} - 1)$ to $2^{31} - 1$. Integer values are used as indices of busses. For example, in the statement

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2, and 3 are integers serving as an index to choose bits of the a signal. We cannot directly index a bus with a STD_LOGIC or STD_LOGIC_VECTOR signal. Instead, we must convert the signal to an INTEGER. This is demonstrated in the example below for an 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The TO_INTEGER function is defined in the IEEE.NUMERIC_STD_UNSIGNED library and performs the conversion from STD_LOGIC_VECTOR to INTEGER for positive (unsigned) values.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(TO_INTEGER(s));
end;
```

VHDL is also strict about out ports being exclusively for output. For example, the following code for two- and three-input AND gates is illegal VHDL because v is an output and is also used to compute w.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in STD_LOGIC;
       v, w: out   STD_LOGIC);
end;

architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

VHDL defines a special port type, buffer, to solve this problem. A signal connected to a buffer port behaves as an output but may also be used within the module. The corrected entity definition follows. Verilog and SystemVerilog do not have this limitation and do not require buffer ports. VHDL 2008 eliminates this restriction by allowing out ports to be readable.

```
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v: buffer   STD_LOGIC;
       w: out      STD_LOGIC);
end;
```

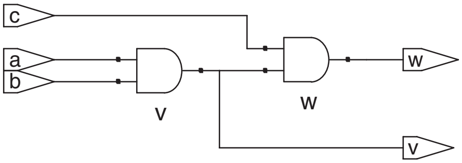


Figure 4.28 and23 **synthesized circuit**

Most operations such as addition, subtraction, and Boolean logic are identical whether a number is signed or unsigned. However, magnitude comparison, multiplication, and arithmetic right shifts are performed differently for signed two's complement numbers than for unsigned binary numbers. These operations will be examined in Chapter 5. HDL Example 4.33 describes how to indicate that a signal represents a signed number.

HDL Example 4.33 (a) UNSIGNED MULTIPLIER (b) SIGNED MULTIPLIER

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y = a * b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);

  assign y = a * b;
endmodule
```

In SystemVerilog, signals are considered unsigned by default. Adding the signed modifier (e.g., logic signed [3:0] a) causes the signal a to be treated as a signed—that is, two's complement—number.

VHDL

```
-- 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

VHDL uses the NUMERIC_STD_UNSIGNED library to perform arithmetic and comparison operations on STD_LOGIC_VECTORs. The vectors are treated as unsigned.

```
use IEEE.NUMERIC_STD_UNSIGNED.all;
```

VHDL also defines UNSIGNED and SIGNED data types in the IEEE.NUMERIC_STD library, but these involve type conversions beyond the scope of this chapter.

4.8 PARAMETERIZED MODULES*

So far, all of our modules have had fixed-width inputs and outputs. For example, we had to define separate modules for 4- and 8-bit-wide 2:1 multiplexers. HDLs permit variable bit widths using parameterized modules.

HDL Example 4.34 declares a parameterized 2:1 multiplexer with a default width of 8, then uses it to create 8- and 12-bit 4:1 multiplexers.

HDL Example 4.34 PARAMETERIZED *N*-BIT 2:1 MULTIPLEXERS

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

SystemVerilog allows a `#(parameter ...)` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, in this case called `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#( )` before the instance name, as shown below.

```
module mux4_12(input  logic [11:0] d0, d1, d2, d3,
               input  logic [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(...)` in defining and overriding parameters.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(1 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers, using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

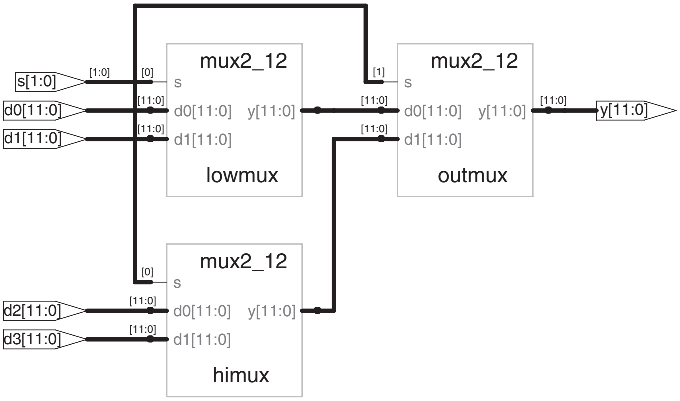


Figure 4.29 `mux4_12` **synthesized circuit**

HDL Example 4.35 PARAMETERIZED *N*:2^N DECODER

SystemVerilog

```
module decoder
  #(parameter N = 3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
     y = 0;
     y[a] = 1;
   end
endmodule
```

`2**N` indicates 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in  STD_LOGIC_VECTOR(N-1 downto 0);
        y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

`2**N` indicates 2^N.

HDL Example 4.35 shows a decoder, which is an even better application of parameterized modules. A large *N*:2^N decoder is cumbersome to specify with `case` statements, but easy using parameterized code that simply sets the appropriate output bit to 1. Specifically, the decoder uses blocking assignments to set all the bits to 0, then changes the appropriate bit to 1.

HDLs also provide `generate` statements to produce a variable amount of hardware, depending on the value of a parameter. `generate` supports `for` loops and `if` statements to determine how many of what types of hardware to produce. HDL Example 4.36 demonstrates how to use `generate` statements to produce an *N*-input AND function from a cascade of two-input AND gates. Of course, a reduction operator would be cleaner and simpler for this application, but the example illustrates the general principle of hardware generators.

Use `generate` statements with caution; it is easy to produce a large amount of hardware unintentionally!

HDL Example 4.36 PARAMETERIZED *N*-INPUT AND GATE

SystemVerilog

```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0] = a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i] = a[i] & x[i-1];
    end

  endgenerate

  assign y = x[width-1];
endmodule
```

The `for` statement loops through `i` = 1, 2, … , `width-1` to produce many consecutive AND gates. The `begin` in a `generate` `for` loop must be followed by a `:` and an arbitrary label (`forloop`, in this case).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
  x(0) <= a(0);
  gen: for i in 1 to width-1 generate
    x(i) <= a(i) and x(i-1);
  end generate;
  y <= x(width-1);
end;
```

The generate loop variable `i` does not need to be declared.

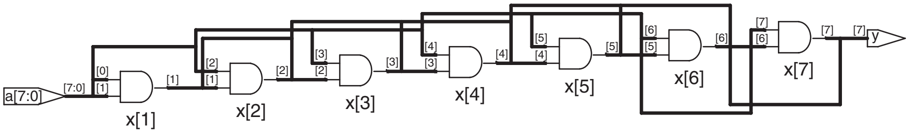


Figure 4.30 `andN` **synthesized circuit**

4.9 TESTBENCHES

A *testbench* is an HDL module that is used to test another module, called the *device under test* (*DUT*). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Some tools also call the module to be tested the *unit under test* (*UUT*).

Consider testing the `sillyfunction` module from Section 4.1.1 that computes $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. This is a simple module, so we can perform exhaustive testing by applying all eight possible test vectors.

HDL Example 4.37 TESTBENCH

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; --wait forever
  end process;
end;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely. Otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

HDL Example 4.37 demonstrates a simple testbench. It instantiates the DUT, then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order. The user must view the results of the simulation and verify by inspection that the correct outputs are produced. Testbenches are simulated the same as other HDL modules. However, they are not synthesizeable.

Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind. If you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle. A much better approach is to write a self-checking testbench, shown in HDL Example 4.38.

Writing code for each test vector also becomes tedious, especially for modules that require a large number of vectors. An even better

HDL Example 4.38 SELF-CHECKING TESTBENCH

SystemVerilog

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;  #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;  #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

The SystemVerilog assert statement checks whether a specified condition is true. If not, it executes the else statement. The $error system task in the else statement prints an error message describing the assertion failure. assert is ignored during synthesis.

In SystemVerilog, comparison using == or != is effective between signals that do not take on the values of x and z. Testbenches use the === and !== operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be x or z.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The assert statement checks a condition and prints the message given in the report clause if the condition is not satisfied. assert is meaningful only in simulation, not in synthesis.

approach is to place the test vectors in a separate file. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.

HDL Example 4.39 demonstrates such a testbench. The testbench generates a clock using an always/process statement with no sensitivity list, so that it is continuously reevaluated. At the beginning of the simulation, it reads the test vectors from a text file and pulses reset for two cycles. Although the clock and reset aren't necessary to test combinational logic, they are included because they would be important when testing a sequential DUT. `example.txt` is a text file containing the *test vectors*, the inputs and expected output written in binary:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL Example 4.39 TESTBENCH WITH TEST VECTOR FILE

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.txt", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin // check result
        $display("Error: inputs = %b", {a, b, c});
        $display(" outputs = %b (%b expected)", y, yexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $stop;
      end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y:  STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset:  STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.txt", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
```

$readmemb reads a file of binary numbers into the testvectors array. $readmemh is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs and the expected output based on the four bits in the current test vector.

The testbench compares the generated output, y, with the expected output, yexpected, and prints an error if they don't match. %b and %d indicate to print the values in binary and decimal, respectively. $display is a system task to print in the simulator window. For example, $display ("%b %b", y, yexpected); prints the two values, y and yexpected, in binary. %h prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the testvectors array. $stop stops the simulation.

Note that even though the SystemVerilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

```
    read(L, vector_out);
    (a, b, c) <= vector_in(2 downto 0) after 1 ns;
    y_expected <= vector_out after 1 ns;

    -- check results on falling edge
    wait until falling_edge(clk);

    if y /= y_expected then
      report "Error: y = " & std_logic'image(y);
      errors := errors + 1;
    end if;

    vectornum := vectornum + 1;
   end loop;

   -- summarize results at end of simulation
   if (errors = 0) then
    report "NO ERRORS -- " &
           integer'image(vectornum) &
           " tests completed successfully."
           severity failure;
   else
    report integer'image(vectornum) &
           " tests completed, errors = " &
           integer'image(errors)
           severity failure;
   end if;
 end process;
end;
```

The VHDL code uses file reading commands beyond the scope of this chapter, but it gives the sense of what a self-checking testbench looks like.

New inputs are applied on the rising edge of the clock, and the output is checked on the falling edge of the clock. Errors are reported as they occur. At the end of the simulation, the testbench prints the total number of test vectors applied and the number of errors detected.

The testbench in HDL Example 4.39 is overkill for such a simple circuit. However, it can easily be modified to test more complex circuits by changing the example.txt file, instantiating the new DUT, and changing a few lines of code to set the inputs and check the outputs.

4.10 SUMMARY

Hardware description languages (HDLs) are extremely important tools for modern digital designers. Once you have learned SystemVerilog or VHDL, you will be able to specify digital systems much faster than if you had to draw the complete schematics. The debug cycle is also often much faster because modifications require code changes instead of tedious schematic rewiring. However, the debug cycle can be much *longer* using HDLs if you don't have a good idea of the hardware your code implies.

HDLs are used for both simulation and synthesis. Logic simulation is a powerful way to test a system on a computer before it is turned into hardware. Simulators let you check the values of signals inside your system that might be impossible to measure on a physical piece of hardware. Logic synthesis converts the HDL code into digital logic circuits.

The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program. The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce. If you don't know what hardware you are implying, you are almost certain not going to get what you want. Instead, begin by sketching a block diagram of your system, identifying which portions are combinational logic, which portions are sequential circuits or finite state machines, and so forth. Then, write HDL code for each portion, using the correct idioms to imply the kind of hardware you need.

Exercises

The following exercises may be done using your favorite HDL. If you have a simulator available, test your design. Print the waveforms and explain how they prove that it works. If you have a synthesizer available, synthesize your code. Print the generated circuit diagram and explain why it matches your expectations.

Exercise 4.1 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise1(input  logic a, b, c,
                 output logic y, z);

  assign y = a & b & c | a & b & ~c | a & ~b & c;
  assign z = a & b | ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or
       (a and not b and c);
  z <= (a and b) or (not a and not b);
end;
```

Exercise 4.2 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise2(input  logic [3:0] a,
                 output logic [1:0] y);
  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(all) begin
    if    a(0) then y <= "11";
    elsif a(1) then y <= "10";
    elsif a(2) then y <= "01";
    elsif a(3) then y <= "00";
    else            y <= a(1 downto 0);
    end if;
  end process;
end;
```

Exercise 4.3 Write an HDL module that computes a four-input XOR function. The input is $a_{3:0}$ and the output is y.

Exercise 4.4 Write a self-checking testbench for Exercise 4.3. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works.

Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.5 Write an HDL module called `minority`. It receives three inputs, `a`, `b`, and `c`. It produces one output, `y`, that is TRUE if at least two of the inputs are FALSE.

Exercise 4.6 Write an HDL module for a hexadecimal seven-segment display decoder. The decoder should handle the digits A, B, C, D, E, and F, as well as 0–9.

Exercise 4.7 Write a self-checking testbench for Exercise 4.6. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.8 Write an 8:1 multiplexer module called `mux8` with inputs $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, and output `y`.

Exercise 4.9 Write a structural module to compute the logic function $y = a\overline{b} + \overline{b}\overline{c} + \overline{a}bc$ using multiplexer logic. Use the 8:1 multiplexer from Exercise 4.8.

Exercise 4.10 Repeat Exercise 4.9 using a 4:1 multiplexer and as many NOT gates as you need.

Exercise 4.11 Section 4.5.4 pointed out that a synchronizer could be correctly described with blocking assignments if the assignments were given in the proper order. Think of a simple sequential circuit that cannot be correctly described with blocking assignments, regardless of order.

Exercise 4.12 Write an HDL module for an eight-input priority circuit.

Exercise 4.13 Write an HDL module for a 2:4 decoder.

Exercise 4.14 Write an HDL module for a 6:64 decoder using three instances of the 2:4 decoders from Exercise 4.13 and a bunch of three-input AND gates.

Exercise 4.15 Write HDL modules that implement the Boolean equations from Exercise 2.13.

Exercise 4.16 Write an HDL module that implements the circuit from Exercise 2.26.

Exercise 4.17 Write an HDL module that implements the circuit from Exercise 2.27.

Exercise 4.18 Write an HDL module that implements the logic function from Exercise 2.28. Pay careful attention to how you handle don't cares.

Exercise 4.19 Write an HDL module that implements the functions from Exercise 2.35.

Exercise 4.20 Write an HDL module that implements the priority encoder from Exercise 2.36.

Exercise 4.21 Write an HDL module that implements the modified priority encoder from Exercise 2.37.

Exercise 4.22 Write an HDL module that implements the binary-to-thermometer code converter from Exercise 2.38.

Exercise 4.23 Write an HDL module implementing the days-in-month function from Question 2.2.

Exercise 4.24 Sketch the state transition diagram for the FSM described by the following HDL code.

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  logic [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always_ff @(posedge clk, posedge reset)
   if (reset) state <= S0;
   else       state <= nextstate;

  always_comb
   case (state)
    S0: if (a ^ b) nextstate = S1;
        else       nextstate = S0;
    S1: if (a & b) nextstate = S2;
        else       nextstate = S0;
    S2: if (a | b) nextstate = S3;
        else       nextstate = S0;
    S3: if (a | b) nextstate = S3;
        else       nextstate = S0;
   endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if (a xor b) then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if (a and b) then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when((state = S1) or (state = S2))
       else '0';
end;
```

Exercise 4.25 Sketch the state transition diagram for the FSM described by the following HDL code. An FSM of this nature is used in a branch predictor on some microprocessors.

SystemVerilog

```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter S0 = 5'b00001;
  parameter S1 = 5'b00010;
  parameter S2 = 5'b00100;
  parameter S3 = 5'b01000;
  parameter S4 = 5'b10000;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken) nextstate = S1;
          else       nextstate = S0;
      S1: if (taken) nextstate = S2;
          else       nextstate = S0;
      S2: if (taken) nextstate = S3;
          else       nextstate = S1;
      S3: if (taken) nextstate = S4;
          else       nextstate = S2;
      S4: if (taken) nextstate = S4;
          else       nextstate = S3;
      default:       nextstate = S2;
    endcase

  assign predicttaken = (state == S4) |
                        (state == S3) |
                        (state == S2 & back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset:    in  STD_LOGIC;
       taken, back:   in  STD_LOGIC;
       predicttaken:  out STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

process(all) begin
    case state is
      when S0 => if taken then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if taken then
                   nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken then
                   nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken then
                   nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others => nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                    ((state = S4) or (state = S3) or
                    (state = S2 and back = '1'))
  else '0';
end;
```

Exercise 4.26 Write an HDL module for an SR latch.

Exercise 4.27 Write an HDL module for a *JK flip-flop*. The flip-flop has inputs, *clk*, *J*, and *K*, and output *Q*. On the rising edge of the clock, *Q* keeps its old value if $J = K = 0$. It sets *Q* to 1 if $J = 1$, resets *Q* to 0 if $K = 1$, and inverts *Q* if $J = K = 1$.

Exercise 4.28 Write an HDL module for the latch from Figure 3.18. Use one assignment statement for each gate. Specify delays of 1 unit or 1 ns to each gate. Simulate the latch and show that it operates correctly. Then, increase the inverter delay. How long does the delay have to be before a race condition causes the latch to malfunction?

Exercise 4.29 Write an HDL module for the traffic light controller from Section 3.4.1.

Exercise 4.30 Write three HDL modules for the factored parade mode traffic light controller from Example 3.8. The modules should be called `controller`, `mode`, and `lights`, and they should have the inputs and outputs shown in Figure 3.33(b).

Exercise 4.31 Write an HDL module describing the circuit in Figure 3.42.

Exercise 4.32 Write an HDL module for the FSM with the state transition diagram given in Figure 3.69 from Exercise 3.22.

Exercise 4.33 Write an HDL module for the FSM with the state transition diagram given in Figure 3.70 from Exercise 3.23.

Exercise 4.34 Write an HDL module for the improved traffic light controller from Exercise 3.24.

Exercise 4.35 Write an HDL module for the daughter snail from Exercise 3.25.

Exercise 4.36 Write an HDL module for the soda machine dispenser from Exercise 3.26.

Exercise 4.37 Write an HDL module for the Gray code counter from Exercise 3.27.

Exercise 4.38 Write an HDL module for the UP/DOWN Gray code counter from Exercise 3.28.

Exercise 4.39 Write an HDL module for the FSM from Exercise 3.29.

Exercise 4.40 Write an HDL module for the FSM from Exercise 3.30.

Exercise 4.41 Write an HDL module for the serial two's complementer from Question 3.2.

Exercise 4.42 Write an HDL module for the circuit in Exercise 3.31.

Exercise 4.43 Write an HDL module for the circuit in Exercise 3.32.

Exercise 4.44 Write an HDL module for the circuit in Exercise 3.33.

Exercise 4.45 Write an HDL module for the circuit in Exercise 3.34. You may use the full adder from Section 4.2.5.

SystemVerilog Exercises

The following exercises are specific to SystemVerilog.

Exercise 4.46 What does it mean for a signal to be declared `tri` in SystemVerilog?

Exercise 4.47 Rewrite the `syncbad` module from HDL Example 4.29. Use nonblocking assignments, but change the code to produce a correct synchronizer with two flip-flops.

Exercise 4.48 Consider the following two SystemVerilog modules. Do they have the same function? Sketch the hardware each one implies.

```
module code1(input  logic clk, a, b, c,
             output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;

  end
endmodule

module code2(input  logic a, b, c, clk,
             output logic y);

  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

Exercise 4.49 Repeat Exercise 4.48 if the <= is replaced by = in every assignment.

Exercise 4.50 The following SystemVerilog modules show errors that the authors have seen students make in the laboratory. Explain the error in each module and show how to fix it.

(a)
```
module latch(input logic       clk,
             input logic [3:0] d,
             output reg  [3:0] q);
  always @(clk)
    if (clk) q <= d;
endmodule
```

(b)
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always @(a)
   begin
     y1 = a & b;
     y2 = a | b;
     y3 = a ^ b;
     y4 = ~(a & b);
     y5 = ~(a | b);
   end
endmodule
```

(c)
```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   always @(posedge s)
     if(s) y <= d1;
     else  y <= d0;
endmodule
```

(d)
```
module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);
   always @(posedge clk)
     q1=d1;
     q0=d0;
endmodule
```

(e)
```
module FSM(input  logic clk,
           input  logic a,
           output logic out1, out2);

    logic state;

    // next state logic and register (sequential)
    always_ff @(posedge clk)
     if (state==0) begin
         if (a)  state <= 1;
     end else begin
         if (~a) state <= 0;
     end
```

```
   always_comb // output logic (combinational)
     if (state == 0) out1 = 1;
     else            out2 = 1;
endmodule
```

(f)
```
module priority(input  logic [3:0] a,
                output logic [3:0] y);

   always_comb
     if      (a[3]) y = 4'b1000;
     else if (a[2]) y = 4'b0100;
     else if (a[1]) y = 4'b0010;
     else if (a[0]) y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

   logic [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // state register
   always_ff @(posedge clk, posedge reset)
     if (reset) state <= S0;
     else       state <= nextstate;

   // next state logic
   always @(state)
     case (state)
       S0: nextstate = S1;
       S1: nextstate = S2;
       S2: nextstate = S0;
     endcase

   // output logic
   assign out = (state == S2);
endmodule
```

(h)
```
module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output tri   [3:0] y);
   tristate t0(d0, s, y);
   tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
                input  logic [3:0] d,
                output logic [3:0] q);
```

```
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;

  always @(set)
    if (set)   q <= 1;
endmodule
```

(j)
```
module and3(input  logic a, b, c,
            output logic y);

  logic tmp;

  always @(a, b, c)
  begin
    tmp <= a & b;
    y   <= tmp & c;
  end
endmodule
```

VHDL Exercises

The following exercises are specific to VHDL.

Exercise 4.51 In VHDL, why is it necessary to write

```
q <= '1' when state = S0 else '0';
```

rather than simply

```
q <= (state = S0);
```

Exercise 4.52 Each of the following VHDL modules contain errors. For brevity, only the architecture is shown; assume that the `library` use clause and `entity` declaration are correct. Explain the errors and show how to fix them.

(a)
```
architecture synth of latch is
begin
  process(clk) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process(a) begin
    Y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process(clk)
    if rising_edge(clk) then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
   process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    end if;
   end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(state) begin
    case state is
      when S0 => nextstate <= S1;
      when S1 => nextstate <= S2;
      when S2 => nextstate <= S0;
    end case;
  end process;

  q <= '1' when state = S0 else '0';
end;
```

(f)
```
architecture struct of mux2 is
   component tristate
     port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
          en: in  STD_LOGIC;
          y:  out STD_LOGIC_VECTOR(3 downto 0));
   end component;

begin
  t0: tristate port map(d0, s, y);
  t1: tristate port map(d1, s, y);
end;
```

(g)
```
architecture asynchronous of floprs is
begin
  process(clk, reset) begin
    if reset then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;

  process(set) begin
    if set then
      q <= '1';
    end if;
  end process;
end;
```

Interview Questions

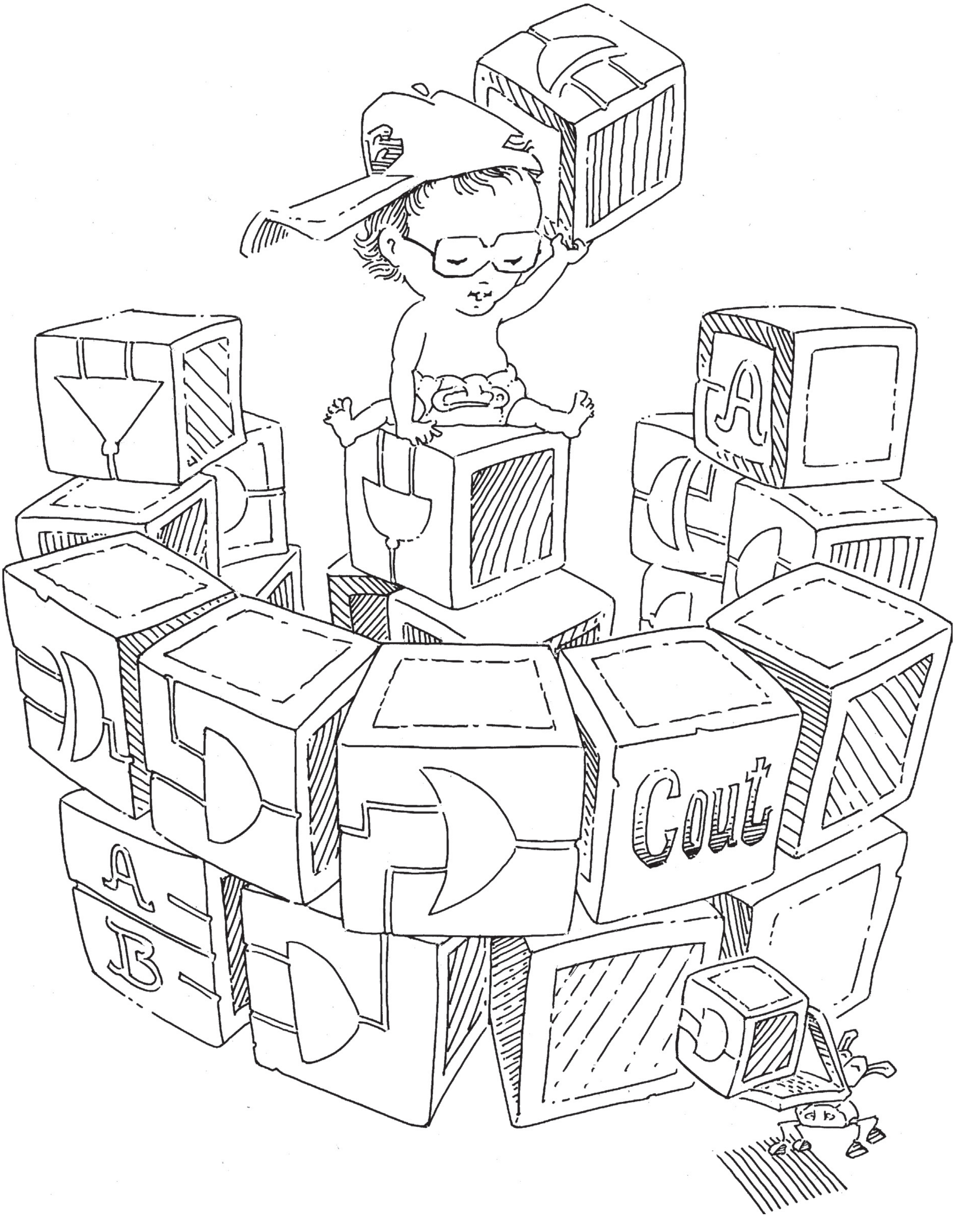
The following exercises present questions that have been asked at interviews for digital design jobs.

Question 4.1 Write a line of HDL code that gates a 32-bit bus called `data` with another signal called `sel` to produce a 32-bit `result`. If `sel` is TRUE, `result = data`. Otherwise, `result` should be all 0's.

Question 4.2 Explain the difference between blocking and nonblocking assignments in SystemVerilog. Give examples.

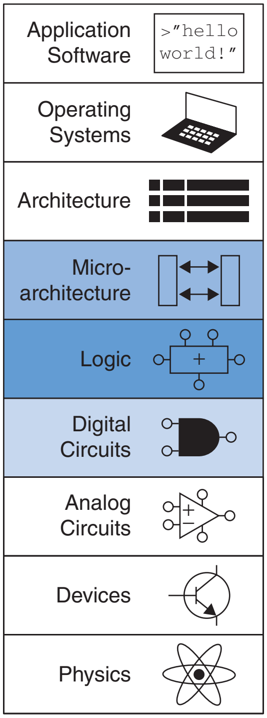
Question 4.3 What does the following SystemVerilog statement do?

```
assign result = |(data[15:0] & 16'hC820);
```


A
Cout
A
B

Digital Building Blocks

5



5.1 INTRODUCTION

Up to this point, we have examined the design of combinational and sequential circuits using Boolean equations, schematics, and HDLs. This chapter introduces more elaborate combinational and sequential building blocks used in digital systems. These blocks include arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are not only useful in their own right but they also demonstrate the principles of hierarchy, modularity, and regularity. The building blocks are hierarchically assembled from simpler components, such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes. In Chapter 7, we use many of these building blocks to build a microprocessor.

5.2 ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

5.2.1 Addition

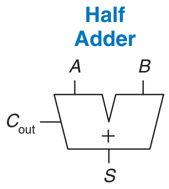
Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to *N*-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00005-2

Half Adder

We begin by building a *half adder*. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half adder can be built from an XOR gate and an AND gate.

In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output C_{out} of the first column of 1-bit addition and the input C_{in} to the second column of addition. However, the half adder lacks a C_{in} input to accept the C_{out} of the previous column. The *full adder*, described in the next section, solves this problem.



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

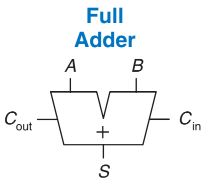
Figure 5.1 1-bit half adder

$$\begin{array}{r} 1 \\ 0001 \\ +0101 \\ \hline 0110 \end{array}$$

Figure 5.2 Carry bit

Full Adder

A *full adder*, introduced in Section 2.1, accepts the carry in C_{in} as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 5.3 1-bit full adder

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in C_{in} to produce an N-bit result S and a carry out C_{out}. It is commonly called a *carry propagate adder* (*CPA*) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and prefix adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. It is a good application of modularity and regularity: the full adder module is reused many times to form a larger system. The ripple-carry adder has the disadvantage of being slow when N is large. S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}, as shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder,

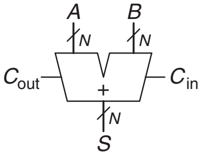


Figure 5.4 Carry propagate adder

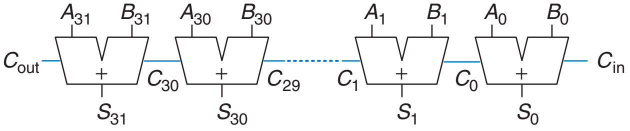


Figure 5.5 32-bit ripple-carry adder

t_{ripple}, grows directly with the number of bits, as given in Equation. 5.1, where t_{FA} is the delay of a full adder.

$$t_{\text{ripple}} = Nt_{FA} \tag{5.1}$$

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder (*CLA*) is another type of carry propagate adder that solves this problem by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus, it is said to *look ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder may be divided into eight 4-bit blocks.

CLAs use *generate* (*G*) and *propagate* (*P*) signals that describe how a column or block determines the carry out. The *i*th column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The *i*th column of an adder is guaranteed to generate a carry C_i if A_i and B_i are both 1. Hence, G_i, the generate signal for column *i*, is calculated as $G_i = A_i \,\&\, B_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The *i*th column will propagate a carry in, C_{i-1}, if either A_i or B_i is 1. Thus, $P_i = \mathrm{A}_i \mid B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the adder. The *i*th column of an adder will generate a carry out C_i if it either generates a carry, G_i, or propagates a carry in, $P_i\mathrm{C}_{i-1}$. In equation form,

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1} \tag{5.2}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define 1-bit block generate and propagate signals, $G_{i:j}$ and $P_{i:j}$, for blocks spanning columns *i* through *j*.

A block generates a carry if the most significant column generates a carry, or if the previous column generated a carry and the most significant column propagates it, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)) \tag{5.3}$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3P_2P_1P_0 \tag{5.4}$$

Schematics typically show signals flowing from left to right. Arithmetic circuits break this rule because the carries flow from right to left (from the least significant column to the most significant column). We keep the least significant column on the right and the most significant column on the left because that is how we're used to viewing and adding numbers.

Throughout the ages, people have used many devices to perform arithmetic. Toddlers count on their fingers (and some adults stealthily do too). The Chinese and Babylonians invented the abacus as early as 2400 BC. Slide rules, invented in 1630, were in use until the 1970's, when scientific hand calculators became prevalent. Computers and digital calculators are ubiquitous today. What will be next?

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i, using the carry in to the block, C_{j-1}.

$$C_i = G_{i:j} + P_{i:j}C_{j-1} \tag{5.5}$$

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the column generate and propagate signals, G_i and P_i, using signals A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

All of the CLA blocks compute the 1-bit column and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. Then, C_{in} advances directly to C_{out} through the AND/OR gate in each block until the last. Specifically, after all of the column and block propagate and generate signals are calculated, C_{in} proceeds through the AND/OR gate to produce C_3; C_3 then proceeds through its block's AND/OR gate to produce C_7; C_7 proceeds through its block's AND/OR gate to produce C_{11}, and so on until

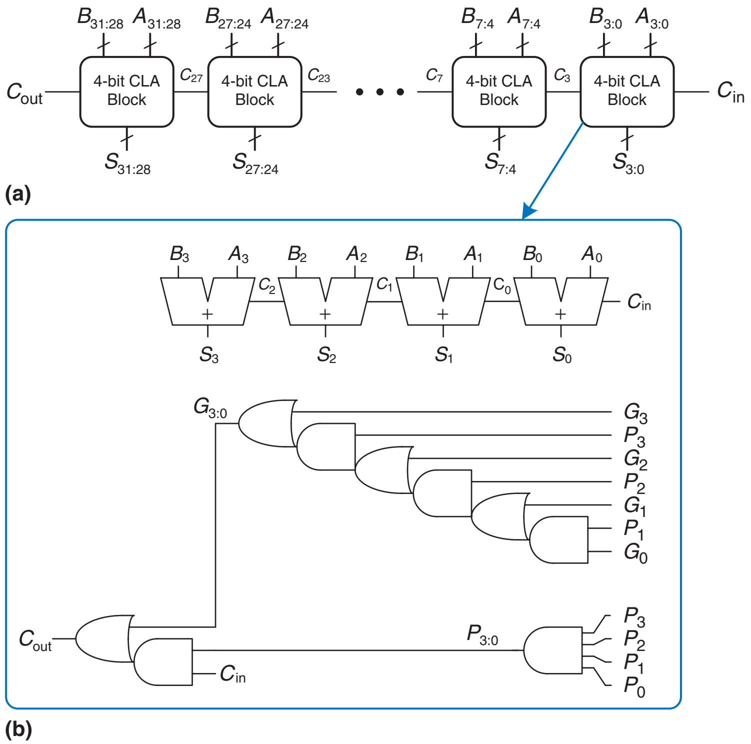


Figure 5.6 (a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block

C_{27}, the carry in to the last block. For a large adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N-bit adder divided into k-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right)t_{\text{AND\_OR}} + kt_{FA} \tag{5.6}$$

where t_{pg} is the delay of the column propagate and generate gates (a single AND or OR gate) to generate P_i and G_i, t_{pg_block} is the delay to find the block propagate and generate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and $t_{\text{AND\_OR}}$ is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block. For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N.

Example 5.1 RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution According to Equation 5.1, the propagation delay of the 32-bit ripple-carry adder is 32 × 300 ps = 9.6 ns.

The CLA has $t_{pg} = 100$ ps, $t_{pg\_\text{block}} = 6 \times 100$ ps $= 600$ ps, and $t_{\text{AND\_OR}} = 2 \times 100$ ps $=$ 200 ps. According to Equation 5.6, the propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus 100 ps + 600 ps + (32/4 − 1) × 200 ps + (4 × 300 ps) = 3.3 ns, almost three times faster than the ripple-carry adder.

Prefix Adder*

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5.7}$$

Define column $i = -1$ to hold C_{in}, so $G_{-1} = C_{\text{in}}$ and $P_{-1} = 0$. Then, $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i - 1$ if the block spanning columns $i - 1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation 5.7 as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5.8}$$

Early computers used ripple-carry adders because components were expensive and ripple-carry adders used the least hardware. Virtually all modern PCs use prefix adders on critical paths because transistors are now cheap and speed is of great importance.

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$,…, $G_{N-2:-1}$. These signals, along with $P_{-1:-1}$,$P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$,…, $P_{N-2:-1}$, are called *prefixes*.

Figure 5.7 shows an $N = 16$-bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits $i{:}k$ and from the lower part spanning bits $k-1{:}j$. It combines these parts to form generate and propagate signals for the entire block spanning bits $i{:}j$ using the equations

$$G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j} \tag{5.9}$$

$$P_{i:j} = P_{i:k}\ P_{k-1:j} \tag{5.10}$$

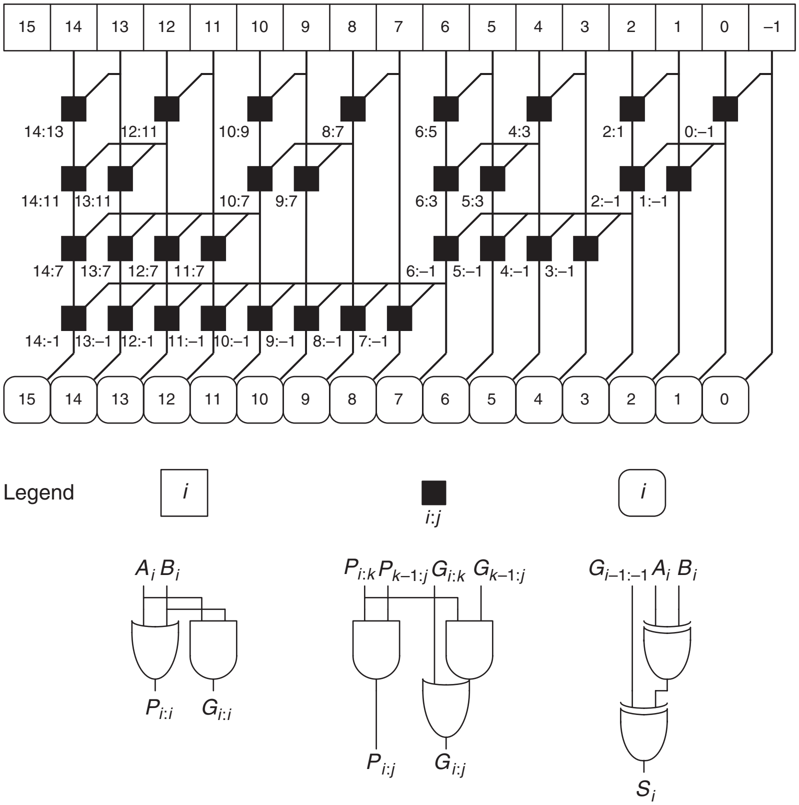


Figure 5.7 **16-bit prefix adder**

In other words, a block spanning bits *i:j* will generate a carry if the upper part (*i*:*k*) generates a carry or if the upper part propagates a carry that is generated in the lower part (*k* – 1:*j*). The block will propagate a carry if both the upper and lower parts propagate the carry. Finally, the prefix adder computes the sums using Equation 5.8.

In summary, the prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. This speedup is significant, especially for adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-lookahead adder. The network of black cells is called a *prefix tree*.

The general principle of using prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique. With some cleverness, it can be applied to many other types of circuits (see, for example, Exercise 5.7).

The critical path for an *N*-bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all of the prefixes. $G_{i\text{-}1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i. Mathematically, the delay of an *N*-bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \tag{5.11}$$

where t_{pg_prefix} is the delay of a black prefix cell.

Notice that the delay of the final (sum) box in the prefix adder is one XOR delay (not two) because the computation of $A_i \oplus B_i$ already occurred earlier because A_i and B_i were available.

Example 5.2 PREFIX ADDER DELAY

Compute the delay of a 32-bit prefix adder. Assume that each two-input gate delay is 100 ps.

Solution The propagation delay of each black prefix cell t_{pg_prefix} is 200 ps (i.e., two gate delays). Thus, using Equation 5.11, the propagation delay of the 32-bit prefix adder is $100\text{ ps} + \log_2(32) \times 200\text{ ps} + 100\text{ ps} = 1.2\text{ ns}$, which is about three times faster than the carry-lookahead adder and eight times faster than the ripple-carry adder from Example 5.1. In practice, the benefits are not quite this great, but prefix adders are still substantially faster than the alternatives.

Putting It All Together

This section introduced the half adder, full adder, and three types of carry propagate adders: ripple-carry, carry-lookahead, and prefix adders. Faster adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate adder for a design.

Hardware description languages provide the + operation to specify a CPA. Modern synthesis tools select among many possible implementations,

HDL Example 5.1 CARRY PROPAGATE ADDER (CPA)

SystemVerilog

```
module adder #(parameter N = 8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(N-1 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result <= (“0” & a) + (“0” & b) + cin;
  s      <= result(N-1 downto 0);
  cout   <= result(N);
end;
```

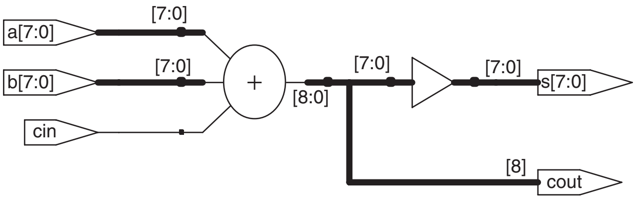


Figure 5.8 Synthesized adder

choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the designer's job. HDL Example 5.1 describes a CPA with carries in and out, and Figure 5.8 shows the resulting hardware.

5.2.2 Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy: flip (i.e., reverse) the sign of the second number, then add. Flipping the sign of a two's complement number is done by inverting the bits and adding 1.

To compute $Y = A - B$, first create the two's complement of B: invert the bits of B to obtain $\overline{B}$ and add 1 to get $-B = \overline{B} + 1$. Add this quantity to A to get $Y = A + \overline{B} + 1 = A - B$. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{in} = 1$. Figure 5.9 shows the symbol for a subtractor and the underlying hardware for performing $Y = A - B$. HDL Example 5.2 describes a subtractor, and Figure 5.10 shows the resulting hardware.

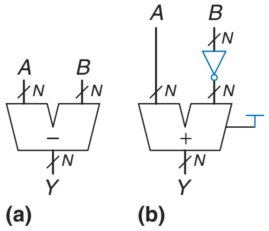


Figure 5.9 Subtractor: (a) symbol, (b) implementation

HDL Example 5.2 SUBTRACTOR

SystemVerilog

```
module subtractor #(parameter N = 8)
                  (input  logic [N-1:0] a, b,
                   output logic [N-1:0] y);

  assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity subtractor is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y:    out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

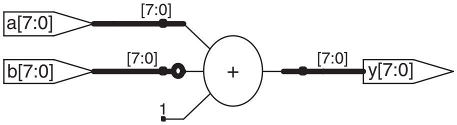


Figure 5.10 Synthesized subtractor

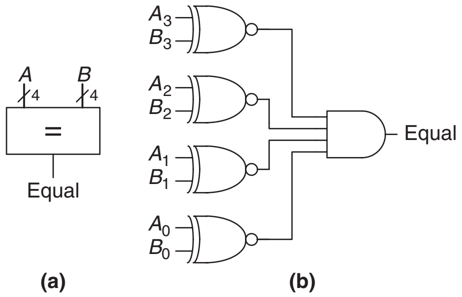


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

5.2.3 Comparators

A *comparator* determines whether two binary numbers are equal or if one is greater or less than the other. A comparator receives two N-bit binary numbers A and B and outputs a 1-bit comparison result.

An *equality comparator* produces a single output, indicating whether A is equal to B ($A == B$). A *magnitude comparator* produces one or more outputs, indicating the relative values of A and B.

The equality comparator is the simpler piece of hardware. Figure 5.11 shows the symbol and implementation of a 4-bit equality comparator. It first checks to determine whether the corresponding bits in each column of A and B are equal, using XNOR gates. The numbers are equal if all of the columns are equal.

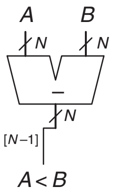


Figure 5.12 ***N*-bit signed comparator**

Magnitude comparison of signed numbers is usually done by computing $A - B$ and looking at the sign (most significant bit) of the result as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise, A is greater than or equal to B. This comparator, however, functions incorrectly upon overflow. Exercises 5.9 and 5.10 explore this limitation and how to fix it. HDL Example 5.3 shows how to use various comparison operations for unsigned numbers, and Figure 5.13 shows the resulting hardware.

HDL Example 5.3 COMPARATORS

SystemVerilog

```
module comparators #(parameter N = 8)
                    (input  logic [N-1:0] a, b,
                     output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity comparators is
  generic(N: integer := 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparators is
begin
  eq  <= ‘1’ when (a = b)  else ‘0’;
  neq <= ‘1’ when (a /= b) else ‘0’;
  lt  <= ‘1’ when (a < b)  else ‘0’;
  lte <= ‘1’ when (a <= b) else ‘0’;
  gt  <= ‘1’ when (a > b)  else ‘0’;
  gte <= ‘1’ when (a >= b) else ‘0’;
end;
```

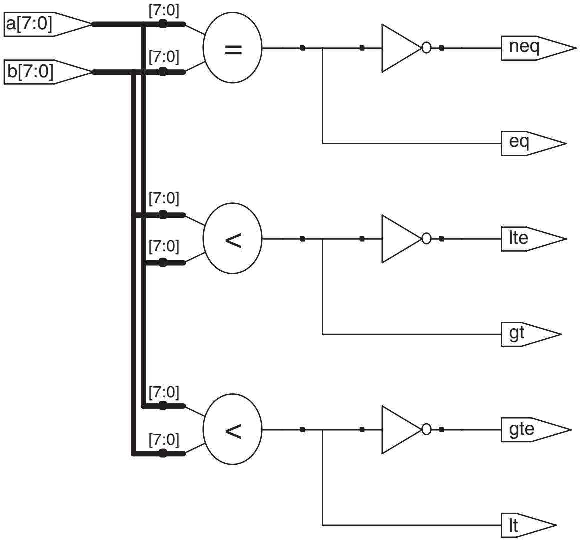


Figure 5.13 **Synthesized comparators**

Table 5.1 ALU operations

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

5.2.4 ALU

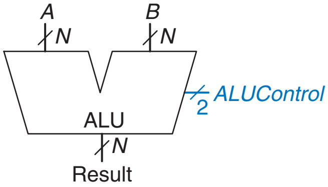


Figure 5.14 ALU symbol

An *Arithmetic/Logical Unit* (*ALU*) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an *N*-bit ALU with *N*-bit inputs and outputs. The ALU receives a 2-bit control signal *ALUControl* that specifies which function to perform. Control signals will generally be shown in blue to distinguish them from the data. Table 5.1 lists typical functions that the ALU can perform.

Figure 5.15 shows an implementation of the ALU. The ALU contains an *N*-bit adder and *N* 2-input AND and OR gates. It also contains inverters

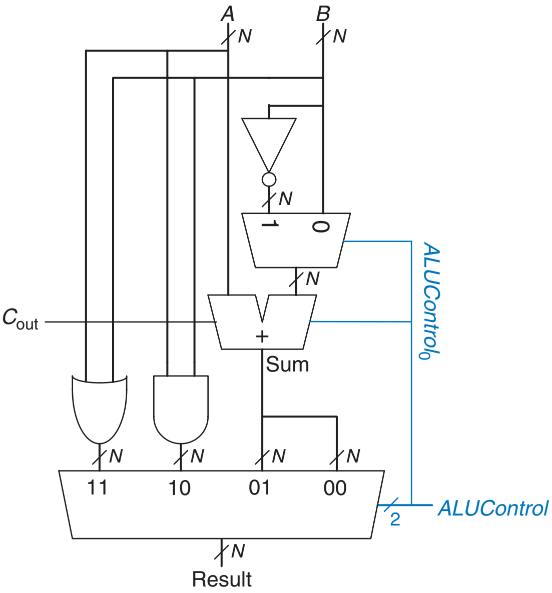


Figure 5.15 *N*-bit ALU

and a multiplexer to invert input B when $ALUControl_0$ is asserted. A 4:1 multiplexer chooses the desired function based on $ALUControl$.

More specifically, if $ALUControl = 00$, the output multiplexer chooses $A + B$. If $ALUControl = 01$, the ALU computes $A - B$. (Recall from Section 5.2.2 that $\overline{B} + 1 = -B$ in two's complement arithmetic. Because $ALUControl_0$ is 1, the adder receives inputs A and $\overline{B}$ and an asserted carry in, causing it to perform subtraction: $A + \overline{B} + 1 = A - B$.) If $ALUControl = 10$, the ALU computes A AND B. If $ALUControl = 11$, the ALU performs A OR B.

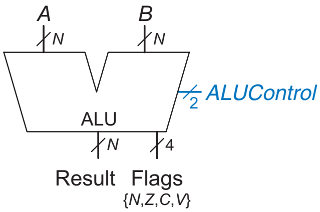


Figure 5.16 ALU symbol with output flags

Some ALUs produce extra outputs, called *flags*, that indicate information about the ALU output. Figure 5.16 shows the ALU symbol with a 4-bit *Flags* output. As shown in the schematic of this ALU in Figure 5.17, the *Flags* output is composed of the N, Z, C, and V flags that indicate, respectively, that the ALU output, *Result*, is negative or zero or that the adder produced a carry out or overflowed. Recall that the most significant bit of a two's complement number is 1 if it is negative and 0 otherwise. Thus, the N (Negative) flag is connected to the most significant bit of the ALU output, $Result_{31}$. The Z (Zero) flag is asserted when all of the bits of *Result* are 0, as detected by the N-bit NOR gate in Figure 5.17. The C (Carry out) flag is asserted when the adder produces a carry out *and* the ALU is performing addition or subtraction (indicated by $ALUControl_1 = 0$).

Overflow detection, as shown on the left side of Figure 5.17, is trickier. Recall from Section 1.4.6 that overflow occurs when the addition of

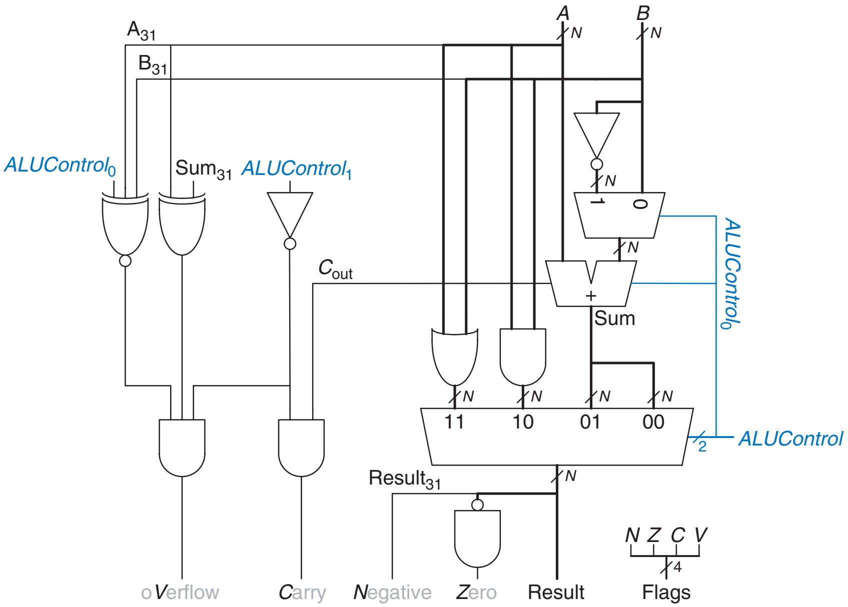


Figure 5.17 *N*-bit ALU with output flags

Table 5.2 Signed and unsigned comparisons

Comparison	Signed	Unsigned
=	Z	Z
≠	$\overline{Z}$	$\overline{Z}$
<	N⊕V	$\overline{C}$
≤	Z + (N⊕V)	Z + $\overline{C}$
>	$\overline{Z} \bullet (\overline{N \oplus V})$	$\overline{Z} \bullet C$
≥	$(\overline{N \oplus V})$	C

two same signed numbers produces a result with the opposite sign. So, V (oVerflow) is asserted when all three of the following conditions are true: (1) the ALU is performing addition or subtraction ($ALUControl_1 = 0$), (2) A and *Sum* have opposite signs, as detected by the XOR gate, and (3) overflow is possible. That is, as detected by the XNOR gate, either A and B have the same sign and the adder is performing addition ($ALUControl_0 = 0$) or A and B have opposite signs and the adder is performing subtraction ($ALUControl_0 = 1$). The 3-input AND gate detects when all three conditions are true and asserts V.

The ALU flags can also be used for comparisons, as shown in Table 5.2. To compare inputs A and B, the ALU computes $A - B$ and looks at the flags. If Z is asserted, the result is 0, so $A = B$. Otherwise, A is not equal to B.

Magnitude comparison is messier and depends on whether the numbers are signed or unsigned. For example, to determine $A < B$, we compute $A - B$ and check whether the result is negative. If the numbers are unsigned, the result is negative if there is no carry out.[1] If the numbers are signed, we can't rely on the carry because small negative numbers are represented in the same way as large positive unsigned numbers. Instead, we simply compute $A - B$ and see whether the answer is negative, indicated by the N flag. However, if overflow occurs, the N flag will be incorrect. Hence, A is less than B if $N \oplus V$ (in other words, if the answer is negative and there is no overflow or if the answer is positive but overflow occurred). In summary, let us define L (the Less than signal) to be true if $A < B$. For unsigned numbers, $L = \overline{C}$. For signed

[1] You can check this by trying some numbers. Alternatively, note that reversing the sign (i.e., taking the two's complement) of N-bit numbers for subtraction produces $-B = \overline{B} + 1 = 2^N - B$. Now, $A + (-B) = 2^N + A - B$. This will have a carry out (a 1 in column N) if $A \geq B$ and, hence, no carry out if $A < B$.

numbers, $L = N \oplus V$. The remainder of the checks are easier. Less than or equal to (≤) is L OR Z because L indicates less than and Z indicates equal to. Greater than or equal to (≥) is the opposite of less than: $\overline{L}$. Greater than (>) is indicated by greater than or equal to, but not equal: $\overline{L}$ AND $\overline{Z}$.

Example 5.3 COMPARISONS

Consider 4-bit numbers $A = 1111$ and $B = 0010$. Determine whether $A < B$, first interpreting the numbers as unsigned (15 and 2) and then as signed (−1 and 2).

Solution Compute $A - B = A + \overline{B} + 1 = 1111 + 1101 + 1 = 1\mathit{1}101$. The carry out C is 1, as shown in blue. The N flag is 1, as shown in italics. The V flag is 0 because the result has the same sign bit as A. The Z flag is 0 because the answer is not 0000.

For unsigned comparison, $L = {\sim}C = 0$ because 15 is not less than 2. For signed comparison, $L = N \oplus V = 1$ because −1 is less than 2.

Certain ALUs also implement an instruction called *set if less than* (SLT). When $A < B$, $Result = 1$. Otherwise, $Result = 0$. This is convenient for computers that do not have access to ALU flags because it essentially stores flag information in the result. SLT typically treats inputs as signed. Another flavor (SLTU) treats inputs as unsigned. Many variations on this basic ALU exist that support other functions, such as NOT, XOR, or XNOR. The HDL for an N-bit ALU, including versions that support SLT and output flags, is left to Exercises 5.11 to 5.14.

Example 5.4 EXPANDING THE ALU TO HANDLE SLT

Expand the ALU to handle the set if less than (SLT) operation.

Solution To add another function to the ALU, we must expand the multiplexer to have five inputs. We determine if A is less than B by performing $A - B$; if the result is negative, A is less than B. Table 5.3 shows the updated $ALUControl$ signal for handling SLT, and Figure 5.18(a) shows the expanded circuit, with changes highlighted in blue (for control signals) and black. We use $ALUControl = 101$ for the SLT operation and take advantage of the fact that making $ALUControl_0 = 1$ causes the adder to perform $A - B$. When $Sum_{N-1} = 1$, the result of A − B is negative, and A is less than B. So, we zero-extend Sum_{N-1} and feed it into the 101 multiplexer input to complete the SLT operation. Note, however, that this implementation does not account for overflow. When overflow occurs, Sum will have the incorrect sign. So, we XOR the sign bit of Sum with V, the overflow signal, to correctly indicate a negative Sum, as shown in Figure 5.18(b).

Table 5.3 **ALU expanded operation for SLT**

$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

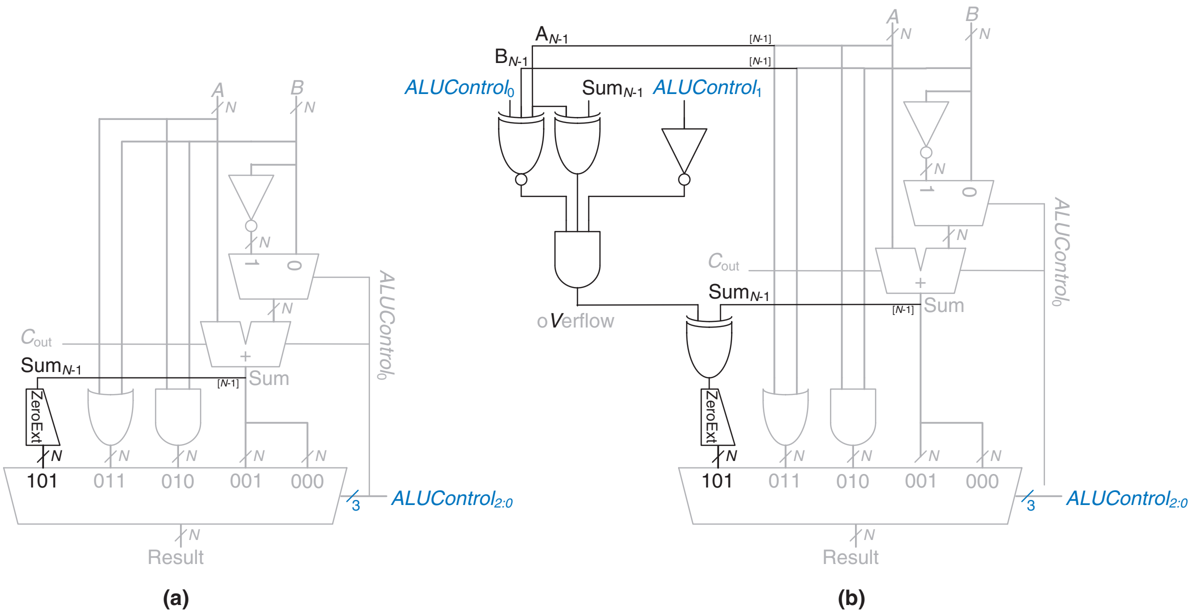


Figure 5.18 ALU expanded to support SLT (a) not accounting for overflow, (b) accounting for overflow

5.2.5 Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. As the name implies, a shifter shifts a binary number left or right by a specified number of positions. Several kinds of commonly used shifters exist:

- **Logical shifter**—shifts the number to the left or right and fills empty spots with 0's.

  Example: 11001 >> 2 = 00110; 11001 << 2 = 00100

- **Arithmetic shifter**—is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit (msb). This is useful for multiplying and dividing signed numbers (see Sections 5.2.6 and 5.2.7). Arithmetic shift left is the same as logical shift left.

  Example: 11001 >>> 2 = 11110; 11001 << 2 = 00100. The operators <<, >>, and >>> typically indicate shift left, logical shift right, and arithmetic shift right, respectively.

- **Rotator**—rotates a number in a circle such that empty spots are filled with bits shifted off the other end.

  Example: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111. ROR = rotate right; ROL = rotate left.

An *N*-bit shifter can be built from *N* *N*:1 multiplexers. The input is shifted by 0 to $N-1$ bits, depending on the value of the $\log_2 N$-bit select lines. Figure 5.19 shows the symbol and hardware of 4-bit shifters. Depending on the value of the 2-bit shift amount $shamt_{1:0}$, the output *Y* receives the input *A* shifted by 0 to 3 bits. For all shifters, when $shamt_{1:0} = 00$, $Y = A$. Exercise 5.22 covers rotator designs.

A left shift is a special case of multiplication. A left shift by *N* bits multiplies the number by 2^N. For example, $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \times 2^4 = 48_{10}$.

An arithmetic right shift is a special case of division. An arithmetic right shift by *N* bits divides the number by 2^N. For example, 11100_2 >>> $2 = 11111_2$ is equivalent to $-4_{10}/2^2 = -1_{10}$.

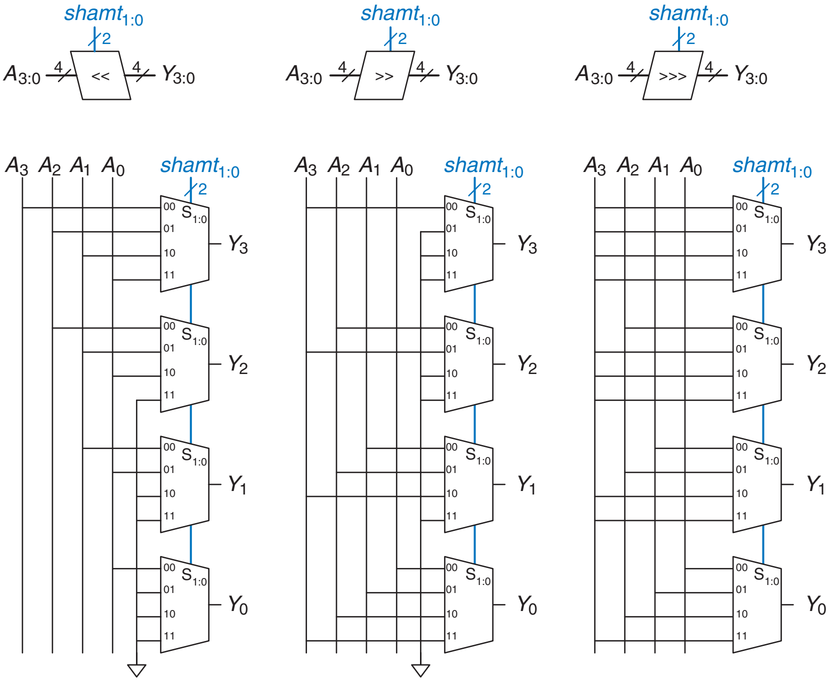


Figure 5.19 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

```
       230   multiplicand       0101
   ×    42   multiplier       × 0111
   -------                    --------
       460    partial           0101
   +  920     products         0101
   -------                    0101
      9660                  + 0000
                              --------
               result        0100011

230 × 42 = 9660                5 × 7 = 35
(a)                            (b)
```

Figure 5.20 **Multiplication: (a) decimal, (b) binary**

5.2.6 Multiplication*

Multiplication of unsigned binary numbers is similar to decimal multiplication but involves only 1's and 0's. Figure 5.20 compares multiplication in decimal and binary. In both cases, *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result.

In general, an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N$-bit result. The partial products in binary multiplication are either the multiplicand or all 0's. Multiplication of 1-bit binary numbers is equivalent to the AND operation, so AND gates are used to form the partial products.

Signed and unsigned multiplication differ. For example, consider 0xFE × 0xFD. If these 8-bit numbers are interpreted as signed integers, they represent −2 and −3, so the 16-bit product is 0x0006. If these numbers are interpreted as unsigned integers, the 16-bit product is 0xFB06. Notice that, in either case, the least significant byte is 0x06.

Figure 5.21 shows the symbol, function, and implementation of an unsigned 4 × 4 multiplier. The unsigned multiplier receives the multiplicand and multiplier, A and B, and produces the product P. Figure 5.21(b) shows how partial products are formed. Each partial product is a single multiplier bit (B_3, B_2, B_1, or B_0) AND the multiplicand bits (A_3, A_2, A_1, A_0). With N-bit operands, there are N partial products and $N - 1$ stages of 1-bit adders. For example, for a 4 × 4 multiplier, the partial product of the first row is B_0 AND (A_3, A_2, A_1, A_0). This partial product is added to the shifted second partial product, B_1 AND (A_3, A_2, A_1, A_0). Subsequent rows of AND gates and adders form and add the remaining partial products.

The HDL for signed and unsigned multipliers is in HDL Example 4.33. As with adders, many different multiplier designs with different speed/cost trade-offs exist. Synthesis tools may pick the most appropriate design given the timing constraints.

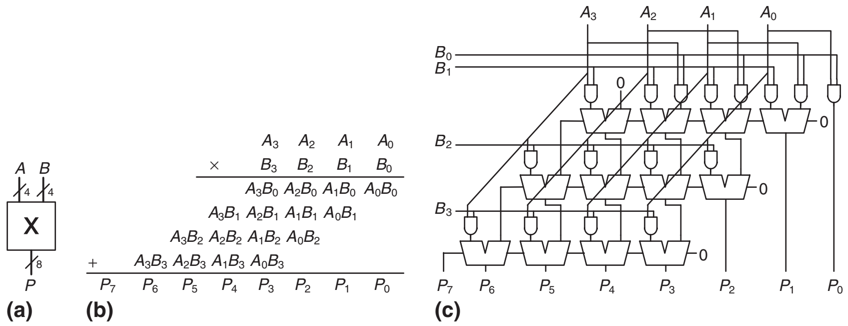

A
B
4
4
X
8
P
(a)
A_3 A_2 A_1 A_0
× B_3 B_2 B_1 B_0
A_3B_0 A_2B_0 A_1B_0 A_0B_0
A_3B_1 A_2B_1 A_1B_1 A_0B_1
A_3B_2 A_2B_2 A_1B_2 A_0B_2
+ A_3B_3 A_2B_3 A_1B_3 A_0B_3
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
(b)
A_3 A_2 A_1 A_0
B_0
B_1
B_2
B_3
0
0
0
0
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
(c)

Figure 5.21 **4 × 4 multiplier: (a) symbol, (b) function, (c) implementation**

A *multiply accumulate* operation multiplies two numbers and adds them to a third number—typically, the accumulated value. These operations, also called MACs, are often used in *digital signal processing* (DSP) algorithms such as the Fourier transform, which requires a summation of products.

5.2.7 Division*

Binary division can be performed using the following algorithm for N-bit unsigned numbers in the range $[0, 2^N - 1]$:

```
R′ = 0
for i = N−1 to 0
   R = {R′ << 1, $A_i$}
   D = R − B
   if D < 0 then $Q_i$ = 0, R′ = R   // R < B
   else          $Q_i$ = 1, R′ = D   // R ≥ B
R = R′
```

The *partial remainder* R is initialized to 0 ($R' = 0$), and the most significant bit of the dividend A becomes the least significant bit of R ($R = \{R' << 1, Ai\}$). The divisor B is subtracted from this partial remainder to determine whether it fits ($D = R - B$). If the difference D is negative (i.e., the sign bit of D is 1), then the quotient bit Q_i is 0 and the difference is discarded. Otherwise, Q_i is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R, and the process repeats. The result satisfies $\frac{A}{B} = Q + \frac{R}{B}$.

Figure 5.22 shows a schematic of a 4-bit array divider. The divider computes A/B and produces a quotient Q and a remainder R. The legend shows the symbol and schematic for each block in the array

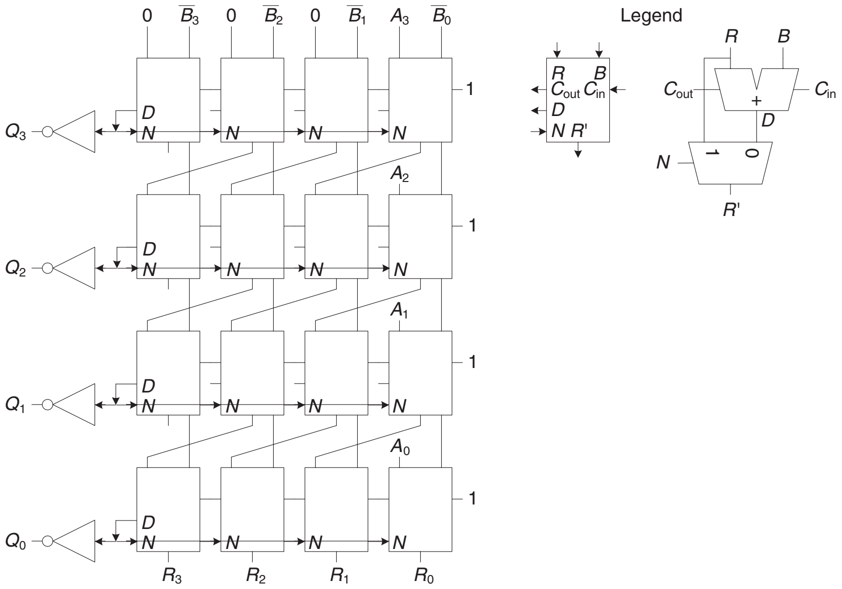


Figure 5.22 Array divider

divider. Each row performs one iteration of the division algorithm. Specifically, each row calculates the difference $D = R - B$. (Recall that $R + \overline{B} + 1 = R - B$). The multiplexer select signal, N (for Negative), receives 1 when a row's difference D is negative. So N is driven by the most significant bit of D, which is 1 when the difference is negative. Each quotient bit (Q_i) is 0 when D is negative and 1 otherwise. The multiplexer passes R to the next row if the difference is negative and D otherwise. The following row shifts the new partial remainder left by one bit, appends the next most significant bit of A, and then repeats the process.

The delay of an N-bit array divider increases proportionally to N^2 because the carry must ripple through all N stages in a row before the sign is determined and the multiplexer selects R or D. This repeats for all N rows. Division is a slow and expensive operation in hardware; therefore, it should be used as infrequently as possible.

5.2.8 Further Reading

Computer arithmetic could be the subject of an entire text. *Digital Arithmetic*, by Ercegovac and Lang, is an excellent overview of the entire field. *CMOS VLSI Design*, by Weste and Harris, covers high-performance circuit designs for arithmetic operations.

5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

5.3.1 Fixed-Point Number Systems

(a) 01101100

(b) 0110.1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Figure 5.23 Fixed-point notation of 6.75 with four integer bits and four fraction bits

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. For example, Figure 5.23(a) shows a fixed-point number with four integer bits and four fraction bits. Figure 5.23(b) shows the implied binary point in blue, and Figure 5.23(c) shows the equivalent decimal value.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.24 Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement

Signed fixed-point numbers can use either two's complement or sign/magnitude notation. Figure 5.24 shows the fixed-point representation of −2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

Like all binary number representations, fixed-point numbers are just a collection of bits. There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number.

In general, we use *Ua.b* to denote an unsigned fixed-point number with *a* integer bits and *b* fraction bits. *Qa.b* denotes a signed (two's complement) fixed point number with *a* integer bits (including the sign bit) and *b* fractional bits.

Example 5.5 ARITHMETIC WITH FIXED-POINT NUMBERS

Compute 0.75 + (−0.625) using Q4.4 fixed-point numbers.

Solution First, convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because

```
   0000.1010   Binary Magnitude
   1111.0101   One's Complement
+          1   Add 1
   ---------
   1111.0110   Two's Complement
```

Figure 5.25 **Fixed-point two's complement conversion**

```
   0000.1100          0.75
+  1111.0110     +  (–0.625)
  ----------     -----------
  10000.0010          0.125
(a)              (b)
```

Figure 5.26 **Addition: (a) binary fixed-point, (b) decimal equivalent**

$0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$.

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.25 shows the conversion of –0.625 to fixed-point two's complement notation.

Figure 5.26 shows the fixed-point binary addition and the decimal equivalent for comparison. Note that the leading 1 in the binary fixed-point addition of Figure 5.26(a) is discarded from the 8-bit result.

Fixed-point number systems are commonly used in digital signal processing (DSP), graphics, and machine learning applications because the computations are faster and consume less power than they would in floating-point systems. Q1.15 (also known as Q15) is the most common format, storing signed numbers in the range (–1, 1) with 15 bits of precision. Q1.31 (also called just Q31) is sometimes used for higher-precision intermediate results, such as in a Fast Fourier Transform. U8.8 is sometimes used for sensor readings sampled by analog/digital converters (ADCs). Note that all of these formats pack into 16- or 32-bit words for efficient storage in computer memories, which are typically a power of 2 in width.

5.3.2 Floating-Point Number Systems*

Floating-point numbers are analogous to scientific notation. They circumvent the limitation of having a constant number of integer and fraction bits, allowing the representation of very large and very small numbers. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.27. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. The decimal point *floats* to the position right after the most significant digit. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

$$\pm M \times B^E$$

Figure 5.27 **Floating-point numbers**

Example 5.6 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution First, convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.28 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

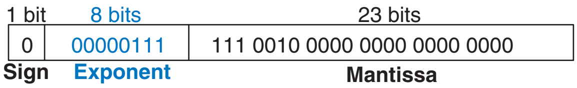


Figure 5.28 **32-bit floating-point version 1**

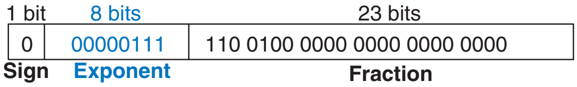


Figure 5.29 **32-bit floating-point version 2**

In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.29 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

As may be apparent, there are many reasonable ways to represent floating-point numbers. For many years, computer manufacturers used incompatible floating-point formats. Results from one computer could not directly be interpreted by another computer.

The Institute of Electrical and Electronics Engineers (IEEE) solved this problem by creating the *IEEE 754 Floating-Point Standard* in 1985, that defines floating-point numbers. This floating-point format is now almost universally used and is the one discussed in this section.

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a *biased* exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. For example, for the exponent 7, the biased exponent is $7 + 127 = 134 = 10000110_2$. For the exponent −4, the biased exponent is: $-4 + 127 = 123 = 01111011_2$. Figure 5.30 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of 134 (7 + 127). This notation conforms to the IEEE 754 floating-point standard.

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all l's are reserved for these special cases. Table 5.4 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Single-, Double-, and Quad-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called *single-precision*, *single*, or *float*. The IEEE 754 standard also

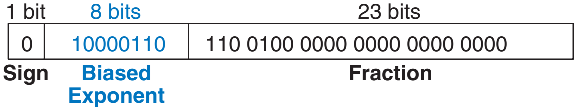


Figure 5.30 **IEEE 754 floating-point notation**

Table 5.4 IEEE 754 floating-point notations for 0, ±∞, and NaN

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Table 5.5 Floating-point formats

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits	Bias
Single	32	1	8	23	127
Double	64	1	11	52	1023
Quad	128	1	15	112	16363

defines 64-bit *double-precision* numbers (also called *doubles*) and 128-bit *quadruple-precision* numbers (also called *quads*) that provide greater precision and greater range. Table 5.5 shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$. They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits. Quads have 34 decimal digits of precision but are not yet widely supported in hardware or software.

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest. In the round-to-nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round-to-nearest mode, overflows are rounded up to ±∞ and underflows are rounded down to 0.

Floating-point cannot represent some numbers exactly, such as 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999.... To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example, the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A–F encodings are not used) and, thus, decreased performance. So for compute-intensive applications, floating-point is much faster.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

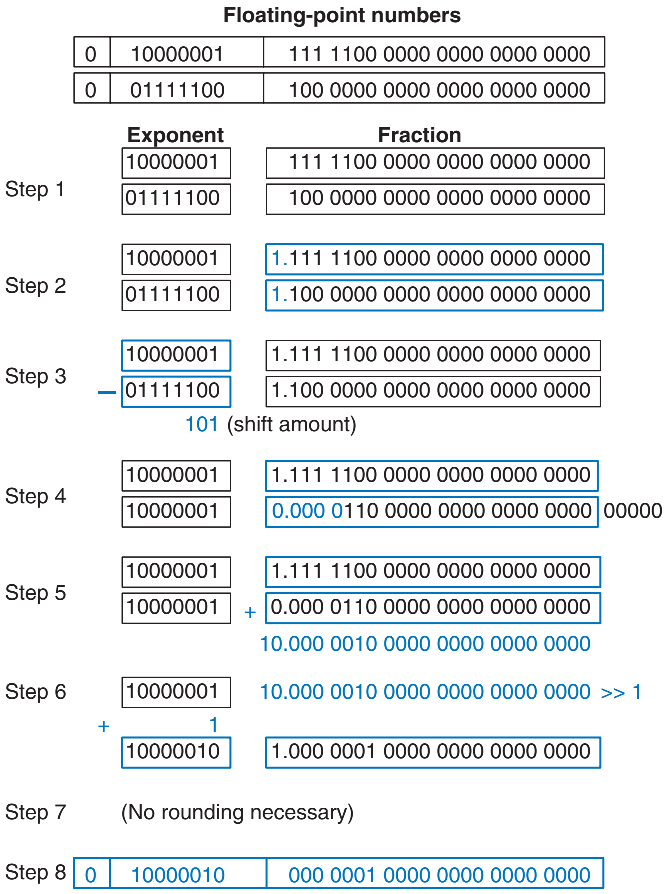


Figure 5.31 Floating-point addition

Figure 5.31 shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}). The result is 8.0625 (1.0000001×2^3). After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the *floating-point unit* (*FPU*), is typically distinct from the *central processing unit* (*CPU*). The infamous *floating-point division* (*FDIV*) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly.

5.4 SEQUENTIAL BUILDING BLOCKS

This section examines sequential building blocks, including counters and shift registers.

5.4.1 Counters

An N-bit *binary counter*, shown in Figure 5.32, is a sequential arithmetic circuit with clock and reset inputs and an N-bit output Q. *Reset* initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

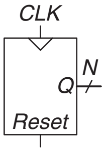


Figure 5.32 Counter symbol

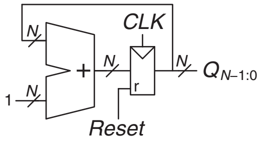


Figure 5.33 *N*-bit counter

Figure 5.33 shows an N-bit counter composed of an adder and a resettable register. On each cycle, the counter adds 1 to the value stored in the register. HDL Example 5.4 shows a binary counter with asynchronous reset, and Figure 5.34 shows the resulting hardware.

The most significant bit of an N-bit counter toggles every 2^N cycles. Thus, it reduces the frequency of the clock by a factor of 2^N. This is called a *divide-by-*2^N counter and is useful for slowing down fast signals. For example, if a digital system has a 50 MHz internal clock, you can use a 24-bit counter to produce a (50×10^6 Hz/2^{24}) = 2.98 Hz signal that blinks a light-emitting diode (LED) at a rate the human eye can observe.

A further counter generalization to produce arbitrary frequencies is called a *digitally controlled oscillator* (DCO, Example 5.7). Consider an N-bit counter that adds p on each cycle, rather than 1. If the counter receives a clock with frequency f_{clk}, the most significant bit now toggles at $f_{\text{out}} = f_{clk} \times p/2^N$. With a judicious choice of p and N, you can produce an output of any frequency. Larger N gives more precise control at the expense of more hardware.

HDL Example 5.4 COUNTER

SystemVerilog

```
module counter #(parameter N = 8)
                (input  logic         clk,
                 input  logic         reset,
                 output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity counter is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       q:          out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process(clk, reset) begin
    if reset then                q <= (OTHERS => ‘0’);
    elsif rising_edge(clk) then q <= q + ‘1’;
    end if;
  end process;
end;
```

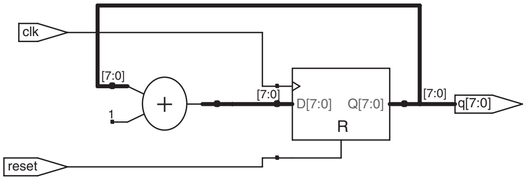


Figure 5.34 Synthesized counter

Example 5.7 DIGITALLY CONTROLLED OSCILLATOR

Suppose you have a 50 MHz clock and want to produce a 500 Hz output. Consider using an N = 24- or 32-bit counter. What value of p should you choose and how close can you come to 500 Hz?

Solution We want $p/2^N$ = 500 Hz/50 MHz = 0.001. If N = 24, choose p = 168 to get f_{out} = 500.68 Hz. If N = 32, choose p = 42950 to get f_{out} = 500.038 Hz.

Other types of counters, such as Up/Down counters, are explored in Exercises 5.51 through 5.54.

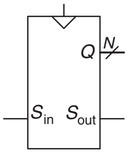


Figure 5.35 Shift register symbol

5.4.2 Shift Registers

A *shift register* has a clock, a serial input S_{in}, a serial output S_{out}, and N parallel outputs $Q_{N-1:0}$, as shown in Figure 5.35. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}.

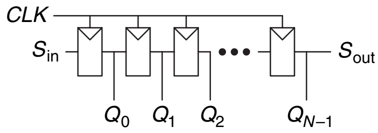


Figure 5.36 Shift register schematic

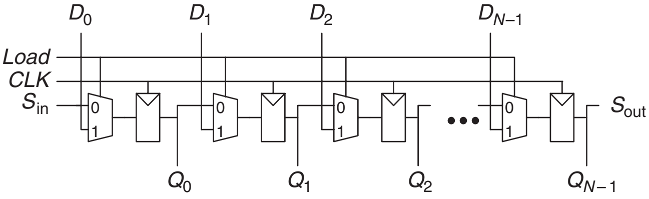


Figure 5.37 Shift register with parallel load

Shift registers can be viewed as *serial-to-parallel converters*. The input is provided serially (one bit at a time) at S_{in}. After N cycles, the past N inputs are available in parallel at Q.

A shift register can be constructed from N flip-flops connected in series, as shown in Figure 5.36. Some shift registers also have a reset signal to initialize all of the flip-flops.

A related circuit is a *parallel-to-serial* converter that loads N bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both serial-to-parallel and parallel-to-serial operations by adding a parallel input $D_{N-1:0}$ and a control signal *Load*, as shown in Figure 5.37. When *Load* is asserted, the flip-flops are loaded in parallel from the D inputs. Otherwise, the shift register shifts normally. HDL Example 5.5 describes such a shift register, and Figure 5.38 shows the resulting hardware.

Scan Chains*

Shift registers are often used to test sequential circuits, using a technique called *scan chains*. Testing combinational circuits is relatively straightforward. Known inputs called *test vectors* are applied, and the outputs are checked against the expected result. Testing sequential circuits is more difficult because the circuits have state. Starting from a known initial condition, a large number of cycles of test vectors may be needed to put the circuit into a desired state. For example, testing that the most significant bit of a 32-bit counter advances from 0 to 1 requires resetting the counter, then applying 2^{31} (about two billion) clock pulses!

To solve this problem, designers like to be able to directly observe and control all of the machine's state. This is done by adding a test mode in which the contents of all flip-flops can be read out or loaded with desired values. Most systems have too many flip-flops to dedicate individual pins to read and write each flip-flop. Instead, all flip-flops in the system are connected together into a shift register called a *scan chain*. In

Don't confuse *shift registers* with the *shifters* from Section 5.2.5. Shift registers are sequential logic blocks that shift in a new bit on each clock edge. Shifters are unclocked combinational logic blocks that shift an input by a specified amount.

HDL Example 5.5 SHIFT REGISTER WITH PARALLEL LOAD

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic         clk,
                  input  logic         reset, load,
                  input  logic         sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic         sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(N-1 downto 0);
       q:          out STD_LOGIC_VECTOR(N-1 downto 0);
       sout:       out STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process(clk, reset) begin
    if reset = ‘1’ then q <= (OTHERS => ‘0’);
    elsif rising_edge(clk) then
      if load then      q <= d;
      else              q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;

  sout <= q(N-1);
end;
```

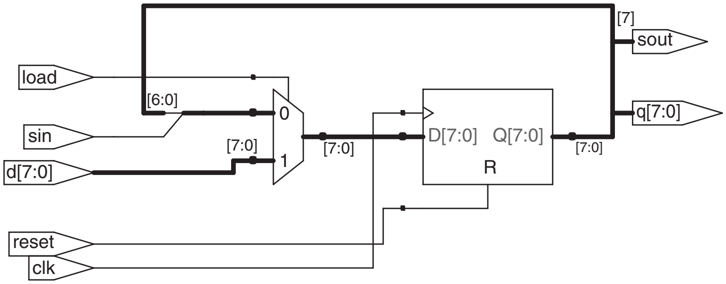


Figure 5.38 Synthesized shiftreg

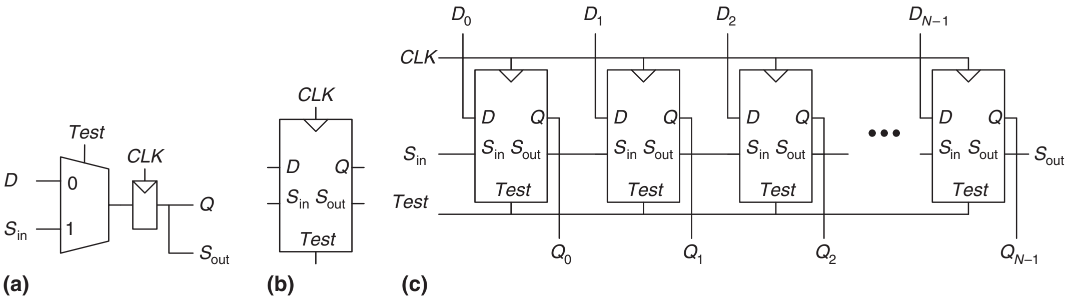


Figure 5.39 Scannable flip-flop: (a) schematic, (b) symbol, and (c) *N*-bit scannable register

normal operation, the flip-flops load data from their D input and ignore the scan chain. In test mode, the flip-flops serially shift their contents out and shift in new contents using S_{in} and S_{out}. The load multiplexer is usually integrated into the flip-flop to produce a *scannable flip-flop*. Figure 5.39 shows the schematic and symbol for a scannable flip-flop and illustrates how the flops are cascaded to build an N-bit scannable register.

For example, the 32-bit counter could be tested by shifting in the pattern 011111…111 in test mode, counting for one cycle in normal mode, then shifting out the result, which should be 100000…000. This requires only 32 + 1 + 32 = 65 cycles.

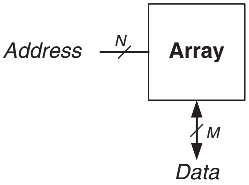


Figure 5.40 Generic memory array symbol

5.5 MEMORY ARRAYS

The previous sections introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits. Registers built from flip-flops are a kind of memory that stores small amounts of data. This section describes *memory arrays* that can efficiently store large amounts of data.

We begin with an overview describing characteristics shared by all memory arrays and then introduce three types of memory arrays: dynamic random access memory (DRAM), static random access memory (SRAM), and read only memory (ROM). Each memory differs in the way it stores data. We briefly discuss area and delay trade-offs and show how memory arrays are used, not only to store data but also to perform logic functions. We then finish this section by showing the HDL for a memory array.

5.5.1 Overview

Figure 5.40 shows a generic symbol for a memory array. The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is specified by an *address*. The value read or written is called *data*. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is called a *word*. Thus, the array contains 2^N M-bit words.

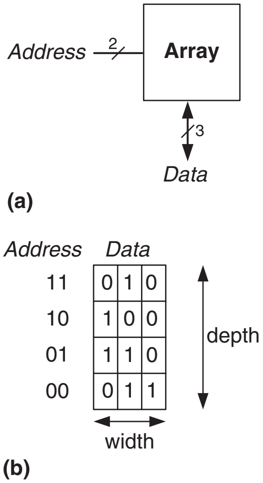


Figure 5.41 4 × 3 memory array: (a) symbol, (b) function

Figure 5.41 shows a memory array with two address bits and three data bits. The two address bits specify one of the four rows (data words) in the array. Each data word is three bits wide. Figure 5.41(b) shows some possible contents of the memory array.

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the *word size*. The size of an array is given as *depth* × *width*. Figure 5.41 is a 4-word × 3-bit array, or simply 4 × 3 array. The symbol for a 1024-word × 32-bit array is shown in Figure 5.42. The total size of this array is 32 kilobits (Kb), also referred to as 32 kibibits (Kib).

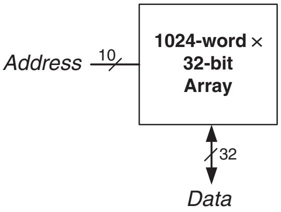


Figure 5.42 32-Kb array: depth = 2^{10} = 1024 words, width = 32 bits

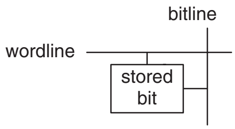


Figure 5.43 **Bit cell**

Bit Cells

Memory arrays are built as an array of *bit cells*, each of which stores 1 bit of data. Figure 5.43 shows that each bit cell is connected to a *wordline* and a *bitline*. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. The circuitry to store the bit varies with memory type.

To read a bit cell, the bitline is initially left floating (Z). Then, the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then, the wordline is turned ON, connecting the bitline to the stored bit. The strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

Organization

Figure 5.44 shows the internal organization of a 4 × 3 memory array. Of course, practical memories are much larger, but the behavior of larger arrays can be extrapolated from the smaller array. In this example, the array stores the data from Figure 5.41(b).

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitlines HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first and then a wordline is asserted, allowing the bitline values to be stored in that row of bit cells. For example, to read *Address* 10, the bitlines are left floating, the decoder asserts *wordline*$_2$, and the data stored in that row of bit cells (100) reads out onto the *Data* bitlines. To write the value 001 to

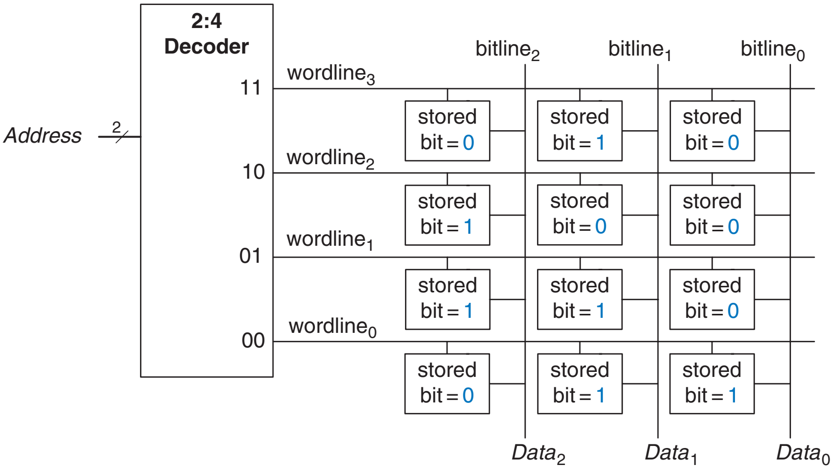


Figure 5.44 **4 × 3 memory array**

Address 11, the bitlines are driven to the value 001, then *wordline*$_3$ is asserted and the new value (001) is stored in the bit cells.

Memory Ports

All memories have one or more *ports*. Each port gives read and/or write access to one memory address. The previous examples were all single-ported memories.

Multiported memories can access several addresses simultaneously. Figure 5.45 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address *A1* onto the read data output *RD1*. Port 2 reads the data from address *A2* onto *RD2*. Port 3 writes the data from the write data input *WD3* into address *A3* on the rising edge of the clock if the write enable *WE3* is asserted.

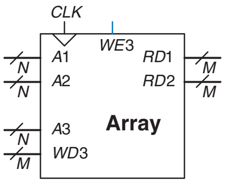


Figure 5.45 **Three-ported memory**

Memory Types

Memory arrays are specified by their size (depth × width) and the number and type of ports. All memory arrays store data as an array of bit cells, but they differ in how they store bits.

Memories are classified based on how they store bits in the bit cell. The broadest classification is *random access memory* (*RAM*) versus *read only memory* (*ROM*). RAM is *volatile*, meaning that it loses its data when the power is turned off. ROM is *nonvolatile*, meaning that it retains its data indefinitely, even without a power source.

RAM and ROM received their names for historical reasons that are no longer very meaningful. RAM is called *random* access memory because any data word is accessed with the same delay as any other. In contrast, a *sequential* access memory, such as a tape recorder, accesses nearby data more quickly than faraway data (e.g., at the other end of the tape). ROM is called *read only* memory because, historically, it could only be read but not written. These names are confusing, because ROMs are also randomly accessed. Worse yet, most modern ROMs can be written as well as read! The important distinction to remember is that RAMs are volatile and ROMs are nonvolatile.

The two major types of RAMs are *dynamic RAM* (*DRAM*) and *static RAM* (*SRAM*). Dynamic RAM stores data as a charge on a capacitor, whereas static RAM stores data using a pair of cross-coupled inverters. There are many flavors of ROMs that vary by how they are written and erased. These various types of memories are discussed in the subsequent sections.

Robert Dennard, 1932–
Invented DRAM in 1966 at IBM. Although many were skeptical that the idea would work, by the mid-1970's DRAM was in virtually all computers. He claims to have done little creative work until, arriving at IBM, they handed him a patent notebook and said, "put all your ideas in there." Since 1965, he has received 35 patents in semiconductors and microelectronics. (Photo courtesy of IBM.)

5.5.2 Dynamic Random Access Memory (DRAM)

DRAM, pronounced "dee-ram," stores a bit as the presence or absence of charge on a capacitor. Figure 5.46 shows a DRAM bit cell. The bit value is stored on a capacitor. The nMOS transistor behaves as a switch

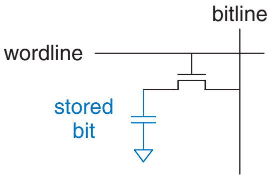


Figure 5.46 **DRAM bit cell**

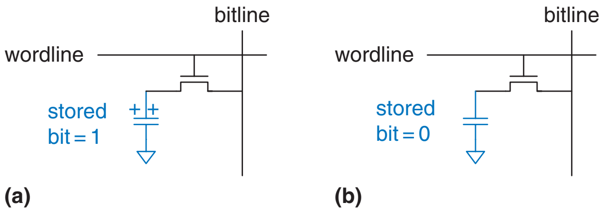


Figure 5.47 DRAM stored values

that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.

As shown in Figure 5.47(a), when the capacitor is charged to V_{DD}, the stored bit is 1; when it is discharged to GND (Figure 5.47(b)), the stored bit is 0. The capacitor node is *dynamic* because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND.

Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to the capacitor. Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

5.5.3 Static Random Access Memory (SRAM)

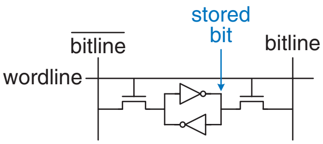


Figure 5.48 SRAM bit cell

SRAM, pronounced "es-ram," is *static* because stored bits do not need to be refreshed. Figure 5.48 shows an SRAM bit cell. The data bit is stored on cross-coupled inverters like those described in Section 3.2. Each cell has two outputs, bitline and $\overline{\text{bitline}}$. When the wordline is asserted, both nMOS transistors turn on, and data values are transferred to or from the bitlines. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

5.5.4 Area and Delay

Flip-flops, SRAMs, and DRAMs are all volatile memories, but each has different area and delay characteristics. Table 5.6 shows a comparison of these three types of volatile memory. The data bit stored in a flip-flop is available immediately at its output. But flip-flops take at least 20 transistors to build. Generally, the more transistors a device has, the more area, power, and cost it requires. DRAM latency is longer than that of SRAM because its bitline is not actively driven by a transistor. DRAM must wait for charge to move (relatively) slowly from the capacitor to the bitline. DRAM also fundamentally has lower throughput

Table 5.6 **Memory comparison**

Memory Type	Transistors per Bit Cell	Latency
Flip-flop	~20	Fast
SRAM	6	Medium
DRAM	1	Slow

than SRAM, because it must refresh data periodically and after a read. DRAM technologies such as *synchronous DRAM* (*SDRAM*) and *double data rate* (*DDR*) SDRAM have been developed to overcome this problem. SDRAM uses a clock to pipeline memory accesses. DDR SDRAM, sometimes called simply DDR, uses both the rising and falling edges of the clock to access data, thus doubling the throughput for a given clock speed. DDR was first standardized in 2000 and ran at 100 to 200 MHz. Later standards, DDR2, DDR3, and DDR4, increased the clock speeds, with speeds in 2021 being over 3 GHz.

Memory latency and throughput also depend on memory size; larger memories tend to be slower than smaller ones if all else is the same. The best memory type for a particular design depends on the speed, cost, and power constraints.

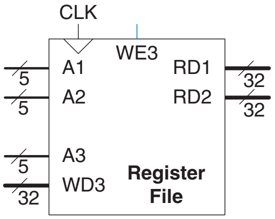


Figure 5.49 **32 × 32 register file with two read ports and one write port**

5.5.5 Register Files

Digital systems often use a number of registers to store temporary variables. This group of registers, called a *register file*, is usually built as a small, multiported SRAM array because it is more compact than an array of flip-flops. In some register files, a particular entry, such as register 0, is hardwired to always read the value 0 because 0 is a commonly used constant.

Figure 5.49 shows a 32-register × 32-bit three-ported register file built from the three-ported memory of Figure 5.45. The register file has two read ports (*A1*/*RD1* and *A2*/*RD2*) and one write port (*A3*/*WD3*). The 5-bit addresses—*A1*, *A2*, and *A3*—can each access all $2^5 = 32$ registers. So, two registers can be read and one register written simultaneously.

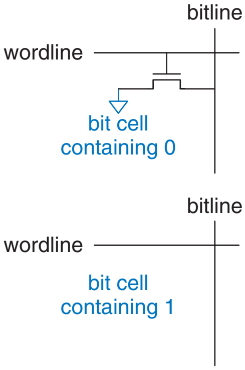


Figure 5.50 **ROM bit cells containing 0 and 1**

5.5.6 Read Only Memory (ROM)

ROM stores a bit as the presence or absence of a transistor. Figure 5.50 shows a simple ROM bit cell. To read the cell, the bitline is weakly pulled HIGH. Then, the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH.

Note that the ROM bit cell is a combinational circuit and has no state to "forget" if power is turned off.

The contents of a ROM can be indicated using *dot notation*. Figure 5.51 shows the dot notation for a 4-word × 3-bit ROM containing the data from Figure 5.41. A dot at the intersection of a row (wordline) and a column (bitline) indicates that the data bit is 1. For example, the top wordline has a single dot on $Data_1$, so the data word stored at *Address* 11 is 010.

Conceptually, ROMs can be built using two-level logic with a group of AND gates followed by a group of OR gates. The AND gates produce all possible minterms and, hence, form a decoder. Figure 5.52 shows the ROM of Figure 5.51 built using a decoder and OR gates. Each dotted row in Figure 5.51 is an input to an OR gate in Figure 5.52. For

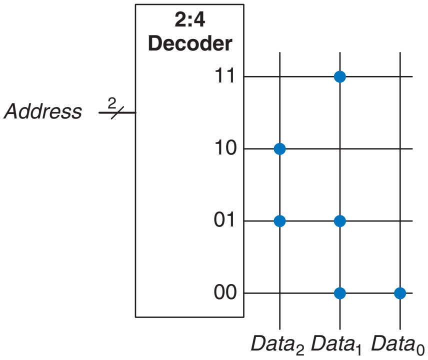


Figure 5.51 **4 × 3 ROM: dot notation**

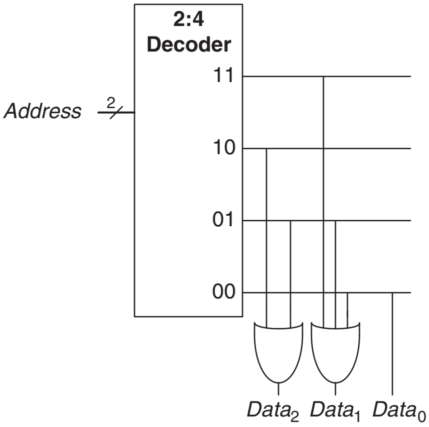


Figure 5.52 **4 × 3 ROM implementation using gates**

data bits with a single dot—in this case, $Data_0$—no OR gate is needed. This representation of a ROM is interesting because it shows how the ROM can perform any two-level logic function. In practice, ROMs are built from transistors instead of logic gates to reduce their size and cost. Section 5.6.3 explores the transistor-level implementation further.

The contents of the ROM bit cell in Figure 5.50 are specified during manufacturing by the presence or absence of a transistor in each bit cell. A *programmable ROM* (*PROM*, pronounced like the dance) places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.

Figure 5.53 shows the bit cell for a *fuse-programmable ROM*. The user programs the ROM by applying a high voltage to selectively blow fuses. If the fuse is present, the transistor is connected to GND and the cell holds a 0. If the fuse is destroyed, the transistor is disconnected from ground and the cell holds a 1. This is also called a *one-time programmable ROM*, because the fuse cannot be repaired once it is blown.

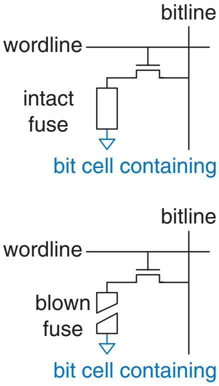


Figure 5.53 Fuse-programmable ROM bit cell

Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND. *Erasable PROMs* (*EPROMs*, pronounced "e-proms") replace the nMOS transistor and fuse with a *floating-gate transistor*. The floating gate is not physically attached to any other wires. When suitable high voltages are applied, electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline (decoder output). When the EPROM is exposed to intense ultraviolet (UV) light for about half an hour, the electrons are knocked off the floating gate, turning the transistor off. These actions are called *programming* and *erasing*, respectively. *Electrically erasable PROMs* (*EEPROMs*, pronounced "e-e-proms" or "double-e proms") and *Flash* memory use similar principles but include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM bit cells are individually erasable; Flash memory erases larger blocks of bits and is cheaper because fewer erasing circuits are needed. In 2021, Flash memory cost about $0.10 per GB, and the price continues to drop by 30% to 40% per year. Flash has become an extremely popular way to store large amounts of data in portable battery-powered systems such as cameras and music players.

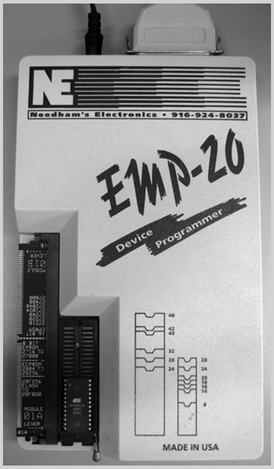
In summary, modern ROMs are not really read only; they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write but are nonvolatile.

Fujio Masuoka, 1944–
Received a Ph.D. in electrical engineering from Tohoku University, Japan. Developed memories and high-speed circuits at Toshiba from 1971 to 1994. Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's. Flash received its name because the process of erasing the memory reminds one of the flash of a camera. Toshiba was slow to commercialize the idea; Intel was first to market in 1988. Flash has grown into a $170 billion per year market. Dr. Masuoka later joined the faculty at Tohoku University and is working to develop a 3-dimensional transistor.

5.5.7 Logic Using Memory Arrays

Although they are used primarily for data storage, memory arrays can also perform combinational logic functions. For example, the $Data_2$ output of the ROM in Figure 5.51 is the XOR of the two *Address* inputs. Likewise,

Flash memory drives with Universal Serial Bus (USB) connectors have replaced floppy disks and CDs for sharing files because Flash costs have dropped so dramatically.



Programmable ROMs can be configured with a device programmer like the one shown below. The device programmer is attached to a computer, which specifies the type of ROM and the data values to program. The device programmer blows fuses or injects charge onto a floating gate on the ROM. Thus, the programming process is sometimes called *burning* a ROM.

$Data_0$ is the NOR of the two inputs. A 2^N-word × M-bit memory can perform any combinational function of N inputs and M outputs. For example, the ROM in Figure 5.51 performs three functions of two inputs.

Memory arrays used to perform logic are called *lookup tables* (*LUTs*). Figure 5.54 shows a 4-word × 1-bit memory array used as a lookup table to perform the function $Y = AB$. Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.

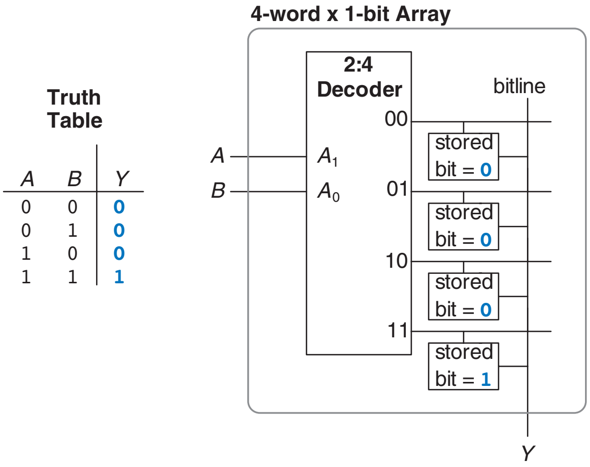
5.5.8 Memory HDL

HDL Example 5.6 describes a 2^N-word × M-bit RAM, and Figure 5.55 shows the resulting hardware. The RAM has a synchronous enabled write. In other words, writes occur on the rising edge of the clock if the write enable *we* is asserted. Reads occur immediately. When power is first applied, the contents of the RAM are unpredictable.

HDL Example 5.7 describes a 4-word × 3-bit ROM. The contents of the ROM are specified in the HDL case statement. A ROM as small as this one may be synthesized into logic gates rather than an array. Note that the seven-segment decoder from HDL Example 4.24 synthesizes into a ROM in Figure 4.20. HDL Example 5.8 shows a 3-ported 32 × 32 register file with entry 0 hardwired to 0.

5.6 LOGIC ARRAYS

Like memory, gates can be organized into regular arrays. If the connections are made programmable, these *logic arrays* can be configured to


Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5.54 4-word × 1-bit memory array used as a lookup table

HDL Example 5.6 RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
       we:   in  STD_LOGIC;
       adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
       din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
       dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we then mem(TO_INTEGER(adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem(TO_INTEGER(adr));
end;
```

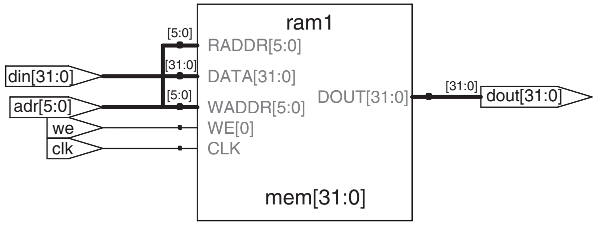


Figure 5.55 Synthesized ram

perform any function without the user having to connect wires in specific ways. The regular structure simplifies design. Logic arrays are mass-produced in large quantities, so they are inexpensive. Software tools allow users to map logic designs onto these arrays. Most logic arrays are also reconfigurable, allowing designs to be modified without replacing the hardware. Reconfigurability is valuable during development and is also useful in the field because a system can be upgraded by simply downloading the new configuration.

This section introduces two types of logic arrays: programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs).

HDL Example 5.7 ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout);

  always_comb
    case(adr)
      2’b00: dout = 3’b011;
      2’b01: dout = 3’b110;
      2’b10: dout = 3’b100;
      2’b11: dout = 3’b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(all) begin
    case adr is
      when “00”   => dout <= “011”;
      when “01”   => dout <= “110”;
      when “10”   => dout <= “100”;
      when “11”   => dout <= “010”;
      when others => dout <= “---”;
    end case;
  end process;
end;
```

HDL Example 5.8 REGISTER FILE

SystemVerilog

```
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [5:0]  a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];
  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity regfile is
  port(clk:        in  STD_LOGIC;
       we3:        in  STD_LOGIC;
       a1, a2, a3: in  STD_LOGIC_VECTOR(5  downto 0);
       wd3:        in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:   out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of STD_LOGIC_VECTOR
                                         (31 downto 0);
  signal mem: ramtype;
begin
  -- three ported register file
  -- read two ports combinationally (A1/RD1, A2/RD2)
  -- write third port on rising edge of clock (A3/WD3/WE3)
  -- register 0 hardwired to 0
  process(clk) begin
    if rising_edge(clk) then
      if we3 = ‘1’ then mem(to_integer(a3)) <= wd3;
      end if;
    end if;
  end process;
  process(a1, a2) begin
    if (to_integer(a1) = 0) then rd1 <= X"00000000";
    else rd1 <= mem(to_integer(a1));
    end if;
    if (to_integer(a2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(to_integer(a2));
    end if;
  end process;
end;
```

PLAs, the older technology, perform only combinational logic functions. FPGAs can perform both combinational and sequential logic.

5.6.1 Programmable Logic Array (PLA)

PLAs implement two-level combinational logic in sum-of-products (SOP) form. PLAs are built from an AND array followed by an OR array, as shown in Figure 5.56. The inputs (in true and complementary form) drive an AND array, which produces implicants. The implicants, in turn, are ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants, and P outputs.

Figure 5.57 shows the dot notation for a 3 × 3 × 2-bit PLA performing the functions $X = \overline{A}\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$. Each row in the AND array forms an implicant. Dots in each row of the AND array indicate which literals comprise the implicant. The AND array in Figure 5.57 forms three implicants: $\overline{A}\overline{B}C$, $AB\overline{C}$, and $A\overline{B}$. Dots in the OR array indicate which implicants are part of the output function.

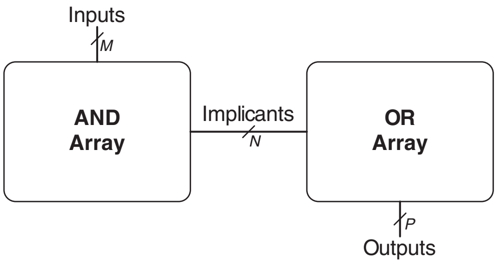


Figure 5.56 $M \times N \times P$**-bit PLA**

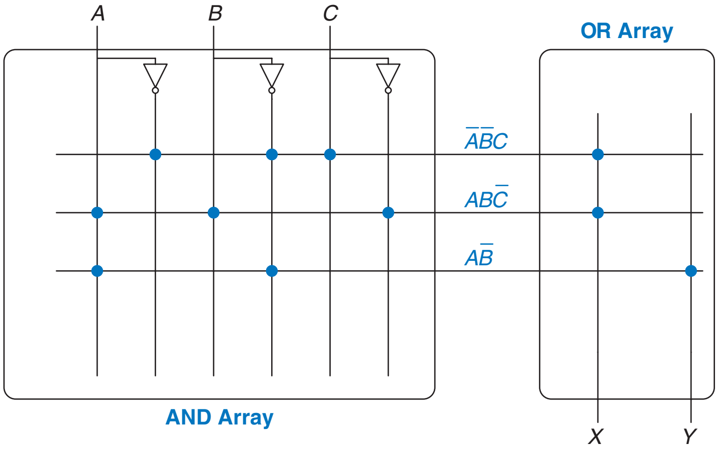


Figure 5.57 **3 × 3 × 2-bit PLA: dot notation**

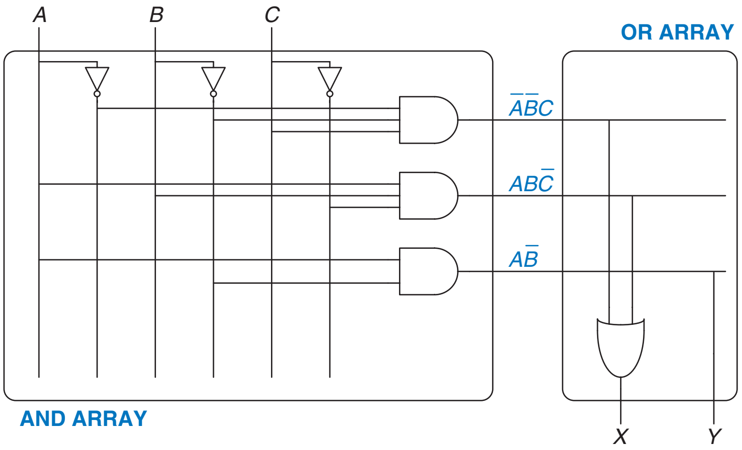


Figure 5.58 3 × 3 × 2-bit PLA using two-level logic

Figure 5.58 shows how PLAs can be built using two-level logic. An alternative implementation is given in Section 5.6.3.

ROMs can be viewed as a special case of PLAs. A $2M$-word × N-bit ROM is simply an $M \times 2^M \times N$-bit PLA. The decoder behaves as an AND plane that produces all 2^M minterms. The ROM array behaves as an OR plane that produces the outputs. If the function does not depend on all 2^M minterms, a PLA is likely to be smaller than a ROM. For example, an 8-word × 2-bit ROM is required to perform the same functions performed by the 3 × 3 × 2-bit PLA shown in Figures 5.57 and 5.58.

Simple programmable logic devices (*SPLDs*) are souped-up PLAs that add registers and various other features to the basic AND/OR planes. However, SPLDs and PLAs have largely been displaced by FPGAs, which are more flexible and efficient for building large systems.

5.6.2 Field Programmable Gate Array (FPGA)

An FPGA is an array of reconfigurable gates. Using software programming tools, a user can implement designs on the FPGA employing either an HDL or a schematic. FPGAs are more powerful and more flexible than PLAs for several reasons. They can implement both combinational and sequential logic. They can also implement multilevel logic functions, whereas PLAs can implement only two-level logic. Modern FPGAs integrate other useful features, such as built-in multipliers, high-speed I/Os, data converters including analog-to-digital converters, large RAM arrays, and processors.

FPGAs are built as an array of configurable *logic elements* (*LEs*), also referred to as *configurable logic blocks* (*CLBs*). Each LE can be configured to perform combinational or sequential functions. Figure 5.59

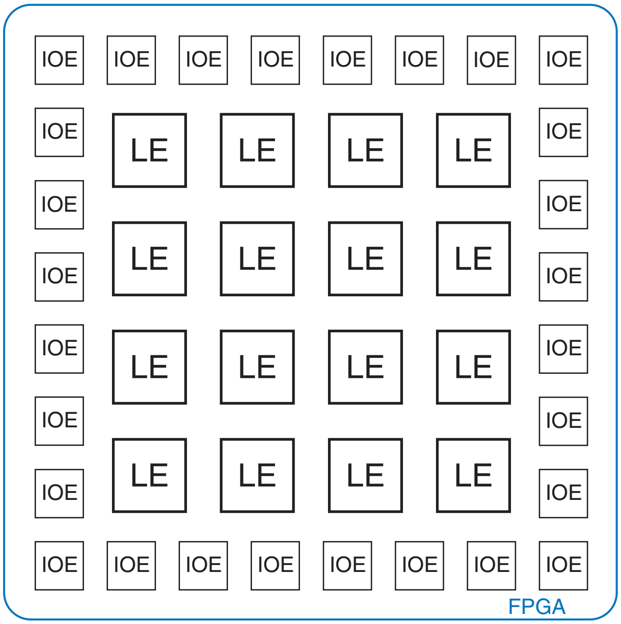


Figure 5.59 **General FPGA layout**

FPGAs are the brains of many consumer products, including automobiles, medical equipment, and media devices, such as MP3 players. The Mercedes Benz S-Class series, for example, has over a dozen Xilinx FPGAs or PLDs for uses ranging from entertainment to navigation to cruise control systems. FPGAs allow for quick time to market and make debugging or adding features late in the design process easier.

shows a general block diagram of an FPGA. The LEs are surrounded by *input/output elements* (*IOEs*) for interfacing with the outside world. The IOEs connect LE inputs and outputs to pins on the chip package. LEs can connect to other LEs and IOEs through programmable routing channels.

Two of the leading FPGA manufacturers are Intel (formerly, Altera Corp.) and Xilinx, Inc. Figure 5.60 shows a single LE from Intel's Cyclone IV FPGA, introduced in 2009. The key elements of the LE are a 4-input lookup table (LUT) and a 1-bit register. The LE also contains configurable multiplexers to route signals through the LE. The FPGA is configured by specifying the contents of the LUTs and the select signals for the multiplexers.

Altera Corporation was founded in 1983 as one of the original FPGA manufacturers, along with Xilinx Inc., which was founded in 1984. Altera was acquired by Intel in 2015 for $16.7 billion.

Each Cyclone IV LE has one 4-input LUT and one flip-flop. By loading the appropriate values into the LUT, it can be configured to perform any function of up to four variables. Configuring the FPGA also involves choosing the select signals that determine how the multiplexers route data through the LE and to neighboring LEs and IOEs. For example, depending on the multiplexer configuration, the LUT may receive one of its inputs from either *data 3* or the output of the LE's own register. The other three inputs always come from *data 1*, *data 2*, and *data 4*. The *data 1–4* inputs come from IOEs or the outputs of other LEs, depending on routing external to the LE. The LUT output either goes directly to

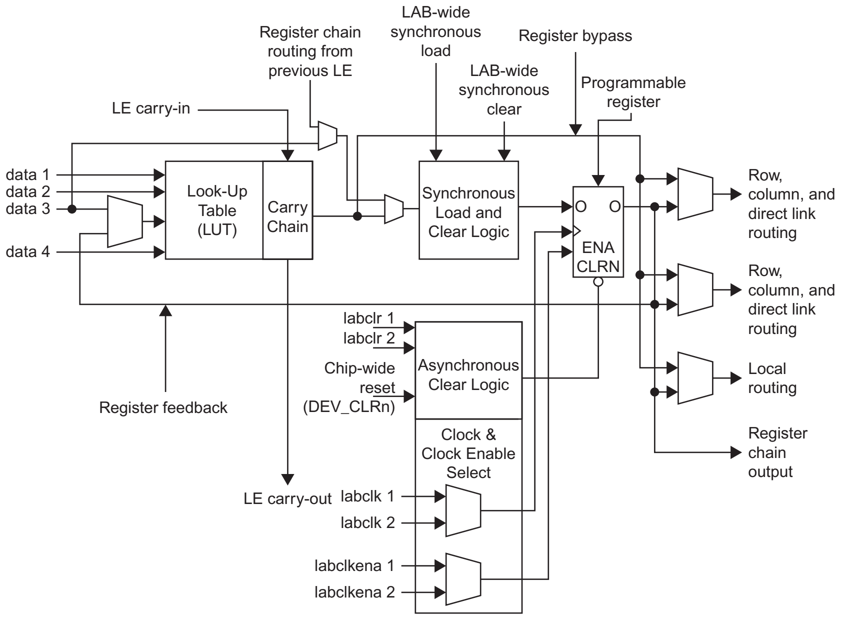


Figure 5.60 **Cyclone IV Logic Element (LE)**
(Reproduced with permission from the Altera Cyclone™ IV Handbook © 2010 Altera Corporation)

the LE output for combinational functions, or it can be fed through the flip-flop for registered functions. The flip-flop input comes from its own LUT output, the *data 3* input, or the register output of the previous LE. Additional hardware includes support for addition using the carry chain hardware, other multiplexers for routing, and flip-flop enable and reset. Altera groups 16 LEs together to create a *logic array block* (*LAB*) and provides local connections between LEs within the LAB.

In summary, each Cyclone IV LE can perform one combinational and/or registered function, which can involve up to four variables. Other brands of FPGAs are organized somewhat differently, but the same general principles apply. For example, Xilinx's 7-series FPGAs use 6-input LUTs instead of 4-input LUTs.

The designer configures an FPGA by first creating a schematic or HDL description of the design. The design is then synthesized onto the FPGA. The synthesis tool determines how the LUTs, multiplexers, and routing channels should be configured to perform the specified functions. This configuration information is then downloaded to the FPGA. Because Cyclone IV FPGAs store their configuration information in SRAM, they are easily reprogrammed. The FPGA may download its SRAM contents from a computer in the laboratory or from an EEPROM chip when the system is turned on. Some manufacturers include an EEPROM directly on the FPGA or use one-time programmable fuses to configure the FPGA.

Example 5.8 FUNCTIONS BUILT USING LEs

Explain how to configure one or more Cyclone IV LEs to perform the following functions: (a) $X = \bar{A}\bar{B}C + AB\bar{C}$ and $Y = A\bar{B}$ (b) $Y = JKLMPQR$; (c) a divide-by-3 counter with binary state encoding (see Figure 3.29(a)). You may show interconnection between LEs as needed.

Solution (a) Configure two LEs. One LUT computes X and the other LUT computes Y, as shown in Figure 5.61. For the first LE, inputs *data 1*, *data 2*, and *data 3* are A, B, and C, respectively (these connections are set by the routing channels). *data 4* is a don't care but must be tied to something, so it is tied to 0. For the second LE, inputs *data 1* and *data 2* are A and B, respectively; the other LUT inputs are don't cares and are tied to 0. Configure the final multiplexers to select the combinational outputs from the LUTs to produce X and Y. In general, a single LE can compute any function of up to four input variables in this fashion.

(b) Configure the LUT of the first LE to compute $X = JKLM$ and the LUT on the second LE to compute $Y = XPQR$. Configure the final multiplexers to select the combinational outputs X and Y from each LE. This configuration is shown in Figure 5.62. Routing channels between LEs, indicated by the dashed blue lines, connect the output of LE 1 to the input of LE 2. In general, a group of LEs can compute functions of N input variables in this manner.

(*A*)	(*B*)	(*C*)		(*X*)
data 1	*data 2*	*data 3*	*data 4*	LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(*A*)	(*B*)			(*Y*)
data 1	*data 2*	*data 3*	*data 4*	LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

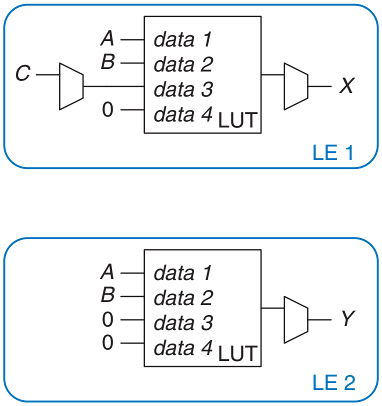


Figure 5.61 LE configuration for two functions of up to four inputs each

(c) The FSM has two bits of state ($S_{1:0}$) and one output (Y). The next state depends on the two bits of current state. Use two LEs to compute the next state from the current state, as shown in Figure 5.63. Use the two flip-flops, one from each LE, to hold this state. The flip-flops have a reset input that can be connected to an external *Reset* signal. The registered outputs are fed back to the LUT inputs using the multiplexer on *data 3* and routing channels between LEs, as indicated by the dashed blue lines. In general, another LE might be necessary to compute the output Y. However, in this case, $Y = S'_0$, so Y can come from LE 1. Hence, the entire FSM fits in two LEs. In general, an FSM requires at least one LE for each bit of state, and it may require more LEs for the output or next state logic if they are too complex to fit in a single LUT.

(*J*)	(*K*)	(*L*)	(*M*)	(*X*)
data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*P*)	(*Q*)	(*R*)	(*X*)	(*Y*)
data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

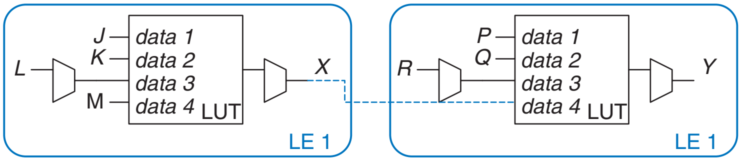


Figure 5.62 LE configuration for one function of more than four inputs

		(S_0)	(S_1)	(S_0')
data 1	data 2	data 3	data 4	LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

		(S_1)	(S_0)	(S_1')
data 1	data 2	data 3	data 4	LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

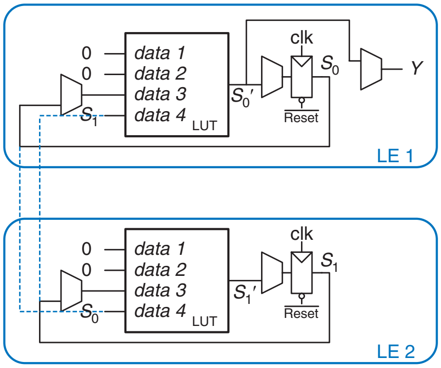


Figure 5.63 LE configuration for FSM with two bits of state

Example 5.9 MORE LOGIC ELEMENT EXAMPLES

How many Cyclone IV LEs are required to build each of the following circuits?

(a) 4-input AND

(b) 7-input XOR

(c) $Y = A(B + C + D + E) + \overline{A}(BCDE)$

(d) 12-bit shift register

(e) 32-bit 2:1 multiplexer

(f) 16-bit counter

(g) Arbitrary finite state machine with 2 bits of state, 2 inputs, and 3 outputs

Solution

(a) 1: The LUT can perform any function of up to 4 inputs.

(b) 2: The first LUT can compute a 4-input XOR. The second LUT can XOR that output with three more inputs.

(c) 3: One LUT computes $B + C + D + E$, a function of 4 inputs. A second LUT computes $BCDE$, a different function of 4 inputs. A third LUT uses 3 inputs (these two outputs and A) to compute Y.

(d) 12: A shift register contains one flip-flop per bit.

(e) 32: A 2:1 multiplexer is a function of three inputs: S, D_0, and D_1, so it requires one LUT per bit.

(f) 16: Each bit of a counter requires a flip-flop and a full adder. The LE contains the flip-flop and adder logic. Although a full adder has two outputs and might seem to need two LUTs, the LE contains special carry chain logic shown in Figure 5.60 optimized to perform addition with a single LE.

(g) 5: The FSM has two flip-flops, two next state signals, and three output signals. Each next state signal is a function of four variables (the two bits of state and two inputs), so it can be computed with one LUT. Thus, two LEs are sufficient for the next state logic and state register. Each output is a function of at most four signals, so one more LUT is needed for each output.

Example 5.10 LE DELAY

Alyssa P. Hacker is building a finite state machine that must run at 200 MHz. She uses a Cyclone IV FPGA with the following specifications: $t_{LE} = 381$ ps per LE, $t_{\text{setup}} = 76$ ps, and $t_{pcq} = 199$ ps for all flip-flops. The wiring delay between LEs is 246 ps. Assume that the hold time for the flip-flops is 0. What is the maximum number of levels of LEs that her design can use between two registers?

Solution Alyssa uses Equation 3.13 to solve for the maximum propagation delay of the logic: $\mathrm{t_{pd}} \leq T_c - (t_{pcq} + t_{\mathrm{setup}})$.

Thus, $t_{pd} = 5\,\mathrm{ns} - (0.199\,\mathrm{ns} + 0.076\,\mathrm{ns})$, so $t_{pd} \leq 4.725\,\mathrm{ns}$. The delay of each LE plus wiring delay between LEs, $t_{LE+wire}$, is $381\,\mathrm{ps} + 246\,\mathrm{ps} = 627\,\mathrm{ps}$. The maximum number of LEs, N, is $Nt_{LE+wire} \leq 4.725\,\mathrm{ns}$. Thus, $N = 7$.

5.6.3 Array Implementations*

Many ROMs and PLAs use *dynamic circuits* in place of pseudo-nMOS circuits. Dynamic gates turn the pMOS transistor ON for only part of the time, saving power when the pMOS is OFF and the result is not needed. Aside from this, dynamic and pseudo-nMOS memory arrays are similar in design and behavior.

To minimize their size and cost, ROMs and PLAs commonly use pseudo-nMOS (see Section 1.7.8) or dynamic circuits instead of conventional logic gates.

Figure 5.64(a) shows the dot notation for a 4 × 3-bit ROM that performs the following functions: $X = A \oplus B, Y = \overline{A} + B$, and $Z = \overline{A}\overline{B}$. These are the same functions as those of Figure 5.51, with the address inputs renamed A and B and the data outputs renamed X, Y, and Z. The pseudo-nMOS implementation is given in Figure 5.64(b). Each decoder output is connected to the gates of the nMOS transistors in its row. Remember that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pull-down (nMOS) network.

Pull-down transistors are placed at every junction without a dot. The dots from the dot notation diagram of Figure 5.64(a) are left visible in Figure 5.64(b) for easy comparison. The weak pull-up transistors pull the output HIGH for each wordline without a pull-down transistor. For example, when $AB = 11$, the 11 wordline is HIGH and transistors on X and Z

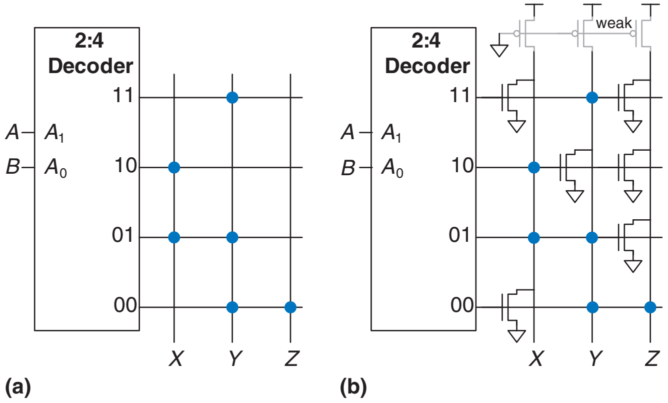


Figure 5.64 ROM implementation: (a) dot notation, (b) pseudo-nMOS circuit

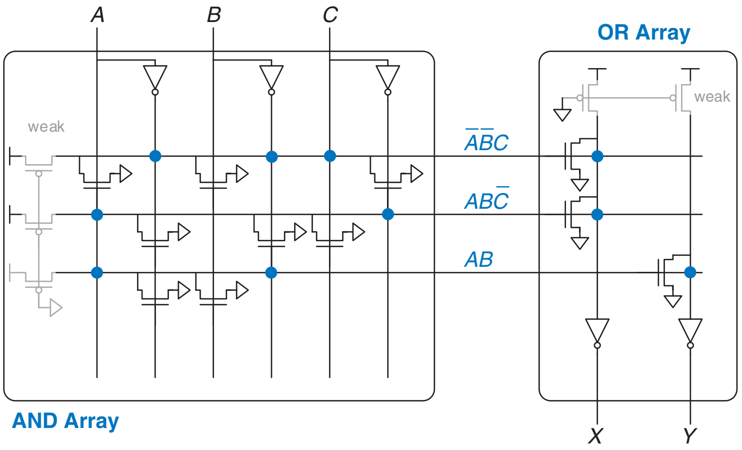


Figure 5.65 3 × 3 × 2-bit PLA using pseudo-nMOS circuits

turn on and pull those outputs LOW. The *Y* output has no transistor connecting to the 11 wordline, so *Y* is pulled HIGH by the weak pull-up.

PLAs can also be built using pseudo-nMOS circuits, as shown in Figure 5.65 for the PLA from Figure 5.57. Pull-down (nMOS) transistors are placed on the *complement* of dotted literals in the AND array and on dotted rows in the OR array. The columns in the OR array are sent through an inverter before they are fed to the output bits. Again, the blue dots from the dot notation diagram of Figure 5.57 are left visible in Figure 5.65 for easy comparison.

5.7 SUMMARY

This chapter introduced digital building blocks used in many digital systems. These blocks include arithmetic circuits, such as adders, subtractors, comparators, shifters, multipliers, and dividers; sequential circuits, such as counters and shift registers; and arrays for memory and logic. The chapter also explored fixed-point and floating-point representations of fractional numbers. In Chapter 7, we use these building blocks to build a microprocessor.

Adders form the basis of most arithmetic circuits. A half adder adds two 1-bit inputs, *A* and *B*, and produces a sum and a carry out. A full adder extends the half adder to also accept a carry in. *N* full adders can be cascaded to form a carry propagate adder (CPA) that adds two *N*-bit numbers. This type of CPA is called a ripple-carry adder because the carry ripples through each of the full adders. Faster CPAs can be constructed using lookahead or prefix techniques.

A subtractor negates the second input and adds it to the first. A magnitude comparator subtracts one number from another and determines the relative value based on the sign or carry out of the result.

A multiplier forms partial products using AND gates, then sums these bits using full adders. A divider repeatedly subtracts the divisor from the partial remainder and checks the sign of the difference to determine the quotient bits. A counter uses an adder and a register to increment a running count.

Fractional numbers are represented using fixed-point or floating-point forms. Fixed-point numbers are analogous to decimals, and floating-point numbers are analogous to scientific notation. Fixed-point numbers use ordinary arithmetic circuits, whereas floating-point numbers require more elaborate hardware to extract and process the sign, exponent, and mantissa.

Large memories are organized into arrays of words. The memories have one or more ports to read and/or write the words. Volatile memories, such as SRAM and DRAM, lose their state when the power is turned off. SRAM is faster than DRAM but requires more transistors. A register file is a small multiported SRAM array. Nonvolatile memories, called ROMs, retain their state indefinitely. Despite their names, most modern ROMs can be written.

Arrays are also a regular way to build logic. Memory arrays can be used as LUTs to perform combinational functions. PLAs are composed of dedicated connections between configurable AND and OR arrays; they implement only combinational logic. FPGAs are composed of many small LUTs and registers; they implement combinational and sequential logic. The LUT contents and their interconnections can be configured to perform any logic function. Modern FPGAs are easy to reprogram and are large and cheap enough to build highly sophisticated digital systems, so they are widely used in low- and medium-volume commercial products, as well as in education.

Exercises

Exercise 5.1 What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

(b) a carry-lookahead adder with 4-bit blocks

(c) a prefix adder

Exercise 5.2 Design two adders: a 64-bit ripple-carry adder and a 64-bit carry-lookahead adder with 4-bit blocks. Use only two-input gates. Each two-input gate is 15 μm^2, has a 50 ps delay, and has 20 fF of total gate capacitance. You may assume that the static power is negligible.

(a) Compare the area, delay, and power of the adders (operating at 100 MHz and 1.2 V).

(b) Discuss the trade-offs between power, area, and delay.

Exercise 5.3 Explain why a designer might choose to use a ripple-carry adder instead of a carry-lookahead adder.

Exercise 5.4 Design the 16-bit prefix adder of Figure 5.7 in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.5 The prefix network shown in Figure 5.7 uses black cells to compute all of the prefixes. Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k-1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

Exercise 5.6 The prefix network shown in Figure 5.7 is not the only way to calculate all of the prefixes in logarithmic time. The *Kogge-Stone* network is another common prefix network that performs the same function using a different connection of black cells. Research Kogge-Stone adders and draw a schematic similar to Figure 5.7 showing the connection of black cells in a Kogge-Stone adder.

Exercise 5.7 Recall that an N-input priority encoder has $\log_2 N$ outputs that encode which of the N inputs gets priority (see Exercise 2.36).

(a) Design an N-input priority encoder that has delay that increases logarithmically with N. Sketch your design and give the delay of the circuit in terms of the delay of its circuit elements.

(b) Code your design in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.8 Design the following comparators for 32-bit unsigned numbers. Sketch the schematics.

(a) not equal

(b) greater than or equal to

(c) less than

Exercise 5.9 Consider the signed comparator of Figure 5.12.

(a) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator correctly computes $A < B$.

(b) Give an example of two 4-bit signed numbers A and B for which a 4-bit signed comparator incorrectly computes $A < B$.

(c) In general, when does the N-bit signed comparator operate incorrectly?

Exercise 5.10 Modify the N-bit signed comparator of Figure 5.12 to correctly compute $A < B$ for all N-bit signed inputs A and B.

Exercise 5.11 Design the 32-bit ALU shown in Figure 5.15 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.12 Design the 32-bit ALU shown in Figure 5.17 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.13 Design the 32-bit ALU shown in Figure 5.18(a) using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.14 Design the 32-bit ALU shown in Figure 5.18(b) using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.15 Write a testbench to test the 32-bit ALU from Exercise 5.11. Then, use it to test the ALU. Include any test vector files necessary. Be sure to test enough corner cases to convince a reasonable skeptic that the ALU functions correctly.

Exercise 5.16 Repeat Exercise 5.15 for the ALU from Exercise 5.12.

Exercise 5.17 Repeat Exercise 5.15 for the ALU from Exercise 5.13.

Exercise 5.18 Repeat Exercise 5.15 for the ALU from Exercise 5.14.

Exercise 5.19 Build an *unsigned comparison unit* that compares two unsigned numbers A and B. The unit's input is the *Flags* signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are HS, LS, HI, and LO, which indicate that A is higher than or the same as (HS), lower than or the same as (LS), higher (HI), or lower (LO) than B.

(a) Write minimal equations for HS, LS, HI, and LO in terms of N, Z, C, and V.

(b) Sketch circuits for HS, LS, HI, and LO.

Exercise 5.20 Build a *signed comparison unit* that compares two signed numbers A and B. The unit's input is the *Flags* signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are GE, LE, GT, and LT, which indicate that A is greater than or equal to (GE), less than or equal to (LE), greater than (GT), or less than (LT) B.

(a) Write minimal equations for GE, LE, GT, and LT in terms of N, Z, C, and V.

(b) Sketch circuits for GE, LE, GT, and LT.

Exercise 5.21 Design a shifter that always shifts a 32-bit input left by 2 bits. The input and output are both 32 bits. Explain the design in words and sketch a schematic. Implement your design in your favorite HDL.

Exercise 5.22 Design 4-bit left and right rotators. Sketch a schematic of each design. Implement your designs in your favorite HDL.

Exercise 5.23 Design an 8-bit left shifter using only 24 2:1 multiplexers. The shifter accepts an 8-bit input A and a 3-bit shift amount, $shamt_{2:0}$. It produces an 8-bit output Y. Sketch the schematic.

Exercise 5.24 Explain how to build any N-bit shifter or rotator using only $N\log_2 N$ 2:1 multiplexers.

Exercise 5.25 The *funnel shifter* in Figure 5.66 can perform any N-bit shift or rotate operation. It shifts a $2N$-bit input right by k bits. The output Y is the N least significant bits of the result. The most significant N bits of the input are called B and the least significant N bits are called C. By choosing appropriate values of B, C, and k, the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of A, *shamt*, and N for

(a) logical right shift of A by *shamt*

(b) arithmetic right shift of A by *shamt*

(c) left shift of A by *shamt*

(d) right rotate of A by *shamt*

(e) left rotate of A by *shamt*

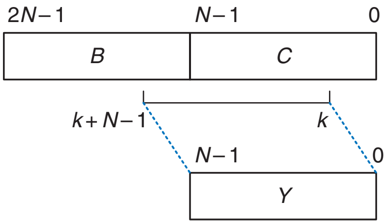


Figure 5.66 Funnel shifter

Exercise 5.26 Find the critical path for the unsigned 4 × 4 multiplier from Figure 5.21 in terms of an AND gate delay (t_{AND}) and a full adder delay (t_{FA}). What is the delay of an $N \times N$ multiplier built in the same way?

Exercise 5.27 Find the critical path for the unsigned 4 × 4 divider from Figure 5.22 in terms of a 2:1 mux delay (t_{MUX}), an adder delay (t_{FA}), and an inverter delay (t_{INV}). What is the delay of an $N \times N$ divider built in the same way?

Exercise 5.28 Design a multiplier that handles two's complement numbers.

Exercise 5.29 A *sign extension unit* extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output (see Section 1.4.6). It receives an M-bit input A and produces an N-bit output Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.30 A *zero extension unit* extends an unsigned number from M to N bits ($N > M$) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.31 Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

Exercise 5.32 What is the range of numbers that can be represented by the following number systems?

(a) U12.12 format (24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits)

(b) 24-bit sign/magnitude fixed-point numbers with 12 integer bits and 12 fraction bits

(c) Q12.12 format (24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits)

Exercise 5.33 Express the following base 10 numbers in 16-bit fixed-point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

(a) −13.5625

(b) 42.3125

(c) −17.15625

Exercise 5.34 Express the following base 10 numbers in 12-bit fixed-point sign/magnitude format with six integer bits and six fraction bits. Express your answer in hexadecimal.

(a) −30.5

(b) 16.25

(c) −8.078125

Exercise 5.35 Express the base 10 numbers in Exercise 5.33 in Q8.8 format (16-bit fixed-point two's complement format with eight integer bits and eight fraction bits). Express your answer in hexadecimal.

Exercise 5.36 Express the base 10 numbers in Exercise 5.34 in Q6.6 format (12-bit fixed-point two's complement format with six integer bits and six fraction bits). Express your answer in hexadecimal.

Exercise 5.37 Express the base 10 numbers in Exercise 5.33 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.38 Express the base 10 numbers in Exercise 5.34 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.39 Convert the following Q4.4 (two's complement binary fixed-point numbers) to base 10. The implied binary point is explicitly shown to aid in your interpretation.

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

Exercise 5.40 Repeat Exercise 5.39 for the following Q6.5 format (two's complement binary fixed-point) numbers.

(a) 011101.10101

(b) 100110.11010

(c) 101000.00100

Exercise 5.41 When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

Exercise 5.42 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0123456 + 81C564B7

(b) D0B10301 + D1B43203

(c) 5EF10324 + 5E039020

Exercise 5.43 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0D20004 + 72407020

(b) C0D20004 + 40DC0004

(c) (5FBE4000 + 3FF80000) + DFDE4000 (Why is the result counterintuitive? Explain.)

Exercise 5.44 Expand the steps in Section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

Exercise 5.45 Consider IEEE 754 single-precision floating-point numbers.

(a) How many numbers can be represented by the IEEE 754 single-precision floating-point format? You need not count ±∞ or NaN.

(b) How many additional numbers could be represented if ±∞ and NaN were not represented?

(c) Explain why ±∞ and NaN are given special representations.

Exercise 5.46 Consider the following decimal numbers: 245 and 0.0625.

(a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.

(b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?

(c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.

(d) Does integer comparison work with your new floating-point notation from part (c)?

(e) Why is it convenient for integer comparison to work with floating-point numbers?

Exercise 5.47 Design a single-precision floating-point adder using your favorite HDL. Before coding the design in an HDL, sketch a schematic of your design. Simulate and test your adder to prove to a skeptic that it functions correctly. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.4.

Exercise 5.48 In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.4.

(a) Write the steps necessary to perform 32-bit floating-point multiplication.

(b) Sketch the schematic of a 32-bit floating-point multiplier.

(c) Design a 32-bit floating-point multiplier in an HDL. Simulate and test your multiplier to prove to a skeptic that it functions correctly.

Exercise 5.49 In this problem, you will explore the design of a 32-bit prefix adder.

(a) Sketch a schematic of your design.

(b) Design the 32-bit prefix adder in an HDL. Simulate and test your adder to prove that it functions correctly.

(c) What is the delay of your 32-bit prefix adder from part (a)? Assume that each two-input gate delay is 100 ps.

(d) Design a pipelined version of the 32-bit prefix adder. Sketch the schematic of your design. How fast can your pipelined prefix adder run? You may assume a sequencing overhead ($t_{pcq} + t_{\text{setup}}$) of 80 ps. Make the design run as fast as possible.

(e) Design the pipelined 32-bit prefix adder in an HDL.

Exercise 5.50 An incrementer adds 1 to an *N*-bit number. Build an 8-bit incrementer using half adders.

Exercise 5.51 Build a 32-bit synchronous *Up/Down counter*. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when $Up = 1$, the circuit counts up, and when $Up = 0$, the circuit counts down.

Exercise 5.52 Design a 32-bit counter that adds 4 at each clock edge. The counter has reset and clock inputs. Upon reset, the counter output is all 0.

Exercise 5.53 Modify the counter from Exercise 5.52 such that the counter will either increment by 4 or load a new 32-bit value, *D*, on each clock edge, depending on a control signal *Load*. When $Load = 1$, the counter loads the new value *D*. Otherwise, it increments by 4.

Exercise 5.54 An *N*-bit *Johnson counter* consists of an *N*-bit shift register with a reset signal. The output of the shift register (S_{out}) is inverted and fed back to the input (S_{in}). When the counter is reset, all of the bits are cleared to 0.

(a) Show the sequence of outputs, $Q_{3:0}$, produced by a 4-bit Johnson counter starting immediately after the counter is reset.

(b) How many cycles elapse until an *N*-bit Johnson counter repeats its sequence? Explain.

(c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs $Y_{9:0}$. When the counter is reset, Y_0 is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.

(d) What advantages might a Johnson counter have over a conventional counter?

Exercise 5.55 Write the HDL for a 4-bit scannable flip-flop like the one shown in Figure 5.39. Simulate and test your HDL module to prove that it functions correctly.

Exercise 5.56 The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly, a 1 will have at least three 1's.

(a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.

(b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.

(c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

Exercise 5.57 Flash EEPROM, simply called Flash memory, is a fairly recent invention that has revolutionized consumer electronics. Research and explain how Flash memory works. Use a diagram illustrating the floating gate. Describe how a bit in the memory is programmed. Properly cite your sources.

Exercise 5.58 The extraterrestrial life project team has just discovered aliens living on the bottom of Mono Lake. They need to construct a circuit to classify the aliens by potential planet of origin based on measured features available from the NASA probe: greenness, brownness, sliminess, and ugliness. Careful consultation with xenobiologists leads to the following conclusions:

- If the alien is green and slimy or ugly, brown, and slimy, it might be from Mars.
- If the critter is ugly, brown, and slimy, or green and neither ugly nor slimy, it might be from Venus.
- If the beastie is brown and neither ugly nor slimy or is green and slimy, it might be from Jupiter.

Note that this is an inexact science; for example, a life form which is mottled green and brown and is slimy but not ugly might be from either Mars or Jupiter.

(a) Program a 4 × 4 × 3 PLA to identify the alien. You may use dot notation.

(b) Program a 16 × 3 ROM to identify the alien. You may use dot notation.

(c) Implement your design in an HDL.

Exercise 5.59 Implement the following functions using a single 16 × 3 ROM. Use dot notation to indicate the ROM contents.

(a) $X = AB + B\overline{C}D + \overline{A}\overline{B}$

(b) $Y = AB + BD$

(c) $Z = A + B + C + D$

Exercise 5.60 Implement the functions from Exercise 5.59 using a 4 × 8 × 3 PLA. You may use dot notation.

Exercise 5.61 Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

(a) a 16-bit adder/subtractor with C_{in} and C_{out}

(b) an 8 × 8 multiplier

(c) a 16-bit priority encoder (see Exercise 2.36)

Exercise 5.62 Consider the ROM circuits in Figure 5.67. For each row, can the circuit in column I be replaced by an equivalent circuit in column II by proper programming of the latter's ROM?

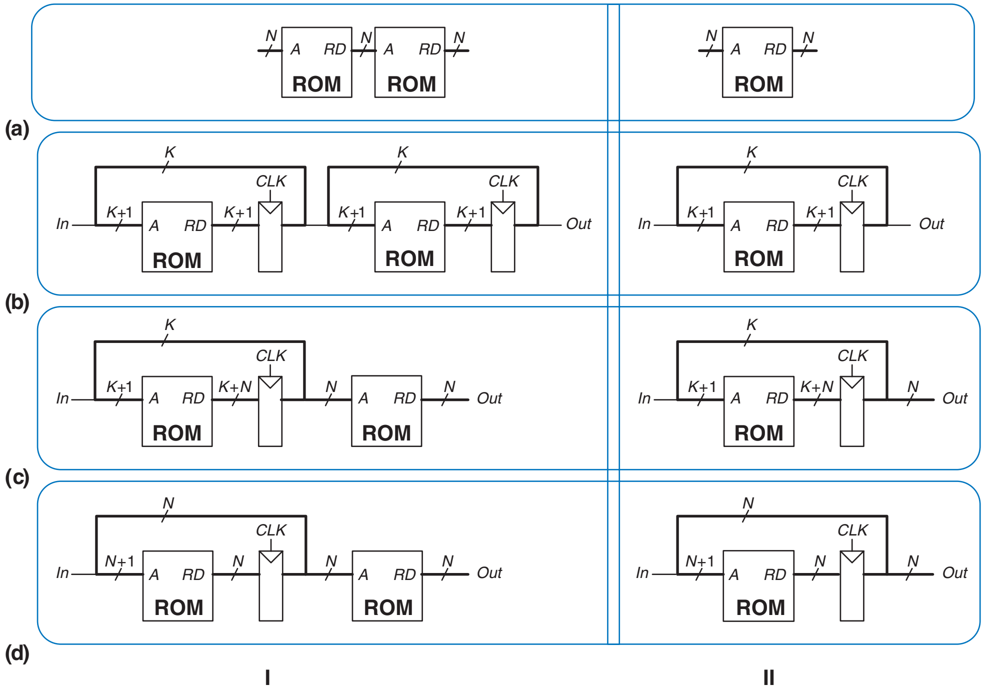


Figure 5.67 ROM circuits

Exercise 5.63 How many Cyclone IV FPGA LEs are required to perform each of the following functions? Show how to configure one or more LEs to perform the function. You should be able to do this by inspection, without performing logic synthesis.

(a) the combinational function from Exercise 2.13(c)

(b) the combinational function from Exercise 2.17(c)

(c) the two-output function from Exercise 2.24

(d) the function from Exercise 2.35

(e) a four-input priority encoder (see Exercise 2.36)

Exercise 5.64 Repeat Exercise 5.63 for the following functions:

(a) an eight-input priority encoder (see Exercise 2.36)

(b) a 3:8 decoder

(c) a 4-bit carry propagate adder (with no carry in or out)

(d) the FSM from Exercise 3.22

(e) the Gray code counter from Exercise 3.27

Exercise 5.65 Consider the Cyclone IV LE shown in Figure 5.60. According to the datasheet, it has the timing specifications given in Table 5.7.

(a) What is the minimum number of Cyclone IV LEs required to implement the FSM of Figure 3.26?

(b) Without clock skew, what is the fastest clock frequency at which this FSM will run reliably?

(c) With 3 ns of clock skew, what is the fastest frequency at which the FSM will run reliably?

Table 5.7 Cyclone IV timing

Name	Value (ps)
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Exercise 5.66 Repeat Exercise 5.65 for the FSM of Figure 3.31(b).

Exercise 5.67 You would like to use an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. The design is to be implemented as an FSM using a Cyclone IV FPGA. According to the data sheet, the FPGA has timing characteristics shown in Table 5.7. You would like your FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which the FSM will run?

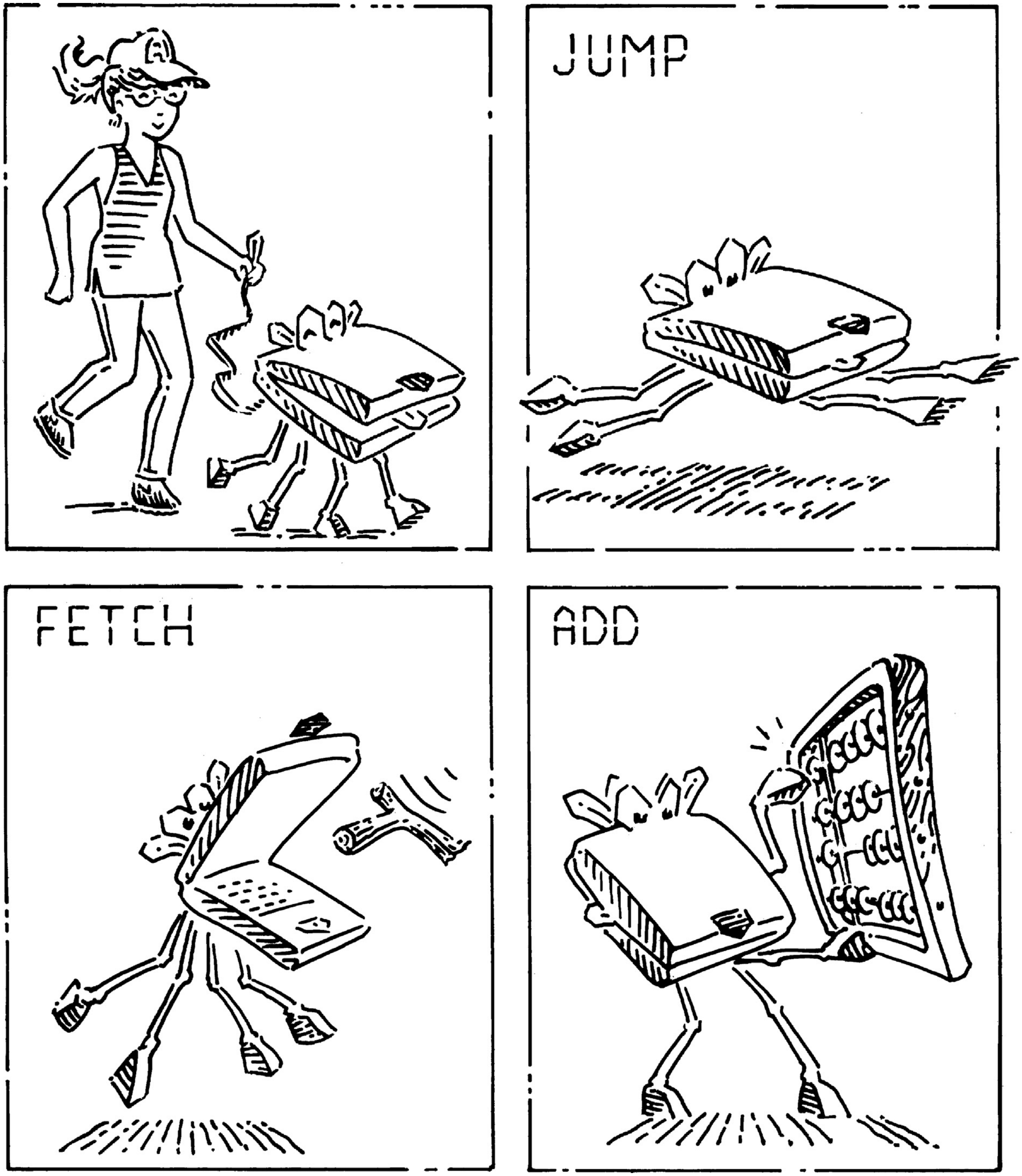
Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 5.1 What is the largest possible result of multiplying two unsigned N-bit numbers?

Question 5.2 *Binary coded decimal* (*BCD*) representation uses four bits to encode each decimal digit. For example, 42_{10} is represented as 01000010_{BCD}. Explain in words why processors might use BCD representation.

Question 5.3 Design hardware to add two 8-bit unsigned BCD numbers (see Question 5.2). Sketch a schematic for your design, and write an HDL module for the BCD adder. The inputs are A, B, and C_{in}, and the outputs are S and C_{out}. C_{in} and C_{out} are 1-bit carries and A, B, and S are 8-bit BCD numbers.


JUMP
FETCH
ADD

6

Architecture

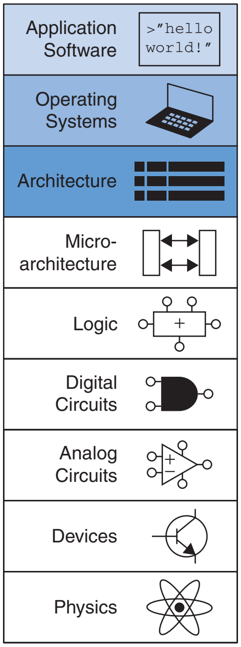
6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as RISC-V, ARM, x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and branch. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, registers, or the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called *machine language*. Just as we use letters to encode human language, computers use binary numbers to encode machine language. The RISC-V architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called *assembly language*.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions—such as add, subtract, and branch—that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.



Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00006-4

Krste Asanović started RISC-V as a summer project. He is a professor of computer science at the University of California, Berkeley and the Chairman of the Board for RISC-V International, formerly known as the RISC-V Foundation. He is also the cofounder of SiFive, a company that develops and commercializes RISC-V chips, boards, and supporting tools. (Photo printed with permission.)

Andrew Waterman designs microprocessors at SiFive, a company he cofounded with Krste Asanović in 2015 to provide low-cost RISC-V cores and custom chips. He earned his PhD in computer science from UC Berkeley in 2016, where, weary of the vagaries of existing instruction-set architectures, he co-designed the RISC-V ISA and the first RISC-V cores. (Photo printed with permission.)

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. These microprocessors can all run the same programs, but they use different underlying hardware. Therefore, they offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in gadgets or laptop computers. The specific arrangement of registers, memories, arithmetic/logical units (ALUs), and other building blocks to form a microprocessor is called the *microarchitecture* and will be the subject of Chapter 7.

In this text, we introduce the RISC-V (pronounced “risk five”) architecture, the first open-source instruction set architecture with broad commercial support. We describe the RISC-V 32-bit integer instruction set (RV32I) version 2.2, which forms the core of RISC-V’s instruction set. Sections 6.6 and 6.7 summarize features of other versions of the architecture. The *RISC-V Instruction Set Manual*, available online, is the authoritative definition of the architecture.

The RISC-V architecture was initially defined in 2010 at the University of California, Berkeley by Krste Asanović, Andrew Waterman, David Patterson, and others. Since its inception, many people have contributed to its development. RISC-V is unusual in that its open-source nature makes it free to use, and it is comparable in capabilities to commercial architectures such as ARM and x86. So far, only a few companies have built commercial chips, including SiFive and Western Digital, but adoption is rapidly increasing. We start our journey into understanding the RISC-V architecture by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

Throughout the chapter, we describe how the design of the RISC-V architecture was motivated by four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design*: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer’s native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the RISC-V instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java. (These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

David Patterson has been a professor of computer science at the University of California, Berkeley since 1976, and he coinvented *reduced instruction set computing* with John Hennessy in 1984. It was later commercialized as the SPARC architecture. He helped develop the RISC-V architecture and continues to play an integral role in its development. (Photo printed with permission.)

6.2.1 Instructions

One of the most common operations computers perform is addition. Code Example 6.1 shows code for adding variables `b` and `c` and writing the result to `a`. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in RISC-V assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	RISC-V Assembly Code
`a = b + c;`	`add a, b, c`

The first part of the assembly instruction, `add`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on `b` and `c`, the *source operands*, and the result is written to `a`, the *destination operand*.

Mnemonic (pronounced nuh-maa-nik) comes from the Greek word μιμνΕσκεστηαι, *to remember*. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

Code Example 6.2 shows that subtraction is similar to addition. The `sub` instruction format is the same as the `add` instruction: destination operand, followed by two sources. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Code Example 6.2 SUBTRACTION

High-Level Code	RISC-V Assembly Code
`a = b − c;`	`sub a, b, c`

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple RISC-V instructions, as shown in Code Example 6.3.

RISC-V is called "five" because it is the fifth RISC architecture developed at Berkeley.

In the Preface, we describe several simulators and tools for compiling and simulating C and RISC-V assembly code. We also provide labs (available on this textbook's companion site, see Preface) that show how to use these tools.

Code Example 6.3 MORE COMPLEX CODE

High-Level Code

```
a = b + c − d;  // single-line comment
                /* multiple-line
                   comment */
```

RISC-V Assembly Code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t − d
```

In the high-level language examples, single-line comments begin with // and continue until the end of the line. Multiline comments begin with /* and end with */. In RISC-V assembly language, only single-line comments are used. They begin with a hash symbol (#) and continue until the end of the line. The assembly language program in Code Example 6.3 stores the intermediate result (b + c) in a temporary variable t. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The RISC-V instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, RISC-V is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a "string move" instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture, such as RISC-V, minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation, whereas an instruction set with 256 instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

John Hennessy is a professor of electrical engineering and computer science at Stanford University and served as the president of Stanford from 2000 to 2016. He coinvented *reduced instruction set computing* with David Patterson. He also developed the MIPS computer architecture and cofounded MIPS Computer Systems in 1984. The MIPS processor was used in many commercial systems, including products from Silicon Graphics, Nintendo, and Cisco. John Hennessy and David Patterson were given the Turing Award in 2017 for pioneering a quantitative approach to the design and evaluation of computer architectures. (Photo printed with permission.)

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.2, the variables a, b, and c are all operands. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. RISC-V is called a 32-bit architecture because it operates on 32-bit data.

64- and 128-bit versions of the RISC-V architecture also exist, but we will focus on the 32-bit version in this book. The wider versions (RV64I and RV128I) are nearly identical to the 32-bit version (RV32I) except for the width of the registers and memory addresses. The main other additions are instructions that operate on only the lower half of a word and memory operations that transfer wider words.

Registers

Instructions need to access operands quickly so that they can run fast, but operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The RISC-V architecture has 32 registers, called the *register set*, stored in a small multiported memory called a *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small register file is faster than reading it from a large memory. A register file is typically built from a small SRAM array (see Section 5.5.3).

Code Example 6.4 shows the add instruction with register operands. The variables a, b, and c are arbitrarily placed in s0, s1, and s2. The name s1 is pronounced "register s1" or simply "s1." The instruction adds the 32-bit values contained in s1 (b) and s2 (c) and writes the 32-bit result to s0 (a). Code Example 6.5 shows RISC-V assembly code using a register, t0, to store the intermediate calculation of b + c.

Appendix B, which is located on the inside covers of the textbook, gives a handy summary of the entire RISC-V instruction set.

Code Example 6.4 REGISTER OPERANDS

High-Level Code

```
a = b + c;
```

RISC-V Assembly Code

```
# s0 = a, s1 = b, s2 = c
  add s0, s1, s2      # a = b + c
```

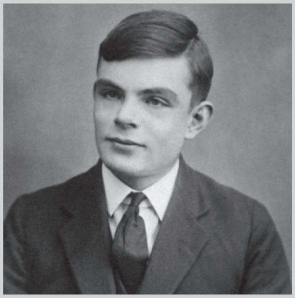
Code Example 6.5 TEMPORARY REGISTERS

High-Level Code

```
a = b + c − d;
```

RISC-V Assembly Code

```
# s0 = a, s1 = b, s2 = c, s3 = d, t0 = t
  add t0, s1, s2  # t = b + c
  sub s0, t0, s3  # a = t − d
```



Alan Turing, 1912–1954
A British mathematician and computer scientist who is considered the founder of theoretical computer science and artificial intelligence. He invented the Turing machine, a mathematical model of computation that represents an abstract processor. He also developed an electromechanical machine to decipher encrypted messages during World War II, which shortened the war and saved millions of lives. In 1952, Turing was prosecuted for homosexual acts and was sentenced to a chemical castration treatment in lieu of prison. Two years later, he died of cyanide poisoning. The Turing Award, which is the highest honor in computing, was named in his honor and has been awarded annually since 1966. It currently includes an accompanying cash prize of $1 million.

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into RISC-V assembly language. Assume variables a−c are held in registers s0−s2, and f−j are in s3−s7.

```
// high-level code
a = b − c;
f = (g + h) − (i + j);
```

Solution The program uses four assembly language instructions.

```
# RISC-V assembly code
# s0 = a, s1 = b, s2 = c, s3 = f, s4 = g, s5 = h, s6 = i, s7 = j
  sub s0, s1, s2       # a = b − c
  add t0, s4, s5       # t0 = g + h
  add t1, s6, s7       # t1 = i + j
  sub s3, t0, t1       # f = (g + h) − (i + j)
```

The Register Set

Table 6.1 lists the name and use for each of the 32 RISC-V registers. Registers are numbered 0 to 31 and are given a special name to indicate a register's conventional purpose. Assembly instructions typically use the special name—for example, s1—for clarity, but they may also use the register number (e.g., x9 for register number 9). The zero register always holds the constant 0; values written to it are discarded. Registers s0 to s11 (registers 8–9 and 18–27) and t0 to t6 (registers 5–7 and 28–31) are used for storing variables; ra and a0 to a7 have special uses during function calls, as discussed in Section 6.3.7. Registers 2 to 4 are also called sp, gp, and tp and will be described later.

Constants/Immediates

In addition to register operations, RISC-V instructions can use constant or *immediate* operands. These constants are called *immediates* because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the add immediate instruction, addi, that adds an immediate to a register. In assembly code, the immediate can be written in decimal, hexadecimal, or binary. Hexadecimal constants in RISC-V assembly language start with 0x and binary constants start with 0b, as they do in C. Immediates are 12-bit two's complement numbers, so they are sign-extended to 32 bits. The addi instruction is a useful way to initialize register values with small constants. Code Example 6.7 initializes the variables i, x, and y to 0, 2032, −78, respectively.

Immediates can be written in decimal, hexadecimal, or binary. For example, the following instructions all put the decimal value 109 into s5:

```
addi s5,x0,0b1101101
addi s5,x0,0x6D
addi s5,x0,109
```

Table 6.1 **RISC-V register set**

Name	Register Number	Use
`zero`	`x0`	Constant value 0
`ra`	`x1`	Return address
`sp`	`x2`	Stack pointer
`gp`	`x3`	Global pointer
`tp`	`x4`	Thread pointer
`t0–2`	`x5–7`	Temporary registers
`s0/fp`	`x8`	Saved register/Frame pointer
`s1`	`x9`	Saved register
`a0–1`	`x10–11`	Function arguments/Return values
`a2–7`	`x12–17`	Function arguments
`s2–11`	`x18–27`	Saved registers
`t3–6`	`x28–31`	Temporary registers

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code

```
a = a + 4;
b = a − 12;
```

RISC-V Assembly Code

```
# s0 = a, s1 = b
  addi s0, s0, 4      # a = a + 4
  addi s1, s0, −12    # b = a − 12
```

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code

```
i = 0;
x = 2032;
y = −78;
```

RISC-V Assembly Code

```
# s4 = i, s5 = x, s6 = y
  addi s4, zero, 0       # i = 0
  addi s5, zero, 2032    # x = 2032
  addi s6, zero, −78     # y = −78
```

To create larger constants, use a *load upper immediate* instruction (`lui`) followed by an add immediate instruction (`addi`), as shown in Code Example 6.8. The `lui` instruction loads a 20-bit immediate into the most significant 20 bits of the instruction and places zeros in the least significant bits.

The int data type in C represents a *signed* number, that is, a two's complement integer. The C specification requires that int be *at least* 16 bits wide but does not require a particular size. Most modern compilers (including those for RV32I) use 32 bits, so an int represents a number in the range $[-2^{31}, 2^{31}-1]$. C also defines int32_t as a 32-bit two's complement integer, but this is more cumbersome to type.

Code Example 6.8 32-BIT CONSTANT EXAMPLE

High-Level Code

```
int a = 0xABCDE123;
```

RISC-V Assembly Code

```
lui  s2, 0xABCDE     # s2 = 0xABCDE000
addi s2, s2, 0x123   # s2 = 0xABCDE123
```

When creating large immediates, if the 12-bit immediate in addi is negative (i.e., bit 11 is 1), the upper immediate in the lui must be incremented by one. Remember that addi *sign*-extends the 12-bit immediate, so a negative immediate will have all 1's in its upper 20 bits. Because all 1's is −1 in two's complement, adding all 1's to the upper immediate results in subtracting 1 from the upper immediate. Code Example 6.9 shows such a case where the desired immediate is 0xFEEDA987. lui s2, 0xFEEDB puts 0xFEEDB000 into s2. The desired 20-bit upper immediate, 0xFEEDA, is incremented by 1. 0x987 is the 12-bit representation of −1657, so addi s2, s2, −1657 adds s2 and the sign-extended 12-bit immediate (0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987) and places the result in s2, as desired.

Code Example 6.9 32-BIT CONSTANT WITH A ONE IN BIT 11

High-Level Code

```
int a = 0xFEEDA987;
```

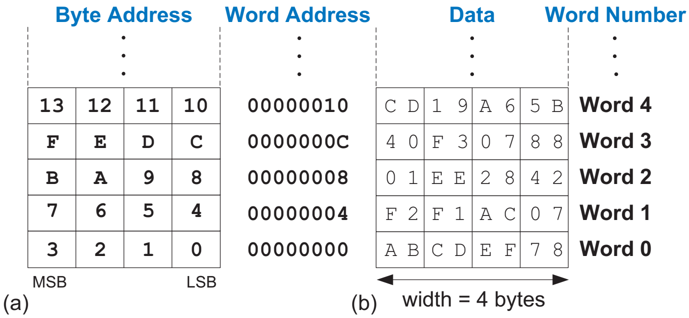
RISC-V Assembly Code

```
lui  s2, 0xFEEDB     # s2 = 0xFEEDB000
addi s2, s2, −1657   # s2 = 0xFEEDA987
```

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 32 variables. However, data can also be stored in memory. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the RISC-V architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from Section 5.5 that memories are organized as an array of data words. The RV32I RISC-V architecture uses 32-bit memory addresses and 32-bit data words.

RISC-V uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in Figure 6.1(a). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4.



Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	C D 1 9 A 6 5 B	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0

Figure 6.1 RISC-V byte-addressable memory showing: (a) byte address and (b) data

The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. The order of bytes within a word is discussed further in Section 6.6.1. Both the 32-bit word address and the data value in Figure 6.1(b) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

The *load word* instruction, `lw`, reads a data word from memory into a register. Code Example 6.10 loads memory word 2, located at address 8, into `a` (`s7`). In C, the number inside the brackets is the *index* or word number, which we discuss further in Section 6.3.6. The `lw` instruction specifies the memory address using an *offset* added to a *base register*. Recall that each data word is 4 bytes, so the word address is four times the word number. Word number 0 is at address 0, word 1 is at address 4, word 2 at address 8, and so on. In this example, the base register, zero, is added to the offset, 8, to get address 8, or word 2. After the load word instruction (`lw`) is executed in Code Example 6.10, `s7` holds the value 0x01EE2842, which is the data value stored at memory address 8 in Figure 6.1.

Code Example 6.10 READING MEMORY

High-Level Code	RISC-V Assembly Code

`a = mem[2];`	`# s7 = a`
	`  lw s7, 8(zero) # s7 = data at memory address (zero + 8)`

The *store word* instruction, `sw`, writes a data word from a register into memory. Code Example 6.11 writes the value 42 from register `t3` into memory word 5, located at address 20.

Many RISC-V implementations require *word-aligned addresses*—that is, a word address that is divisible by four—for `lw` and `sw`. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others require strict alignment for simplicity. In this textbook, we will assume strict alignment. Of course, byte addresses for load byte and store byte, `lb` and `sb` (discussed in Section 6.3.6), need not be word aligned.

Code Example 6.11 WRITING MEMORY

High-Level Code

```
mem[5] = 42;
```

RISC-V Assembly Code

```
addi t3, zero, 42   # t3 = 42
sw   t3, 20(zero)   # data value at memory address 20 = 42
```

Katherine Johnson, 1918–2020
Creola Katherine Johnson was a mathematician and computer scientist and one of the first African American women to work at NASA. At 18 years old, she graduated summa cum laude with a bachelor's degrees in mathematics and French from West Virginia University. When she joined NASA, she worked as a "computer," a group of people, mostly women, who performed precise calculations. In 1961, Johnson calculated the trajectory of Alan Shepard, the first American in space. Early in NASA's history, women were discouraged from having their names on reports, even when they did most of the work. Her NASA colleagues trusted her calculations, so Johnson helped facilitate the adoption of electronic computers by verifying their calculations. President Barack Obama awarded her the Presidential Medal of Freedom in 2015.

6.3 PROGRAMMING

Software languages such as C or Java are called *high-level programming languages* because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs, such as arithmetic and logical operations, if/else statements, for and while loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we begin by discussing program flow and instructions that support these high-level constructs. Then, we explore how to translate the high-level constructs into RISC-V assembly code.

6.3.1 Program Flow

Like data, instructions are stored in memory. Each instruction is 32 bits (4 bytes) long, as we will discuss in Section 6.4, so consecutive instruction addresses increase by four. For example, in the code snippet below, the `addi` instruction is located in memory at address 0x538 and the next instruction, `lw`, is at address 0x53C.

Memory address	Instruction
`0x538`	`addi s1, s2, s3`
`0x53C`	`lw   t2, 8(s1)`
`0x540`	`sw   s3, 3(t6)`

The *program counter*—also called the PC—keeps track of the current instruction. The PC holds the memory address of the current instruction and increments by four after each instruction completes so that the processor can read or *fetch* the next instruction from memory. For example, when `addi` is executing, PC is 0x538. After `addi` completes, PC increments by four to 0x53C and the processor fetches the `lw` instruction at that address.

6.3.2 Logical, Shift, and Multiply Instructions

The RISC-V architecture defines a variety of logical and arithmetic instructions. We introduce these instructions briefly here because they are necessary to implement higher-level constructs.

Logical Instructions

RISC-V *logical operations* include and, or, and xor. These each operate bitwise on two source registers and write the result to a destination register, as shown in Figure 6.2. Immediate versions of these logical operations, andi, ori, and xori, use one source register and a 12-bit sign-extended immediate.[1]

Source registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
and s3, s1, s2	s3	0100 0110	1010 0001	0000 0000	0000 0000
or s4, s1, s2	s4	1111 1111	1111 1111	1111 0001	1011 0111
xor s5, s1, s2	s5	1011 1001	0101 1110	1111 0001	1011 0111

Figure 6.2 Logical operations

The and instruction is useful for *clearing* or *masking* bits (i.e., forcing bits to 0). For example, the and instruction in Figure 6.2 clears the bits in s1 that are low in s2. In this case, the bottom two bytes of s1 are cleared. The unmasked top two bytes of s1, 0x46A1, are placed in s3. Any subset of register bits can be cleared. For example, to clear bit 3 of s0 and place the result in s6, use andi s6,s0,0xFF7.

The or instruction is useful for combining bitfields from two registers. For example, 0x347A0000 OR 0x000072FC = 0x347A72FC. It can also be used to *set* bits in a register (i.e., force a bit to 1). For example, to set bit 5 of s0 and place the result in s7, use ori s7,s0,0x020.

A logical NOT operation can be performed with xori s8,s1,−1. Remember that −1 (0xFFF) is sign-extended to 0xFFFFFFFF (all 1's). XOR with all 1's inverts all the bits, so s8 will get the one's complement of s1.

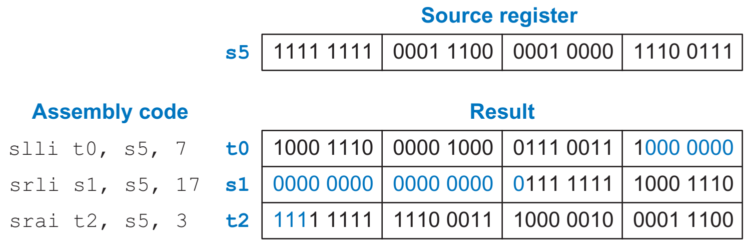
Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. RISC-V shift operations are sll (shift left logical), srl (shift right logical), and sra (shift right arithmetic). As discussed in Section 5.2.5, left shifts always fill the least significant bits with zeros. However, right shifts can be either *logical* (zeros shift into the most significant bits) or *arithmetic* (the sign bit shifts into the most significant bits). These shifts specify the shift amount in the second source register. Immediate versions of each instruction are also available (slli, srli, and srai), where the amount to shift is specified by a 5-bit unsigned immediate.

The RISC-V base instruction set does not currently include bit manipulations beyond shifts. Some instruction set architectures also include rotate instructions, as well as other bit manipulation instructions such as bit clear, bit set, etc. As of 2021, the "B" Standard Extension of RISC-V for Bit Manipulations is planned but not completed.

[1] Sign-extended logical immediates are somewhat unusual. Many other architectures, such as MIPS and ARM, zero-extend the immediate for logical operations.

Figure 6.3 shows the assembly code and resulting register values for `slli`, `srli`, and `srai` when shifting by an immediate value. `s5` is shifted by the immediate amount, and the result is placed in the destination register.



Assembly code		Source register / Result			
	s5	1111 1111	0001 1100	0001 0000	1110 0111
`slli t0, s5, 7`	t0	1000 1110	0000 1000	0111 0011	1000 0000
`srli s1, s5, 17`	s1	0000 0000	0000 0000	0111 1111	1000 1110
`srai t2, s5, 3`	t2	1111 1111	1110 0011	1000 0010	0001 1100

Figure 6.3 Shift instructions with immediate shift amounts

As discussed in Section 5.2.5, shifting a value left by N is equivalent to multiplying it by 2^N. For example, `slli s0, s0, 3` multiplies `s0` by 8 (i.e., 2^3). Likewise, shifting a value right by N is equivalent to dividing it by 2^N. Arithmetic right shifts divide two's complement numbers, while logical right shifts divide unsigned numbers.

Logical shifts are also used with `and` and `or` instructions to extract or assemble bitfields. For example, the following code extracts bits 15:8 from `s7` and places them in the lower byte of `s6`. If `s7` is 0x1234ABCD, then `s6` will be 0xAB after this code completes.

```
srli s6, s7, 8
andi s6, s6, 0xFF
```

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations because multiplying two N-bit numbers produces a $2N$-bit product. The RISC-V architecture provides various *multiply instructions* that result in 32- or 64-bit products. These instructions are not part of RV32I but are included in the RVM (RISC-V multiply/divide) extension.

The *multiply* instruction (`mul`) multiplies two 32-bit numbers and produces a 32-bit product. `mul s1, s2, s3` multiplies the values in `s2` and `s3` and places the least significant 32 bits of the product in `s1`; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose product fits in 32 bits. The bottom 32 bits of the product are the same whether the operands are viewed as signed or unsigned.

Three versions of the "multiply high" operation exist: `mulh`, `mulhsu`, and `mulhu`. These instructions put the high 32 bits of the multiplication result in the destination register. `mulh` (*multiply high signed signed*) treats both operands as signed. mulhsu (*multiply high signed unsigned*) treats the first operand as signed and the second as unsigned, and mulhu (*multiply high unsigned unsigned*) treats both operands as unsigned. For example, mulhsu t1, t2, t3 treats t2 as a 32-bit signed (two's complement) number and t3 as a 32-bit unsigned number, multiplies these two source operands, and puts the upper 32 bits of the result in t1. Using a series of two instructions—one of the "multiply high" instructions followed by the mul instruction—will place the entire 64-bit result of the 32-bit multiplication in the two registers designated by the user. For example, the following code multiplies 32-bit signed numbers in s3 and s5 and places the 64-bit product in t1 and t2. That is, {t1, t2} = s3 × s5.

```
mulh t1, s3, s5
mul  t2, s3, s5
```

6.3.3 Branching

Programs would be boring if they could only run in the same order every time, independent of the input. An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, *if/else* statements, *switch/case* statements, *while* loops, and *for* loops all conditionally execute code depending on some test. *Branch* instructions modify the flow of the program so that the processor can fetch instructions that are not in sequential order in memory. They modify the PC to skip over sections of code or to repeat previous code. *Conditional branch* instructions perform a test and branch only if the test is TRUE. Unconditional branch instructions, called *jumps*, always branch.

Conditional Branches

The RISC-V instruction set has six conditional branch instructions, each of which take two source registers and a label indicating where to go. beq (*branch if equal*) branches when the values in the two source registers are equal. bne (*branch if not equal*) branches when they are unequal. blt (*branch if less than*) branches when the value in the first source register is less than the value in the second, and bge (*branch if greater than or equal to*) branches when the first is greater than or equal to the second. blt and bge treat the operands as signed numbers, while bltu and bgeu treat the operands as unsigned.

Code Example 6.12 illustrates the use of beq. When the program reaches the branch if equal instruction (beq), the value in s0 is equal to the value in s1, so the branch is taken. Thus, the next instruction executed is the add instruction just after the label called target. The addi and sub instructions between the branch and the label are not executed.

There is no need for bgt or ble because these can be obtained by switching the order of the source registers of blt and bge. However, these are available as pseudoinstructions (see Section 6.3.8).

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

RISC-V Assembly Code

```
  addi  s0, zero, 4      # s0 = 0 + 4 = 4
  addi  s1, zero, 1      # s1 = 0 + 1 = 1
  slli  s1, s1, 2        # s1 = 1 << 2 = 4
  beq   s0, s1, target   # s0 == s1, so branch is taken
  addi  s1, s1, 1        # not executed
  sub   s1, s1, s0       # not executed
target:                  # label
  add   s1, s1, s0       # s1 = 4 + 4 = 8
```

Assembly code uses *labels* to indicate instruction locations in the program. A label refers to the instruction just after the label. When the assembly code is translated into machine code, these labels correspond to instruction addresses (as will be discussed in Sections 6.4.3 and 6.4.4). RISC-V assembly labels are followed by a colon (:). Most programmers indent their instructions but not the labels to help make labels stand out.

In Code Example 6.13, the branch is not taken because `s0` is equal to `s1`, and the code continues to execute directly after the `bne` (branch if not equal) instruction. All instructions in this code snippet are executed.

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`

RISC-V Assembly Code

```
  addi  s0, zero, 4      # s0 = 0 + 4 = 4
  addi  s1, zero, 1      # s1 = 0 + 1 = 1
  slli  s1, s1, 2        # s1 = 1 << 2 = 4
  bne   s0, s1, target   # branch not taken
  addi  s1, s1, 1        # s1 = 4 + 1 = 5
  sub   s1, s1, s0       # s1 = 5 − 4 = 1
target:
  add   s1, s1, s0       # s1 = 1 + 4 = 5
```

Jumps

A program can jump—that is, unconditionally branch—using one of three instructions: *jump* (`j`), *jump and link* (`jal`), or *jump register* (`jr`). `j` jumps directly to the instruction at the specified label. Code Example 6.14 shows the use of the `j` (jump) instruction to skip over three instructions and continue at the `add` instruction after the label `target`. After the `j` instruction executes, this program unconditionally continues executing the `add` instruction at the label `target`. All of the instructions between the jump and the label are skipped. We will discuss `jal` and `jr` instructions in Section 6.3.7, where they are used for function calls.

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`

RISC-V Assembly Code

```
  j    target        # jump to target
  srai s1, s1, 2     # not executed
  addi s1, s1, 1     # not executed
  sub  s1, s1, s0    # not executed
target:
  add  s1, s1, s0    # s1 = s1 + s0
```

6.3.4 Conditional Statements

If, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into RISC-V assembly language.

If Statements

An *if* statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.15 shows how to translate an if statement into RISC-V assembly code. The assembly code for the if statement tests the opposite condition of the one in the high-level code. In Code Example 6.15, the high-level code tests for `apples == oranges`. The assembly code tests for `apples != oranges` using `bne` to skip the if block if the condition is not satisfied. Otherwise (i.e., when `apples == oranges`), the branch is not taken, and the if block is executed.

In C and many other high-level programming languages, the double equals sign, ==, is an equality comparison, returning TRUE if both sides are equal. != is an inequality comparison.

Code Example 6.15 IF STATEMENT

High-Level Code

```
if (apples == oranges)
  f = g + h;
apples = oranges − h;
```

RISC-V Assembly Code

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # skip if (apples != oranges)
    add s2, s3, s4 # f = g + h
L1: sub s0, s1, s4 # apples = oranges − h
```

If/else Statements

If/else statements execute one of two blocks of code, depending on a condition. When the condition in the if statement is met, the *if* block is executed. Otherwise, the *else* block is executed. Code Example 6.16 shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In Code Example 6.16, the high-level code tests for `(apples == oranges)` and the assembly code tests for

Code Example 6.16 IF/ELSE STATEMENT

High-Level Code

```
if (apples == oranges)
  f = g + h;
else
  apples = oranges − h;
```

RISC-V Assembly Code

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
    bne s0, s1, L1 # skip if (apples != oranges)
    add s2, s3, s4 # f = g + h
    j   L2
L1: sub s0, s1, s4 # apples = oranges − h
L2:
```

(apples != oranges). If that opposite condition is TRUE, bne skips the if block and executes the else block. Otherwise, the if block executes and finishes with a jump (j) past the else block.

Switch/case Statements*

Switch/case statements, also called simply *case* statements, execute one of several blocks of code, depending on the conditions. If no conditions are met, the *default* block is executed. A case statement is equivalent to a series of nested if/else statements. Code Example 6.17 shows two high-level code snippets with the same functionality: they calculate whether to dispense $20, $50, or $100 from an ATM (automatic

Code Example 6.17 SWITCH/CASE STATEMENTS

High-Level Code

```
switch (button) {
  case 1:  amt = 20;  break;




  case 2:  amt = 50;  break;




  case 3:  amt = 100; break;




  default: amt = 0;
}

// equivalent function using
// if/else statements
if      (button == 1)  amt = 20;
else if (button == 2)  amt = 50;
else if (button == 3)  amt = 100;
else                   amt = 0;
```

RISC-V Assembly Code

```
# s0 = button, s1 = amt

case1:
 addi  t0, zero, 1        # t0 = 1
 bne   s0, t0, case2      # button == 1?
 addi  s1, zero, 20       # if yes, amt = 20
 j     done               # break out of case
case2:
 addi  t0, zero, 2        # t0 = 2
 bne   s0, t0, case3      # button == 2?
 addi  s1, zero, 50       # if yes, amt = 50
 j     done               # break out of case
case3:
 addi  t0, zero, 3        # t0 = 3
 bne   s0, t0, default    # button == 3?
 addi  s1, zero, 100      # if yes, amt = 100
 j     done               # break out of case
default:
 add   s1, zero, zero     # amt=0
done:
```

teller machine) depending on the button pressed. The RISC-V assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code, depending on a condition. While loops and for loops are commonly used in high-level languages. This section shows how to translate them into RISC-V assembly language, taking advantage of conditional branching.

While Loops

While loops repeatedly execute a block of code while a condition is met—that is, *until* a condition is *not* met. The while loop in Code Example 6.18 determines the value of x such that $2^x = 128$. It executes seven times, until pow = 128.

Code Example 6.18 WHILE LOOP

High-Level Code

```
// determines the power
// of x such that 2^x =128
int pow = 1;
int x   = 0;


while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

RISC-V Assembly Code

```
#  s0 = pow, s1 = x
       addi s0,  zero, 1      # pow = 1
       add  s1,  zero, zero   # x = 0

       addi t0,  zero, 128    # t0 = 128
while: beq  s0,  t0, done     # pow = 128?
       slli s0,  s0, 1        # pow = pow * 2
       addi s1,  s1, 1        # x = x + 1
       j    while             # repeat loop
done:
```

Like if/else statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, s0 == 128), the while loop is finished. Otherwise, the branch isn't taken and the loop body executes. Code Example 6.18 initializes pow to 1 and x to 0 before the while loop. The while loop compares pow to 128 and exits the loop if it is equal. Otherwise, it doubles pow (using a left shift), increments x, and branches back to the start of the while loop.

Do/while loops are similar to while loops, but they execute the body of the loop once before checking the condition. Code Example 6.19 illustrates such a loop. Notice that, unlike previous examples, the branch checks the same condition as in the high-level code.

For Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time

Code Example 6.19 DO/WHILE LOOP

High-Level Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

do {
 pow = pow * 2;
 x = x + 1;
} while (pow != 128);
```

RISC-V Assembly Code

```
#  s0 = pow, s1 = x
       addi s0,  zero, 1          # pow = 1
       add  s1,  zero, zero       # x = 0

       addi t0,  zero, 128        # t0 = 128
while: slli s0,  s0, 1            # pow = pow * 2
       addi s1,  s1, 1            # x = x + 1
       bne  s0,  t0, while        # pow = 128?
done:
```

through the while loop. *For* loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The high-level code format of the for loop is:

```
for (initialization; condition; loop operation)
  statement
```

The initialization code executes *before* the for loop begins. The condition is tested at the *beginning of each* loop iteration. If the condition is not met, the loop exits. If the condition is met, the statement (or statements) in the loop body are executed. The loop operation executes at the *end* of each loop iteration.

Code Example 6.20 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as `i` is less than 10. Note that this example also illustrates relative comparisons. The loop checks the < condition to continue, so the assembly code checks the opposite condition, >=, to exit the loop.

For loops are especially useful for accessing large amounts of similar data stored in memory arrays, which are discussed next.

Code Example 6.20 FOR LOOP

High-Level Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

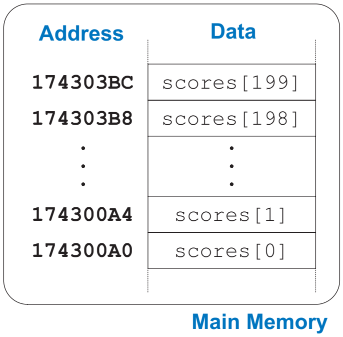
RISC-V Assembly Code

```
# s0 = i, s1 = sum
      addi  s1, zero, 0      # sum = 0
      addi  s0, zero, 0      # i = 0
      addi  t0, zero, 10     # t0 = 10
for:  bge   s0, t0,  done    # i >= 10?
      add   s1, s1,  s0      # sum = sum + i
      addi  s0, s0,  1       # i = i + 1
      j     for              # repeat loop
done:
```

6.3.6 Arrays

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array. Figure 6.4 shows a 200-element array of integer scores stored in memory. Each consecutive element address increases by 4, the number of bytes in an integer. The address of the 0th element of an array is called the array's *base address*.

Code Example 6.21 is a grade inflation algorithm that adds 10 points to each of the scores. The code for initializing the `scores` array is not shown. Assume that `s0` is initially 0x174300A0, the base address of the array. The index into the array is a variable (`i`) that increments by 1 for each array element, so we multiply it by 4 before adding it to the base address.



Address	Data
174303BC	scores[199]
174303B8	scores[198]
⋮	⋮
174300A4	scores[1]
174300A0	scores[0]

Figure 6.4 **Memory holding** `scores[200]` **starting at base address 0x174300A0**

Code Example 6.21 USING A FOR LOOP TO ACCESS AN ARRAY

High-Level Code

```
int i;
int scores[200];




for (i = 0; i < 200; i = i + 1)




 scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i


 addi s1, zero, 0   # i = 0
 addi t2, zero, 200 # t2 = 200

for:
 bge  s1, t2, done  # if i >= 200 then done
 slli t0, s1, 2     # t0 = i * 4
 add  t0, t0, s0    # address of scores[i]
 lw   t1, 0(t0)     # t1 = scores[i]
 addi t1, t1, 10    # t1 = scores[i] + 10
 sw   t1, 0(t0)     # scores[i] = t1
 addi s1, s1, 1     # i = i + 1
 j    for           # repeat
done:
```

Bytes and Characters

Numbers in the range [–128, 127] can be stored in a single byte rather than an entire word. Because the English language keyboard has fewer than 256 characters, English characters are often represented using bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.2 shows these character

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see *www.unicode.org*.

Table 6.2 **ASCII encodings**

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	−	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

The *load byte* (`lb`), *load byte unsigned* (`lbu`), and *store byte* (`sb`) instructions access individual bytes in memory. `lb` sign-extends the byte, whereas `lbu` zero-extends the byte to fill the entire 32-bit register. `sb` stores the least significant byte of the 32-bit register into the specified byte address in memory. All three instructions are illustrated in Figure 6.5, with the base address, `s4`, being 0xD0. `lbu s1, 2(s4)` loads the byte at memory address 0xD2 into the least significant byte of `s1` and fills the remaining register bits with 0. `lb s2, 3(s4)` loads the byte at memory address 0xD3 into the least significant byte of `s2` and sign-extends the byte into the upper 24 bits of the register. `sb s3, 1(s4)` stores the least significant byte of `s3` (0x9B) into memory byte address 0xD1; it replaces 0x42 with 0x9B. No other memory bytes are changed, and the more significant bytes of `s3` are ignored.

RISC-V also defines `lh`, `lhu`, and `sh` *half-word* loads and stores that operate on 16-bit data. Memory addresses for these instructions must be half-word aligned.

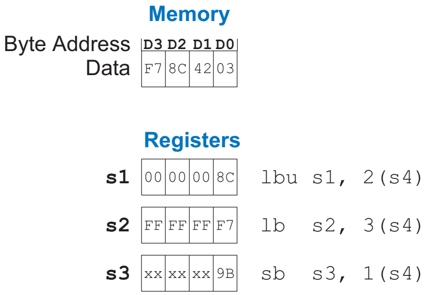


Figure 6.5 Instructions for loading and storing bytes

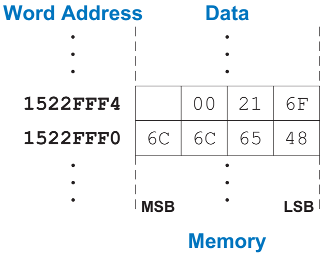


Figure 6.6 The string "Hello!" stored in memory

A series of characters, such as a word or sentence, is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.6 shows the string "Hello!" (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

Example 6.2 USING `lb` AND `sb` TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into RISC-V assembly language. Remember that array elements are now 1 byte, not 4 bytes, so consecutive elements are in consecutive addresses. Assume that `s0` already holds the base address of `chararray`.

```
// high-level code
// chararray[10] was declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] − 32;
```

Solution

```
# RISC-V assembly code
# s0 = base address of chararray (initialized earlier), s1 = i
      addi s1, zero, 0        # i = 0
      addi t3, zero, 10       # t3 = 10
for:  bge  s1, t3, done       # i >= 10 ?
      add  t4, s0, s1         # t4 = address of chararray[i]
      lb   t5, 0(t4)          # t5 = chararray[i]
      addi t5, t5, −32        # t5 = chararray[i] − 32
      sb   t5, 0(t4)          # chararray[i] = t5
      addi s1, s1, 1          # i = i + 1
      j    for                # repeat loop
done:
```

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as –, – …, – . – . , and – .., respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001. However, the 32 possible encodings of this 5-bit code were not sufficient for all keyboard characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

6.3.7 Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions may have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In RISC-V programs, the caller conventionally places up to eight arguments in registers `a0` to `a7` before making the function call, and the callee places the return value in register `a0` before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

RISC-V actually provides two registers for the return value, `a0` and `a1`. This allows for 64-bit return values, such as `int64_t`.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the *return address register* `ra` at the same time it jumps to the callee using the jump and link instruction (`jal`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (`s0–s11`), the return address (`ra`), and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It also shows how functions access arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

RISC-V uses the *jump and link* instruction (`jal`) to call a function and the *jump register* instruction (`jr`) to return from a function. Code Example 6.22 shows the `main` function calling the `simple` function. `main` is the caller and `simple` is the callee. The `simple` function

Code Example 6.22 `simple` **FUNCTION CALL**

High-Level Code

```
int main() {
  simple();
  ...
}

// void means the function
// returns no value
void simple() {
  return;
}
```

RISC-V Assembly Code

```
0x00000300 main:   jal  simple   # call function
0x00000304 ...
...        ...





0x0000051c simple: jr   ra       # return
```

is called with no input arguments and generates no return value; it just returns to the caller. In Code Example 6.22, example instruction addresses are given to the left of each RISC-V instruction.

`jal` and `jr ra` are the two essential instructions needed for a function call and return. In Code Example 6.22, the `main` function calls the `simple` function by executing `jal simple`, which performs two tasks: it jumps to the target instruction located at `simple` (0x0000051C) and it stores the *return address*, the address of the instruction after `jal` (in this case, 0x00000304) in the return address register (`ra`). The programmer can specify which register gets written with the return address, but the default is `ra`. So, `jal simple` is equivalent to `jal ra, simple` and is the preferred style. The `simple` function returns immediately by executing the instruction `jr ra`, which jumps to the instruction address held in `ra`. The `main` function then continues executing at this address (0x00000304).

The instruction address of the currently executing instruction is held in PC, the program counter. So, the following instruction address is referred to as PC+4.

Input Arguments and Return Values

The `simple` function in Code Example 6.22 is not very useful because it receives no input from the calling function (`main`) and returns no output. By RISC-V convention, functions use `a0` to `a7` for input arguments and `a0` for the return value. In Code Example 6.23, the function `diffofsums` is called with four arguments and returns one result. `result` is a local variable, which we choose to keep in `s3`. (Saving and restoring registers will be discussed soon.)

According to RISC-V convention, the calling function, `main`, places the function arguments from left to right into the input registers, `a0` to `a7`, before calling the function. The called function, `diffofsums`, stores

Code Example 6.23 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main(){
  int y;
  . . .




  y = diffofsums(2, 3, 4, 5);
  . . .
}

int diffofsums(int f, int g, int h, int i){
  int result;

  result = (f + g) − (h + i);

  return result;
}
```

RISC-V Assembly Code

```
# s7 = y
main:
   . . .
   addi a0, zero, 2   # argument 0 = 2
   addi a1, zero, 3   # argument 1 = 3
   addi a2, zero, 4   # argument 2 = 4
   addi a3, zero, 5   # argument 3 = 5
   jal  diffofsums    # call function
   add  s7, a0, zero  # y = returned value
   . . .
# s3 = result
diffofsums:
   add  t0, a0, a1    # t0 = f+g
   add  t1, a2, a3    # t1 = h+i
   sub  s3, t0, t1    # result = (f+g)−(h+i)
   add  a0, s3, zero  # put return value in a0
   jr   ra            # return to caller
```

`j` and `jr` are *pseudoinstructions*. They are not part of the instruction set but are convenient for programmers to use. The RISC-V assembler replaces them with actual RISC-V instructions. The assembler replaces `j target` with `jal zero, target`, which jumps and discards the return address by writing it to the `zero` register; and the assembler replaces `jr ra` with `jalr zero, ra, 0`.

The jump and link register instruction (`jalr`) is like `jal`, but it takes the destination address from a register, optionally added to a 12-bit signed immediate. For example, `jalr ra, s1, 0x4C` jumps to address `s1` + `0x4C` and puts PC+4 in `ra`.

the return value in the return register, a0. When a function with more than eight arguments is called, the additional input arguments are placed on the stack, which we discuss next.

The Stack

The stack is memory that is used as scratch space—that is, to save temporary information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary values, we explain how the stack works.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* (or placed) onto the stack (the top dish) is the first one that can be *popped* off (removed). Each function may allocate stack space to store local variables and to use as scratch space, but the function must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the RISC-V stack grows down in memory. That is, the stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.7 shows a picture of the stack. The stack pointer, sp (register 2), is an ordinary RISC-V register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. sp points to (gives the address of) data. For example, in Figure 6.7(a), the stack pointer, sp, holds the address value 0xBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (sp) starts at a high memory address and decrements to expand as needed. Figure 6.7(b) shows the stack expanding to allow two more data words of temporary storage. To do so, sp decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, a

The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. Some architectures also allow for *ascending stacks* that grow up toward higher memory addresses. The stack pointer (sp) typically points to the topmost element on the stack; this is called a *full stack*. Some architectures, such as ARM, also allow for *empty stacks* in which sp points one word beyond the top of the stack. The RISC-V architecture defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate. It specifies a *full descending* stack, which we will use in this chapter.

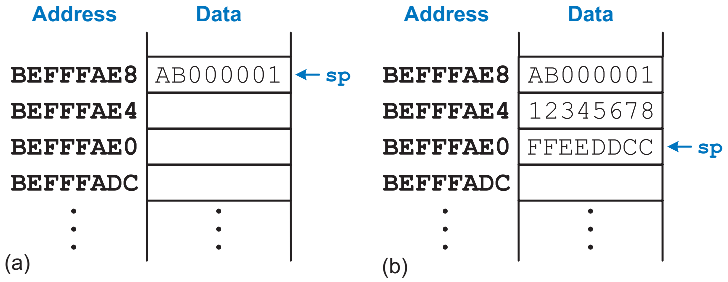


Figure 6.7 The stack (a) before expansion and (b) after two-word expansion

function should not modify any registers besides a0, the one containing the return value. The diffofsums function in Code Example 6.23 violates this rule because it modifies t0, t1, and s3. If main had been using these registers before the call to diffofsums, their contents would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them and then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

Code Example 6.24 shows an improved version of diffofsums that saves and restores t0, t1, and s3. Figure 6.8 shows the stack before, during,

Code Example 6.24 FUNCTION THAT SAVES REGISTERS ON THE STACK

High-Level Code

```
int diffofsums(int f, int g, int h, int
i){
 int result;







 result = (f + g) − (h + i);





 return result;
}
```

RISC-V Assembly Code

```
# s3 = result
diffofsums:
  addi sp, sp, −12    # make space on stack to
                      # store three registers
  sw   s3, 8(sp)      # save s3 on stack
  sw   t0, 4(sp)      # save t0 on stack
  sw   t1, 0(sp)      # save t1 on stack
  add  t0, a0, a1     # t0 = f + g
  add  t1, a2, a3     # t1 = h + i
  sub  s3, t0, t1     # result = (f + g) − (h + i)
  add  a0, s3, zero   # put return value in a0
  lw   s3, 8(sp)      # restore s3 from stack
  lw   t0, 4(sp)      # restore t0 from stack
  lw   t1, 0(sp)      # restore t1 from stack
  addi sp, sp, 12     # deallocate stack space
  jr   ra             # return to caller
```

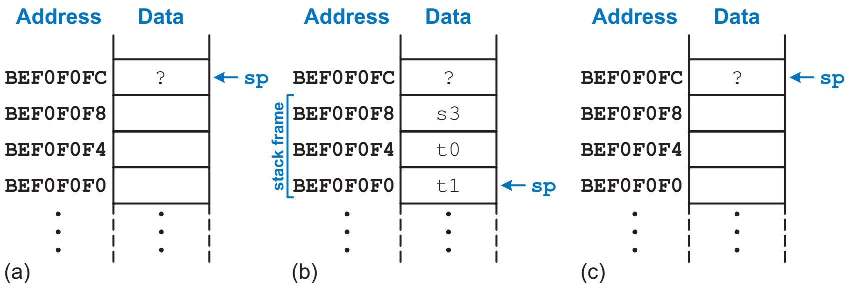


Figure 6.8 The stack: (a) before, (b) during, and (c) after the diffofsums function call

and after a call to the `diffofsums` function from Code Example 6.24. The stack starts at 0xBEF0F0FC. `diffofsums` makes room for three words on the stack by decrementing the stack pointer `sp` by 12. It then stores the current values held in `t0`, `t1`, and `s3` in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, `a0` holds the result, but there are no other side effects: `t0`, `t1`, `s3`, and `sp` have the same values as they did before the function call.

Saving a register value on the stack is called *pushing* a register onto the stack. Restoring the register value from the stack is called *popping* a register off of the stack.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`' stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Preserved Registers

Code Example 6.24 assumes that all of the used registers (`t0`, `t1`, and `s3`) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, RISC-V divides registers into *preserved* and *nonpreserved* categories. Preserved registers must contain the same values at the beginning and end of a called function because the caller expects preserved register values to be the same after the call.

The preserved registers are `s0` to `s11` (hence their name, *saved*), `sp`, and `ra`. The nonpreserved registers, also called *scratch* registers, are `t0` to `t6` (hence their name, *temporary*) and `a0` to `a7`, the argument registers. A function can change the nonpreserved registers freely but must save and restore any of the preserved registers that it uses.

Code Example 6.25 shows a further improved version of `diffofsums` that saves only `s3` on the stack. `t0` and `t1` are nonpreserved registers, so they need not be saved.

Code Example 6.25 FUNCTION THAT SAVES PRESERVED REGISTERS ON THE STACK

RISC-V Assembly Code

```
# s3 = result
diffofsums:
  addi sp, sp, −4     # make space on stack to store one register
  sw   s3, 0(sp)      # save s3 on stack
  add  t0, a0, a1     # t0 = f + g
  add  t1, a2, a3     # t1 = h + i
  sub  s3, t0, t1     # result = (f + g) − (h + i)
  add  a0, s3, zero   # put return value in a0
  lw   s3, 0(sp)      # restore s3 from stack
  addi sp, sp, 4      # deallocate stack space
  jr   ra             # return to caller
```

Table 6.3 Preserved and nonpreserved registers and memory

Preserved (*callee*-saved)	Nonpreserved (*caller*-saved)
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp`	
Stack above the stack pointer	Stack below the stack pointer

Because a callee function may freely change any nonpreserved registers, the caller must save any nonpreserved registers containing essential information before making a function call and then restore these registers afterward. For these reasons, preserved registers are also called *callee-saved* and nonpreserved registers are called *caller-saved*.

Table 6.3 summarizes which registers are preserved. The convention of which registers are preserved or not preserved is part of the standard calling convention[2] for the RISC-V Architecture, instead of being part of the architecture itself.

`s0` to `s11` are generally used to hold local variables within a function, so they must be saved. `ra` must also be saved so that the callee knows where to return. `t0` to `t6` are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved across a function call, and it is rare that the caller needs to save them. `a0` to `a7` are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns.

The stack above the stack pointer is automatically preserved, as long as the callee does not write to memory addresses above `sp`. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from `sp` at the beginning of the function.

The astute reader or an optimizing compiler may notice that `diffofsums`' local variable, `result`, is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store the calculation directly in the return register `a0`, eliminating the need to both allocate space on the stack frame and move the result from `s3` to `a0`. Code Example 6.26 shows this even further optimized `diffofsums` function.

[2] From the *RISC-V Instruction Set Manual*, Volume I, version 2.2 © 2017.

Code Example 6.26 OPTIMIZED `diffofsums` FUNCTION

RISC-V Assembly Code

```
# a0 = result
diffofsums:
  add t0, a0, a1  # t0 = f + g
  add t1, a2, a3  # t1 = h + i
  sub a0, t0, t1  # result = (f + g) − (h + i)
  jr  ra          # return to caller
```

Nonleaf Function Calls

A function that does not call other functions is called a *leaf function*; `diffofsums` is an example. A function that does call others is called a *nonleaf function*. Nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically, they must follow these rules:

> **Caller save rule**: Before a function call, the caller must save any nonpreserved registers (`t0-t6` and `a0-a7`) that it needs after the call. After the call, it must restore these registers before using them.
>
> **Callee save rule**: Before a callee disturbs any of the preserved registers (`s0-s11` and `ra`), it must save the registers. Before it returns, it must restore these registers.

Code Example 6.27 shows a nonleaf function `f1` and a leaf function `f2`, including all of the necessary saving and preserving of registers. `f1` keeps `i` in `s4` and `x` in `s5`; `f2` keeps `r` in `s4`. `f1` uses preserved registers `s4`, `s5`, and `ra`, so it initially pushes them onto the stack according to the callee save rule. It uses `t3` to hold the intermediate result (`a-b`) so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` saves `a0` and `a1` onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. `ra` changes because it is overwritten by the call to `f2`. Although `t3` is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs `t3` and does not have to save it. `f1` then passes the argument to `f2` in `a0`, makes the function call, and gets the result in `a0`. `f1` then restores `a0` and `a1` because it still needs them. When `f1` is done, it puts the return value in `a0`, restores registers `s4`, `s5`, `ra`, and `sp`, and returns. `f2` saves and restores `s4` (and `sp`) according to the callee save rule.

On careful inspection, one might note that `f2` does not modify `a1`, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call. An

A nonleaf function overwrites `ra` when it calls another function using `jal`. Thus, a nonleaf function must always save `ra` on its stack and restore it before returning.

Code Example 6.27 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a − b);

  for (i = 0; i < a; i++)

     x = x + f2(b + i);

  return x;
 }

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

RISC-V Assembly Code

```
# a0 = a, a1 = b, s4 = i, s5 = x
f1:
  addi sp, sp, −12      # make room on stack for 3 registers
  sw   ra, 8(sp)        # save preserved registers used by f1
  sw   s4, −4(sp)
  sw   s5, 0(sp)
  add  s5, a0, a1       # x = (a + b)
  sub  t3, a0, a1       # temp = (a − b)
  mul  s5, s5, t3       # x = x * temp = (a + b) * (a − b)
  addi s4, zero, 0      # i = 0
for:
  bge  s4, a0, return   # if i >= a, exit loop
  addi sp, sp, −8       # make room on stack for 2 registers
  sw   a0, 4(sp)        # save nonpreserved regs. on stack
  sw   a1, 0(sp)
  add  a0, a1, s4       # argument is b + i
  jal  f2               # call f2(b + i)
  add  s5, s5, a0       # x = x + f2(b + i)
  lw   a0, 4(sp)        # restore nonpreserved registers
  lw   a1, 0(sp)
  addi sp, sp, 8
  addi s4, s4, 1        # i++
  j    for              # continue for loop
return:
  add  a0, zero, s5     # return value is x
  lw   ra, 8(sp)        # restore preserved registers
  lw   s4, 4(sp)
  lw   s5, 0(sp)
  addi sp, sp, 12       # restore sp
  jr   ra               # return from f1

# a0 = p,  s4 = r
f2:
  addi sp, sp, −4       # save preserved regs. used by f2
  sw   s4, 0(sp)
  addi s4, a0, 5        # r = p + 5
  add  a0, s4, a0       # return value is r + p
  lw   s4, 0(sp)        # restore preserved registers
  addi sp, sp, 4        # restore sp
  jr   ra               # return from f2
```

optimizing compiler could observe that f2 is a leaf procedure and could allocate r to a nonpreserved register, avoiding the need to save and restore s4. Figure 6.9 shows the stack during execution of the functions. For this example, the stack pointer originally starts at 0xBEF7FF0C.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$. The factorial function can be written recursively as $factorial(n) = n \times factorial(n - 1)$, as shown in Code Example 6.28. The

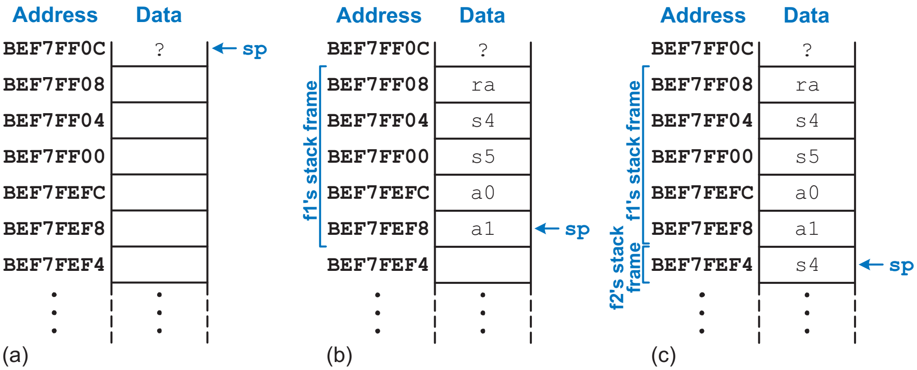


Figure 6.9 **The stack: (a) before function calls, (b) during `f1`, and (c) during `f2`**

Code Example 6.28 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;







  else
    return (n * factorial(n − 1));
}
```

RISC-V Assembly Code

```
0x8500 factorial: addi sp, sp, −8       # make room for a0, ra
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1      # temporary = 1
0x8510            bgt  a0, t0, else     # if n > 1, go to else
0x8514            addi a0, zero, 1      # otherwise, return 1
0x8518            addi sp, sp, 8        # restore sp
0x851C            jr   ra               # return
0x8520 else:      addi a0, a0, −1       # n = n − 1
0x8524            jal  factorial        # recursive call
0x8528            lw   t1, 4(sp)        # restore n into t1
0x852C            lw   ra, 0(sp)        # restore ra
0x8530            addi sp, sp, 8        # restore sp
0x8534            mul  a0, t1, a0       # a0 = n * factorial(n − 1)
0x8538            jr   ra               # return
```

factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address 0x8500. According to the callee save rule, `factorial` is a nonleaf function and must save `ra`. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save `a0`. Hence, it pushes both registers onto the stack at the start. It then checks whether `n` ≤ 1. If so, it puts the return value of 1 in `a0`, restores the stack pointer, and returns to the caller. It does not have to restore `ra` in this case, because it was never modified. If `n` > 1, the function recursively calls `factorial(n−1)`. It then restores the value of `n` and the return address register (`ra`) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into `t1` so as not to overwrite the returned value. The multiply instruction (`mul a0, t1, a0`) multiplies `n` (`t1`) and the returned value (`a0`) and puts the result in `a0`, the return register.

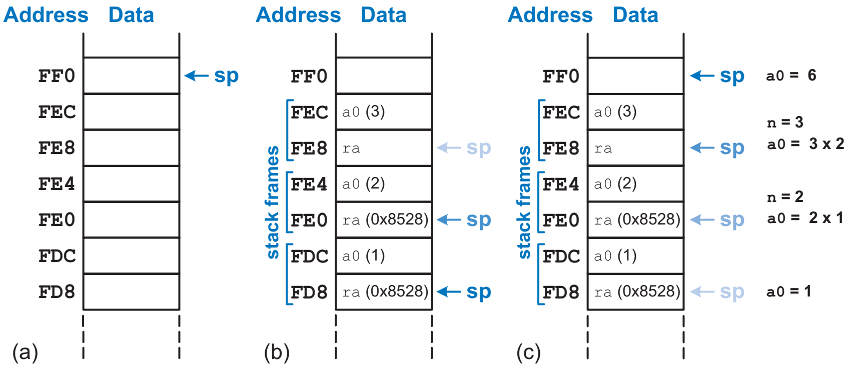


Figure 6.10 **Stack: (a) before, (b) during, and (c) after `factorial` function call with `n` = 3**

For clarity, we save registers at the start of a function call. An optimizing compiler might observe that there is no need to save `a0` and `ra` when `n` ≤ 1 and, thus, save registers on the stack only in the `else` portion of the function.

Figure 6.10 shows the stack when executing `factorial(3)`. For illustration, we show `sp` initially pointing to 0xFF0 (the upper address bits are 0), as shown in Figure 6.10(a). The function creates a two-word stack frame to hold `n` (`a0`) and `ra`. On the first invocation, `factorial` saves `a0` (holding `n` = 3) at 0xFEC and `ra` at 0xFE8, as shown in Figure 6.10(b). The function then changes `n` to 2 and recursively calls `factorial(2)`, making `ra` hold 0x8528. On the second invocation, it saves `a0` (holding `n` = 2) at 0xFE4 and `ra` at 0xFE0. This time, we know that `ra` contains 0x8528. The function then changes `n` to 1 and recursively calls `factorial(1)`. On the third invocation, it saves `a0` (holding `n` = 1) at 0xFDC and `ra` at 0xFD8. This time, `ra` again contains 0x8528. The third invocation of `factorial` returns the value 1 in `a0` and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` (into `t1`) to 2, restores `ra` to 0x8528 (it happened to already have this value), deallocates the stack frame, and returns `a0` = 2 × 1 = 2 to the first invocation. The first invocation restores `n` (into `t1`) to 3, restores `ra`, the return address of the caller, deallocates the stack frame, and returns `a0` = 3 × 2 = 6. Figure 6.10(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xFF0), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. `a0` holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than eight input arguments and may have too many local variables to keep in preserved registers. The stack is used to

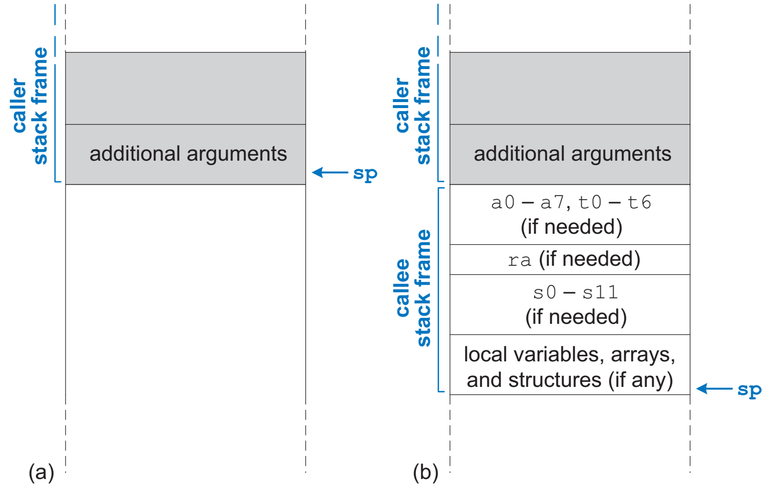


Figure 6.11 **Expanded stack frame with additional arguments (a) before call, (b) after call**

store this information. By RISC-V convention, if a function has more than eight arguments, the first eight are passed in the argument registers (`a0`–`a7`) as usual. Additional arguments are passed on the stack, just above `sp`. The caller must expand its stack to make room for the additional arguments. Figure 6.11(a) shows the caller's stack for calling a function with more than eight arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in `s0` to `s11`; if a function has too many local variables, they can also be stored in the function's stack frame. Local arrays and structures are also stored on the stack.

Figure 6.11(b) shows the organization of a callee's stack frame. The stack frame holds the temporary, argument, and return address registers (if they need to be saved because of a subsequent function call), and any of the saved registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than eight arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

Some functions also include a *frame pointer* that points to the *bottom* of the active stack frame – the stack frame of the executing function. By convention, this address is held in the `fp` register (`x8`), which is also a preserved register.

6.3.8 Pseudoinstructions

Before we show how to convert assembly code into machine code, 1's and 0's, let us revisit pseudoinstructions. Remember that RISC-V is a reduced instruction set computer (RISC), so the instruction size and hardware complexity are minimized by keeping the number of instructions small. However, RISC-V defines pseudoinstructions that are not actually part of the RISC-V instruction set but that are commonly used by programmers and compilers. When converted to machine code, pseudoinstructions are

Table 6.4 Pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
j label	jal zero, label	jump	PC = label
jr ra	jalr zero, ra, 0	jump register	PC = ra
mv t5, s3	addi t5, s3, 0	move	t5 = t3
not s7, t2	xori s7, t2, −1	one's complement	s7 = ~t2
nop	addi zero, zero, 0	no operation	
li s8, 0x7EF	addi s8, zero, 0x7EF	load 12-bit immediate	s8 = 0x7EF
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF	load 32-bit immediate	s8 = 0x56789DEF
bgt s1, t3, L3	blt t3, s1, L3	branch if >	if (s1 > t3), PC = L3
bgez t2, L7	bge t2, zero, L7	branch if ≥ 0	if (t2 ≥ 0), PC = L7
call L1	jal L1	call nearby function	PC = L1, ra = PC + 4
call L5	auipc ra, imm$_{31:12}$ jalr ra, ra, imm$_{11:0}$	call far away function	PC = L5, ra = PC + 4
ret	jalr zero, ra, 0	return from function	PC = ra

translated into one or more RISC-V instructions. For example, we have already discussed the jump (j) pseudoinstruction that is converted to the jump and link (jal) instruction with x0 as the destination address—that is, no return address is written. We also noted that logical NOT can be performed by XORing the source operand with all 1's.

Table 6.4 gives examples of pseudoinstructions and the RISC-V instructions used to implement them. For example, the move instruction (mv) copies the contents of one register to another register. The load immediate pseudoinstruction (li) loads a 32-bit constant using a combination of the lui and addi instructions. If the constant can be represented in 12 bits, li is translated into an addi instruction. The no operation pseudoinstruction (nop, pronounced "no op") performs no operation. The PC is incremented by 4 upon its execution, but no other registers or memory values are altered. The call pseudoinstruction makes a procedure call. If the call is to a nearby function, call is translated into a jalr instruction. However, if the function is far away, call is translated into two RISC-V instructions: auipc and jalr. For example, auipc s1,0xABCDE adds 0xABCDE000 to the PC and puts the result in s1. So, if PC is 0x02000000, then s1 now holds 0xADCDE000. jalr ra,s1,0x730 then jumps to address s1 + 0x730 (0xADCDE730) and puts PC+4 in ra. The ret pseudoinstruction returns from a function.

Nops are commonly used to generate precise delays in a program.

It translates into `jalr x0,ra,0`. Table B.7 in Appendix B lists the most common RV32I pseudoinstructions. Appendix B is printed on the inside covers of this textbook.

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's, called *machine language*. This section describes RISC-V machine language and the tedious process of converting between assembly and machine language.

RISC-V uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would also encourage a single-instruction format, but that is too restrictive. However, this issue allows us to introduce the last design principle:

Design Principle 4: Good design demands good compromises.

RISC-V makes the compromise of defining four main instruction formats: *R-type*, *I-type*, *S/B-type*, and *U/J-type*. This small number of formats allows for some regularity among instructions and, thus, simpler decoder hardware, while also accommodating different instruction needs. *R-type* (*register*) instructions, such as `add s0,s1,s2`, operate on three registers. *I-type* (immediate) instructions, such as `addi s3,s4,42`, and *S/B-type* (store/branch) instructions, such as `sw a0,4(sp)` or `beq a0,a1,L1`, operate on two registers and a 12- or 13-bit signed immediate. *U/J-type* (upper immediate/jump) instructions, such as `jal ra,factorial`, operate on one register and a 20- or 21-bit immediate. This section discusses these RISC-V machine instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all RV32I instructions.

S/B-type is notably not called B/S-type.

6.4.1 R-Type Instructions

R-type (*register*-type) instructions use three registers as operands: two as sources and one as a destination. Figure 6.12 shows the R-type machine

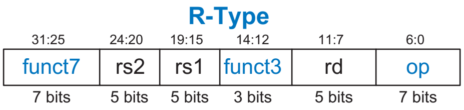
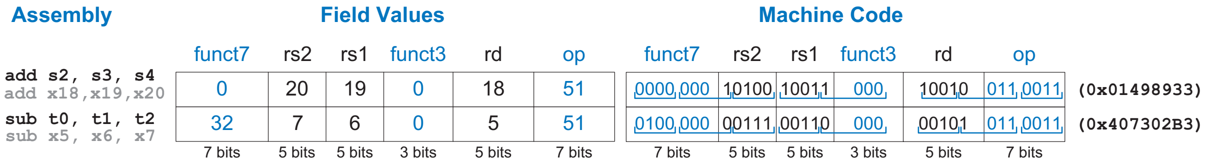


Figure 6.12 R-type instruction format



Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
add s2, s3, s4 add x18,x19,x20	0	20	19	0	18	51	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
sub t0, t1, t2 sub x5, x6, x7	32	7	6	0	5	51	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.13 Machine code for R-type instructions

instruction format. The 32-bit instruction has six fields: **funct7**, **rs2**, **rs1**, **funct3**, **rd**, and **op**. Each field is three to seven bits, as indicated.

The operation the instruction performs is encoded in the three fields highlighted in blue: 7-bit **op** (also called opcode or operation code) and 7- and 3-bit **funct7** and **funct3** (also called the function fields). The specific R-type operation is determined by the opcode and the function fields. These bits together are called the *control bits* because they control what operation to perform. For example, the opcode and function fields for the add instruction are **op** = 51 (0110011_2), **funct7** = 0 (0000000_2) and **funct3** = 0 (000_2). Similarly, the sub instruction has **op** = 51, **funct7** = 32 (0100000_2), and **funct3** = 0 (000_2). Figure 6.13 shows the machine code for two R-type instructions, add and sub. The two source registers and the destination register are encoded in the three fields: **rs1**, **rs2**, and **rd**. The fields contain the register numbers that were given in Table 6.1. For example, s0 is register 8 (x8). Notice that the registers are in the opposite order in the assembly and machine language instructions. For example, the assembly instruction add s2, s3, s4 has **rd** = s2 (18), **rs1** = s3 (19), and **rs2** = s4 (20). These registers are listed left to right in the assembly instruction but right to left in the machine instruction.

Table B.1 in Appendix B lists the opcode and the function fields (**funct3** and **funct7**) for RV32I instructions. The easiest way to translate from assembly to machine code (as shown in Figure 6.13) is to write out the values of each field and convert these values to binary. Then, group the bits into blocks of four to convert to hexadecimal and make the machine language representation more compact.

Appendix B is located on the inside covers of this book.

Other R-type instructions include shifts (sll, srl, and sra) and logical operations (and, or, and xor). Shift instructions with an immediate shift amount (slli, srli, and srai) are I-type instructions, which are discussed next in Section 6.4.2.

Figure 6.14 shows the machine code for shift left logical (sll) and xor. The opcode is 51 (0110011_2) for all R-type operations. Shift instructions with a register shift amount (sll, srl, and sra), shift **rs1** by the unsigned 5-bit value in bits 4:0 of register **rs2**, and place the result in **rd**. For all shift instructions, **funct7** and **funct3** encode the type of shift or logical operation to perform, as given in Table B.1. For sll, **funct7** = 0 and **funct3** = 1; xor uses **funct7** = 0 and **funct3** = 4.

Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`sll s7, t0, s1` `sll x23,x5, x9`	0	9	5	1	23	51	0000 000	01001	00101	001	10111	011 0011	(0x00929BB3)
`xor s8, s9, s10` `xor x24,x25,x26`	0	26	25	4	24	51	0000 000	11010	11001	100	11000	011 0011	(0x01ACCC33)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.14 More machine code for R-type instructions

R-type instructions have 17 bits of **op** and **funct** codes, enough to represent 2^{17} = 131,072 different instructions. This seems grossly excessive, considering we have defined less than a dozen R-type instructions so far. However, only 15 other bits are needed to encode the source and destination registers. This large instruction set space makes RISC-V highly extensible. For example, the *RISC-V F Extension* adds floating-point instructions, described further in Section 6.6.4 and Appendix B.

Example 6.3 TRANSLATING R-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following RISC-V assembly instruction into machine language:

```
add t3, s4, s5
```

Solution According to Table 6.1, `t3`, `s4`, and `s5` are registers 28, 20, and 21. According to Table B.1, `add` has an opcode of 51 (0110011_2) and function codes of funct7 = 0 and funct3 = 0. Thus, the fields and machine code are given in Figure 6.15. The easiest way to write the machine language in hexadecimal is to first write it in binary, then look at consecutive groups of four bits, which correspond to hexadecimal digits (indicated by blue underbars). Hence, the machine language instruction is 0x015A0E33.

Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`add t3, s4, s5` `add x28,x20,x21`	0	21	20	0	28	51	0000 000	10101	10100	000	11100	011 0011	(0x015A0E33)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.15 Machine code for the R-type instruction of Example 6.3

6.4.2 I-Type Instructions

I-type (*immediate*) instructions use two register operands and one immediate operand. I-type instructions include `addi`, `andi`, `ori`, and `xori`, loads (`lw`, `lh`, `lb`, `lhu`, and `lbu`), and register jumps (`jalr`). Figure 6.16 shows the I-type machine instruction format. It is similar to R-type but includes a 12-bit immediate field **imm** instead of the **funct7** and **rs2** fields. **rs1** and **imm** are the source operands, and **rd** is the destination register.

Figure 6.17 shows several examples of encoding I-type instructions. The immediate field represents a 12-bit signed (two's complement)

I-Type

31:20	19:15	14:12	11:7	6:0
$\text{imm}_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

Figure 6.16 I-type instruction format

Assembly	Field Values					Machine Code					
	imm$_{11:0}$	rs1	funct3	rd	op	imm$_{11:0}$	rs1	funct3	rd	op	
addi s0, s1, 12 addi x8, x9, 12	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
addi s2, t1, -14 addi x18,x6, -14	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
lw t2, -6(s3) lw x7, -6(x19)	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
lb s4, 0x1F(s4) lb x20,0x1F(x20)	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
slli s2, s7, 5 slli x18, x23, 5	5	23	1	18	19	0000 0000 0101	10111	001	10010	001 0011	(0x005B9913)
srai t1, t2, 29 srai x6, x7, 29	(upper 7 bits = 32) 29	7	5	6	19	0100 0001 1101	00111	101	00110	001 0011	(0x41D3D313)
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.17 Machine code for I-type instructions

number for all I-type instructions except immediate shift instructions (slli, srli, and srai). For these shift instructions, **imm**$_{4:0}$ is the 5-bit unsigned shift amount; the upper seven **imm** bits are 0 for srli and slli, but srai puts a 1 in **imm**$_{10}$ (i.e., instruction bit 30), as shown in Figure 6.17. As with R-type instructions, the order of the operands in the I-type assembly instructions differs from that of the machine instruction.

Example 6.4 TRANSLATING I-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following assembly instruction into machine language.

```
lw t3, −36(s4)
```

Solution According to Table 6.1, t3, and s4 are registers 28 and 20. **rs1** (s4 = x20) specifies the base address, and **rd** (t3 = x28) specifies the destination. The immediate, **imm**, encodes the 12-bit offset (−36). Table B.1 indicates that lw has an **op** of 3 (0000011_2) and **funct3** of 2 (010_2). The fields and machine code are given in Figure 6.18.

I-type instructions have a 12-bit immediate field, but the immediates are used in 32-bit operations. For example, lw adds a 12-bit offset to a 32-bit base register. What should go in the upper 20 bits of the 32 bits? For positive immediates, the upper bits should be all 0's, but for negative

Assembly	Field Values					Machine Code					
	imm$_{11:0}$	rs1	funct3	rd	op	imm$_{11:0}$	rs1	funct3	rd	op	
lw t3, -36(s4) lw x28, -36(x20)	-36	20	2	28	3	1111 1101 1100	10100	010	11100	000 0011	(0xFDCA2E03)
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.18 Machine code for the I-type instruction of Example 6.4

immediates, the upper bits should be all 1's. Recall from Section 1.4.6 that this is called *sign extension*.

Remember that an M-bit two's complement number is sign-extended to an N-bit number ($N>M$) by copying the sign bit (most significant bit) of the M-bit number into all of the upper bits of the N-bit number. Sign-extending a two's complement number does not change its value. For example, 1101_2 is a 4-bit two's complement number representing -3_{10}. When sign-extended to 8 bits, it becomes 11111101_2 and still represents -3_{10}.

6.4.3 S/B-Type Instructions

Like I-type instructions, S/B-type (*store/branch*) instructions use two register operands and one immediate operand. However, both of the register operands are source registers (**rs1** and **rs2**) in S/B-type, whereas I-type instructions use one source register (**rs1**) and one destination register (**rd**). Figure 6.19 shows the S/B-type machine instruction format. The instruction replaces the **funct7** and **rd** fields of R-type instructions with the 12-bit immediate **imm**. Thus, this immediate field is split across two bit ranges, bits 31:25 and bits 11:7, of the instruction.

Store instructions use S-type and branch instructions use B-type. S- and B-type formats differ only in how the immediate is encoded. S-type instructions encode a 12-bit signed (two's complement) immediate, with the top seven bits ($\mathbf{imm}_{11:5}$) in bits 31:25 of the instruction and the lower five bits ($\mathbf{imm}_{4:0}$) in bits 11:7 of the instruction.

B-type instructions encode a 13-bit signed immediate representing the *branch offset*, but only 12 of the bits are encoded in the instruction. The least significant bit is always 0, because branch amounts are always an even number of bytes, as will be explained later. The immediate of the B-type instruction is a somewhat strange bit swizzling of the immediate. $\mathbf{imm}_{12}$ is in instr_{31}; $\mathbf{imm}_{11}$ is in instr_{7}; $\mathbf{imm}_{10:5}$ is in $\text{instr}_{30:25}$; $\mathbf{imm}_{4:1}$ is in $\text{instr}_{11:8}$; and $\mathbf{imm}_0$ is always 0 and, hence, isn't part of the instruction. This bit mashup is done so that immediate bits occupy the same instruction bit across instruction formats as much as possible and so that the sign bit is always in instr_{31}, as will be described in Section 6.4.5.

Figure 6.20 shows several examples of encoding store instructions using the S-type format. **rs1** is the base address, **imm** is the offset, and **rs2** is the value to be stored to memory. Recall that negative immediate values are represented using 12-bit two's complement notation. For example, in `sw x7,-6(x19)`, register `x19` is the base address (**rs1**), `x7` is the second source (**rs2**), the value to be stored to memory, and −6 is the offset. For all S-type instructions, **op** is 35 (0100011_2) and **funct3** distinguishes between `sb` (0), `sh` (1), and `sw` (2).

In the `sw` assembly instruction, **rs2** is the leftmost register, that is: `sw rs2,offset(rs1)`

31:25	24:20	19:15	14:12	11:7	6:0	
$\text{imm}_{11:5}$	rs2	rs1	funct3	$\text{imm}_{4:0}$	op	S-Type
$\text{imm}_{12,10:5}$	rs2	rs1	funct3	$\text{imm}_{4:1,11}$	op	B-Type
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.19 S- and B-type instruction formats

Assembly	Field Values						Machine Code						
	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
sw t2, -6(s3) sw x7, -6(x19)	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
sh s4, 23(t0) sh x20,23(x5)	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
sb t5, 0x2D(zero) sb x30,0x2D(x0)	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.20 **Machine code for S-type instructions**

```
#Address    # RISC-V Assembly
0x70           beq  s0, t5, L1    1
0x74           add  s1, s2, s3    2
0x78           sub  s5, s6, s7    3
0x7C           lw   t0, 0(s1)     4
0x80       L1: addi s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past beq

$imm_{12:0}$ = 16	0	0	0	0	0	0	0	0	1	0	0	0	$\cancel{0}$
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values						Machine Code						
	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
beq s0, t5, L1 beq x8, x30, 16	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure 6.21 **B-type instruction format and calculations for** beq

Branches (beq, bne, blt, bge, bltu, and bgeu) use the B-type instruction format. Figure 6.21 shows some example code with the branch if equal instruction, beq. Instruction addresses are given to the left of each instruction. The *branch target address* (*BTA*) is the destination of the branch. The beq instruction in Figure 6.21 has a BTA of 0x80, the instruction address of the L1 label. The *branch offset* is sign-extended and added to the address of the branch instruction to obtain the branch target address.

For B-type instructions, **rs1** and **rs2** are the two source registers, and the 13-bit immediate branch offset, $\mathbf{imm}_{12:0}$, gives the *number of bytes* between the branch instruction and the BTA. In this case, the BTA is four instructions after the beq instruction, that is, 4 × 4 = 16 bytes past beq. Thus, the branch offset is 16. Only bits 12:1 are encoded in the instruction because bit 0 of the branch offset is always 0.

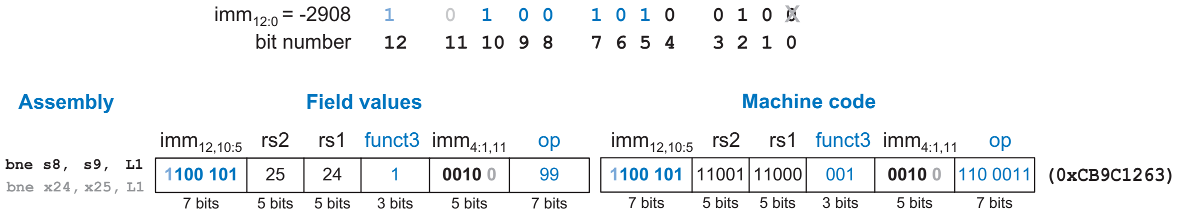
For 32-bit instructions, bits 1:0 of the 13-bit branch offset ($\mathbf{imm}_{12:0}$) are always zero because 32-bit instructions occupy 4 bytes of memory. Thus, instruction addresses are always divisible by four and neither bits 1 nor 0 of the branch offset need to be encoded in the instruction. However, RV32I only omits bit 0. This enables compatibility with 16-bit (2-byte) RISC-V compressed instructions (see Section 6.6.5). Compilers can then mix 16-bit and 32-bit instructions if the processor hardware supports both instruction sizes.

Example 6.5 ENCODING B-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Consider the following RISC-V assembly code snippet. The instruction address is written to the left of each instruction. Translate the branch if not equal (`bne`) instruction into machine code.

Address	Instruction
`0x354`	`L1: addi s1, s1, 1`
`0x358`	`sub t0, t1, s7`
`...`	`...`
`0xEB0`	`bne s8, s9, L1`

Solution According to Table 6.1, `s8` and `s9` are registers 24 and 25. So, **rs1** is 24 and **rs2** is 25. The label `L1` is 0xEB0 − 0x354 = 0xB5C (2908) bytes *before* the `bne` instruction. So, the 13-bit immediate is −2908 (1010010100100_2). From Appendix B, the **op** is 99 (1100011_2) and **funct3** is 1 (001_2). Thus, the machine code is given in Figure 6.22. Notice that branch instructions can branch forward (to higher addresses) or, as in this case, backward (to lower addresses).



`bne s8, s9, L1`
`bne x24, x25, L1`

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	25	24	1	0010 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	11001	11000	001	0010 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

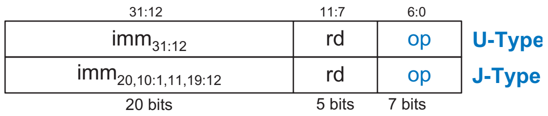
(0xCB9C1263)

Figure 6.22 Machine code for the B-type instruction of Example 6.5

6.4.4 U/J-Type Instructions

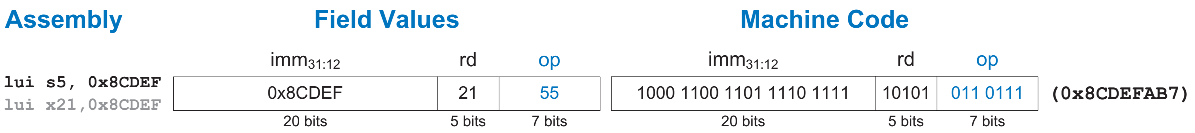
U/J-type (*upper immediate/jump*) instructions have one destination register operand **rd** and a 20-bit immediate field, as shown in Figure 6.23. Like other formats, U/J-type instructions have a 7-bit opcode. In U-type instructions, the remaining bits specify the most significant 20 bits of a 32-bit immediate. In J-type instructions, the remaining 20 bits specify the most significant 20 bits of a 21-bit immediate jump offset. As with B-type instructions, the least significant bit of the immediate is always 0 and is not encoded in the J-type instruction.

As with B-type instructions, the J-type immediate bits are oddly scrambled. Computers don't care, but this is annoying to humans.



31:12	11:7	6:0	
$imm_{31:12}$	rd	op	U-Type
$imm_{20,10:1,11,19:12}$	rd	op	J-Type
20 bits	5 bits	7 bits	

Figure 6.23 **U- and J-type instruction formats**



Assembly	Field Values			Machine Code			
	$imm_{31:12}$	rd	op	$imm_{31:12}$	rd	op	

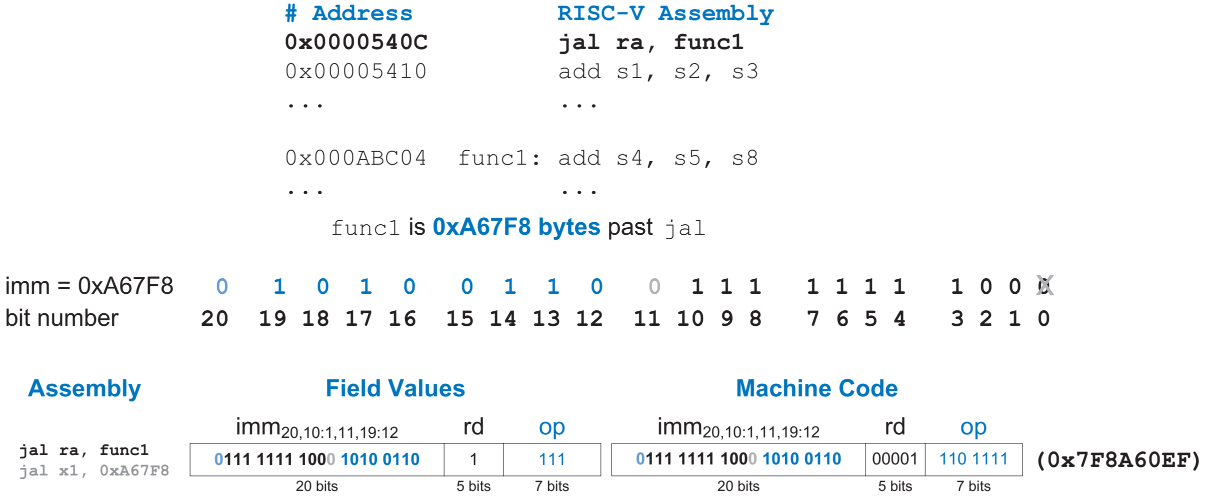
`lui s5, 0x8CDEF`
`lui x21,0x8CDEF`
	0x8CDEF	21	55	1000 1100 1101 1110 1111	10101	011 0111	(0x8CDEFAB7)

	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

Figure 6.24 **Machine code for U-type instruction** `lui`

Figure 6.24 shows the load upper immediate instruction, `lui`, translated into machine code. The 32-bit immediate consists of the upper 20 bits encoded in the instruction and 0's in the lower bits. So, in this case, after the instruction executes, register `s5` (**rd**) holds the value 0x8CDEF000.

Figure 6.25 shows some example code using the jump and link instruction, `jal`. The instruction address is written to the left of each instruction. Like branch instructions, J-type instructions jump to an instruction address that is relative to the current PC, that is, the instruction address of the `jal` instruction. In Figure 6.25, the jump target address (JTA) is 0xABC04, which is 0xA67F8 bytes past the `jal` instruction at address 0x540C because 0xABC04 – 0x540C = 0xA67F8 bytes. Like branch instructions, the least significant bit is not encoded in



```
# Address            RISC-V Assembly
0x0000540C           jal ra, func1
0x00005410           add s1, s2, s3
...                  ...

0x000ABC04   func1:  add s4, s5, s8
...                  ...
```

`func1` is **0xA67F8 bytes** past `jal`

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	$\cancel{0}$
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values			Machine Code			
	$imm_{20,10:1,11,19:12}$	rd	op	$imm_{20,10:1,11,19:12}$	rd	op	

`jal ra, func1`
`jal x1, 0xA67F8`
	0111 1111 1000 1010 0110	1	111	0111 1111 1000 1010 0110	00001	110 1111	(0x7F8A60EF)

	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

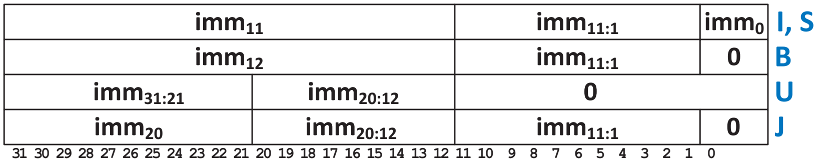
Figure 6.25 **Machine code for J-type instruction** `jal`

jalr is an I-type (*not* J-type!) instruction. jal is the only J-type instruction.

the instruction because it is always 0. The remaining bits are swizzled into the 20-bit immediate field, as shown in Figure 6.25. If a destination register, **rd**, is not specified by a `jal` assembly instruction, that field defaults to `ra` (`x1`). For example, the instruction `jal L1` is equivalent to `jal ra, L1` and has **rd** = 1. Ordinary jump (`j`) is encoded as `jal` with **rd** = 0.

6.4.5 Immediate Encodings

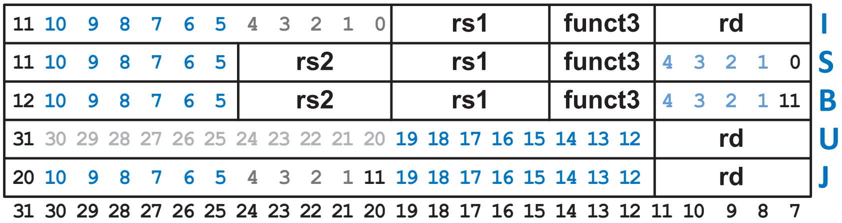
RISC-V uses 32-bit signed immediates. Only 12 to 21 bits of the immediate are encoded in the instruction. Figure 6.26 shows how immediates are formed for each instruction type. I- and S-type instructions encode 12-bit signed immediates. J- and B-type instructions use 21- and 13-bit signed immediates, where the least significant bit is always 0 (see Sections 6.4.3 and 6.4.4). U-type instructions encode the top 20 bits of a 32-bit immediate.



31:21	20:12	11:1	0	
imm_{11}		$\text{imm}_{11:1}$	imm_0	I, S
imm_{12}		$\text{imm}_{11:1}$	0	B
$\text{imm}_{31:21}$	$\text{imm}_{20:12}$	0		U
imm_{20}	$\text{imm}_{20:12}$	$\text{imm}_{11:1}$	0	J

Figure 6.26 RISC-V immediates

Across instruction formats, RV32I attempts to keep immediate bits in the same instruction bits with the aim of simplifying hardware design (and at the cost of complicating instruction encodings). Figure 6.27 highlights this consistency by showing instruction fields for all formats. (The opcode is bits 6:0 for all instructions and is not shown.) instr_{31} always holds the sign bit of the immediate. $\text{instr}_{30:20}$ holds $\textbf{imm}_{30:20}$ for U-type instructions. Otherwise, $\text{instr}_{30:25}$ holds $\textbf{imm}_{10:5}$ $\text{instr}_{19:12}$ holds $\textbf{imm}_{19:12}$ for U/J-type instructions. $\textbf{imm}_{4:1}$ occupies either $\text{instr}_{24:21}$ or $\text{instr}_{11:8}$. Immediate bit 11 (when it is not the sign bit) and bit 0 are roving bits that are in bit 0 or 20 of the instruction.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	
11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd					I
11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0	S
12	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	11	B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd					U
20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd					J

Figure 6.27 RISC-V immediate encodings in machine instructions

Keeping immediate bit locations consistent across instruction formats is another example of regularity simplifying the design—specifically, it minimizes the number of wires and multiplexers needed to extract and sign-extend the immediate. Exercises 6.47 and 6.48 explore the hardware implications of this design decision further.

6.4.6 Addressing Modes

An *addressing mode* defines how an instruction specifies its operands. This section summarizes the modes used for addressing instruction operands. RISC-V uses four main modes: *register*, *immediate*, *base*, and *PC-relative* addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you learn other assembly languages. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC).

Register-Only Addressing

Register-only addressing uses registers for all source and destination operands. All R-type instructions use register-only addressing.

Immediate Addressing

Immediate addressing uses an immediate, along with registers, as operands. Some I-type instructions, such as add immediate (addi) and xori, use immediate addressing with a 12-bit signed immediate. Shift instructions with an immediate shift amount (slli, srli, and srai) are I-type instructions that encode the 5-bit unsigned immediate shift amount in **imm**$_{4:0}$. Load instructions (lb, lh, and lw) use the I-type instruction format but use base addressing, which is discussed next.

Base Addressing

Memory access instructions, such as load word (lw) and store word (sw), use *base addressing*. The effective address of the memory operand is calculated by adding the base address in register **rs1** to the sign-extended 12-bit offset found in the immediate field. Loads are I-type instructions and stores are S-type instructions.

PC-Relative Addressing

Branch and jump and link (jal) instructions use *PC-relative addressing* to specify the new value of the PC. The signed offset encoded in the immediate field is added to the PC to obtain the target address, the new PC; hence, the target address is said to be relative to the current PC. Branches and jal use a 13- and 21-bit signed immediate, respectively, The jump and link register (`jalr`) instruction uses base addressing, not PC-relative addressing. It can jump to any instruction address in the 32-bit address space because its target address is formed by adding **rs1** to the 12-bit signed immediate. The return address, PC+4, is written to **rd**.

The sequence of instructions below allows this program to jump to any address. Instruction addresses are listed to the left of each instruction. In this case, the program jumps to address 0x12345678 and writes 0x0100FE7C (i.e., PC+4) to `t1`.

# Address	RISC-V assembly
`0x0100FE74`	`lui s1, 0x12345`
`0x0100FE78`	`jalr t1, s1, 0x678`
`...`	`...`
`0x12345678`	`...`

for the offset. The most significant bits of the offset are encoded in the 12- and 20-bit immediate fields of the B- and J-type instructions. The offset's least significant bit is always 0, so it is not encoded in the instruction. The `auipc` (add upper immediate to PC) instruction also uses PC-relative addressing. For example, the instruction `auipc s3, 0xABCDE` places PC + 0xABCDE000 in `s3`.

6.4.7 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all formats share a 7-bit opcode field. Thus, the best place to begin is to look at the opcode to determine if it is an R-, I-, S/B-, or U/J-type instruction.

Example 6.6 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

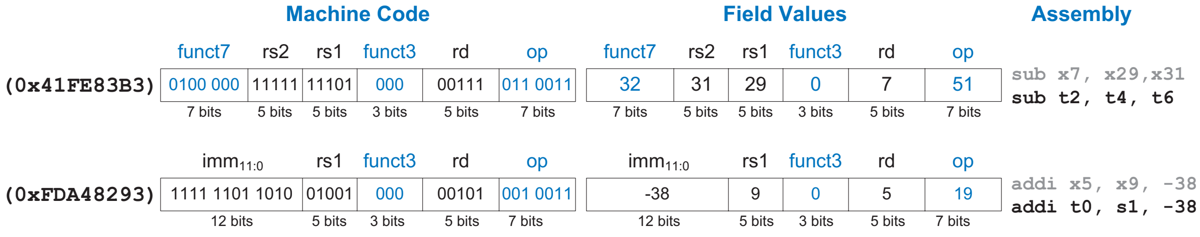
Translate the following machine language code into assembly language.

```
0x41FE83B3
0xFDA48293
```

Solution First, we represent each instruction in binary and look at the seven least significant bits to find the opcode for each instruction.

```
0100 0001 1111 1110 1000 0011 1011 0011  (0x41FE83B3)
1111 1101 1010 0100 1000 0010 1001 0011  (0xFDA48293)
```

The opcode determines how to interpret the rest of the bits. The first instruction's opcode is 0110011_2; so, according to Table B.1 in Appendix B, it is an R-type instruction and we can divide the rest of the bits into the R-type fields, as shown at the top of Figure 6.28. The second instruction's opcode is 0010011_2, which means it is an I-type instruction. We group the remaining bits into the I-type format, as seen in Figure 6.28, which shows the assembly code equivalent of the two machine instructions.



Machine Code

	funct7	rs2	rs1	funct3	rd	op
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

	$imm_{11:0}$	rs1	funct3	rd	op
(0xFDA48293)	1111 1101 1010	01001	000	00101	001 0011
	12 bits	5 bits	3 bits	5 bits	7 bits

Field Values

funct7	rs2	rs1	funct3	rd	op
32	31	29	0	7	51
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

$imm_{11:0}$	rs1	funct3	rd	op
-38	9	0	5	19
12 bits	5 bits	3 bits	5 bits	7 bits

Assembly

```
sub x7, x29,x31
sub t2, t4, t6
```

```
addi x5, x9, -38
addi t0, s1, -38
```

Figure 6.28 Machine code to assembly code translation

6.4.8 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing a new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory accesses and instruction executions. Figure 6.29 shows how machine instructions are stored in memory. In RISC-V programs, the instructions are normally stored starting at low addresses, but this may differ for each implementation. Figure 6.29 shows the code stored between addresses 0x00000830 and 0x0000083C. Remember that RISC-V memory is byte-addressable, so instruction addresses advance by 4, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in the 32-bit program counter (PC) register.

Ada Lovelace, 1815–1852
A British mathematician who wrote the first computer program, which calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the daughter of the poet Lord Byron.

Assembly code	Machine code
`add  s2, s3, s4`	`0x01498933`
`sub  t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw   t2, -6(s3)`	`0xFFA9A383`

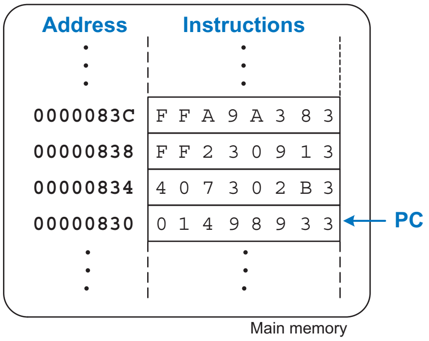


Figure 6.29 Stored program

To execute the code in Figure 6.29, the PC is initialized to address 0x00000830. The processor fetches the instruction at that memory address and executes the instruction, 0x01498933 (`add s2, s3, s4`). The processor then increments the PC by 4 to 0x00000834, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds everything necessary to determine what a program will do. For RISC-V, the architectural state includes the memory, register file, and PC. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

6.5 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING*

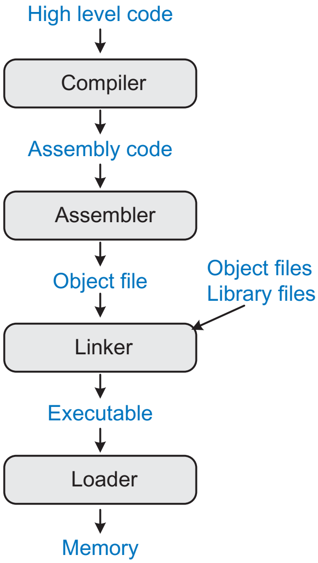


Figure 6.30 Steps for translating and starting a program

Until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution. We begin by introducing an example RISC-V *memory map*, which defines where code, data, and stack memory are located.

Figure 6.30 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

6.5.1 The Memory Map

With 32-bit addresses, the RISC-V address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFFC. Figure 6.31 shows an example memory map. Our memory map divides the address space into five parts or *segments*: the text, global data, and dynamic data segments, and segments for exception handlers and the operating system (OS), which includes memory dedicated to input/output (I/O). The following sections describe each segment. We present an

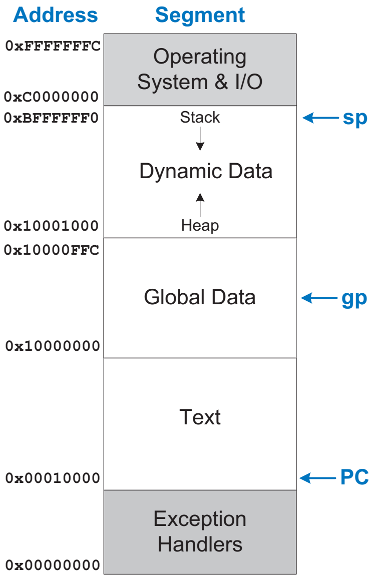


Figure 6.31 **Example RISC-V memory map**

example RISC-V memory map here; however, RISC-V does not define a specific memory map. While the exception handler is typically located at either low or high addresses, the user can define where the text (code and constant data), memory-mapped I/O, stack, and global data are placed. This allows for flexibility, especially with smaller systems, such as handheld devices, where only part of the memory range is used and, thus, populated with physical memory.

The Text Segment

The *text segment* stores the machine language user program. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Local variables are defined within a function and can only be accessed by that function; they are typically located in registers or on the stack. Global variables are allocated in memory before the program begins executing, and they are typically accessed using the *global pointer* register `gp` (register `x3`) that points to the middle of the global data segment. In this case, `gp` is 0x10000800. Using the 12-bit signed offset, programmers can use `gp` to access the entire global data segment.

RISC-V requires that sp maintain 16-byte alignment to enable compatibility with the quad-precision RISC-V base instruction set, RV128I, which operates on 128-bit (i.e., 16-byte) data. So, sp decrements by multiples of 16 to make room on the stack, even if smaller amounts of stack space are needed. We glossed over this requirement in Section 6.3.7 to highlight functionality above convention.

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the heap. The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program.

Upon start-up, the operating system sets up the stack pointer (sp, register x2) to point to the top of the stack, in this case 0xBFFFFFF0. The stack typically grows downward, as shown here. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers. As discussed in Section 6.3.7, functions also use the stack to save and restore registers. Each stack frame is accessed in last-in-first-out order.

The *heap* stores data that is allocated by the program during runtime. In C, memory allocations are made by the malloc function; in C++ and Java, new is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap typically grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator ensures that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

The Exception Handler, OS, and I/O Segments

The lowest part of the example RISC-V memory map is reserved for the exception handlers (see Section 6.6.2) and boot code that is run at start-up. The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see Section 9.2).

6.5.2 Assembler Directives

Assembler directives guide the assembler in allocating and initializing global variables, defining constants, and differentiating between code and data. Table 6.5 lists common RISC-V assembler directives, and Code Example 6.29 shows how to use them.

The .data, .text, .bss, and .section .rodata assembler directives tell the assembler to place the proceeding data or code in the global data, text (code), BSS, or read-only data (.rodata) segments of memory, respectively. The BSS segment is located in the global data segment but is initialized to zero. The read-only data segment is constant data that is placed in the text segment (i.e., in the *program memory*).

BSS stands for *block started symbol* and it was initially a keyword to allocate a block of uninitialized data. Now, most operating systems initialize data in the BSS segment to zero.

The program in Code Example 6.29 begins by making the main label global (.globl main) so that the main function can be called from outside this file, typically by the OS or bootloader. The value N is then set to 5 (.equ N, 5). The assembler replaces N with 5 before translating

Table 6.5 **RISC-V assembler directives**

Assembler Directive	Description
`.text`	Text section
`.data`	Global data section
`.bss`	Global data initialized to 0
`.section .foo`	Section named `.foo`
`.align N`	Align next data/instruction on 2^N-byte boundary
`.balign N`	Align next data/instruction on *N*-byte boundary
`.globl sym`	Label `sym` is global
`.string "str"`	Store string "`str`" in memory
`.word w1,w2,...,wN`	Store *N* 32-bit values in successive memory words
`.byte b1,b2,...,bN`	Store *N* 8-bit values in successive memory bytes
`.space N`	Reserve *N* bytes to store variable
`.equ name, constant`	Define symbol `name` with value `constant`
`.end`	End of assembly code

assembly instructions into machine code. For example, the instruction `lw t5,N*4(t0)` is translated into `lw t5,20(t0)` and then converted into machine code (0x0142AF03). Next, the program allocates the following global variables, as shown in Figure 6.32: `A` (a 7-element array of 32-byte values), `str1` (a null-terminated string), `B` and `C` (4 bytes each), and `D` (1 byte). `A`, `B`, and `str1` are initialized, respectively, to {5, 42, -88, 2, -5033, 720, 314}, 0x32A, and "RISC-V" (i.e., {52, 49, 53, 43, 2D, 56, 00} – see Table 6.2). Remember that, in the C programming language, strings are terminated with the null character (0x00). The variables `C` and `D` are uninitialized by the user and are located in the BSS segment. This assembler includes 16 bytes of unallocated memory between the data and BSS segments, as indicated by the gray boxes in Figure 6.32.

The `.align 2` assembler directive aligns the proceeding data or code on a $2^2 = 4$-byte boundary. The `.balign 4` (byte align 4) assembler directive is equivalent. These assembler directives help maintain alignment for data and instructions. For example, if the `.align 2` were removed before `B` is allocated (i.e., before `B: .word 0x32A`), `B` would have been allocated directly after the `str1` variable, in bytes 0x2157 – 0x215A (instead of 0x2158 – 0x215B).

The program from Code Example 6.29 was run on Western Digital's open-source commercial SweRV EH1 RISC-V core. Other processors use different memory maps, so would place variables and code at different addresses. The free RVfpga (RISC-V FPGA) course from Imagination Technologies shows how to use the SweRV EH1 core targeted to an FPGA to run C and assembly programs and to explore, expand, and modify that RISC-V procesor and system. See https://university.imgtec.com/rvfpga/.

Notice that `str2` is located in the code segment (*not* the data segment) at address 0x140, near the user code (`main`) which starts at address 0x88. By placing code and data together, the program minimizes the needed memory and the number of instructions to access data, which are both critical in handheld and embedded systems.

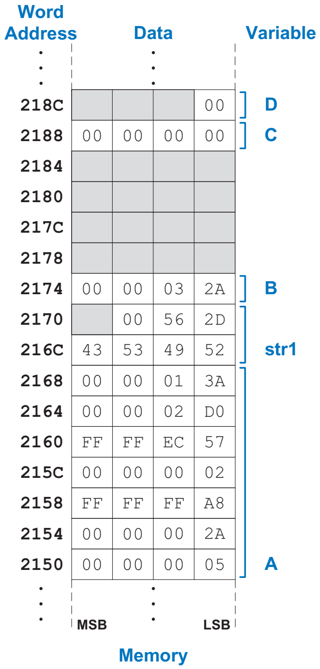


Figure 6.32 Memory allocation of global variables in Code Example 6.29

Code Example 6.29 USING ASSEMBLER DIRECTIVES

```
.globl main          # make the main label global
.equ N, 5            # N = 5

.data                # global data segment
A: .word 5, 42, −88, 2, −5033, 720, 314
str1: .string "RISC-V"
.align 2             # align next data on 2^2-byte boundary
B: .word 0x32A

.bss                 # bss segment – variables initialized to 0
C: .space 4
D: .space 1

.balign 4            # align next instruction on 4-byte boundary
.text                # text segment (code)
main:
  la  t0, A          # t0 = address of A                = 0x2150
  la  t1, str1       # t1 = address of str1             = 0x216C
  la  t2, B          # t2 = address of B                = 0x2174
  la  t3, C          # t3 = address of C                = 0x2188
  la  t4, D          # t4 = address of D                = 0x218C
  lw  t5, N*4(t0)    # t5 = A[N] = A[5] = 720           = 0x2D0
  lw  t6, 0(t2)      # t6 = B = 810                     = 0x32A
  add t5, t5, t6     # t5 = A[N] + C = 720 + 810 = 1530 = 0x5FA
  sw  t5, 0(t3)      # C  = 1530                        = 0x5FA
  lb  t5, N−1(t1)    # t5 = str1[N−1] = str1[4] = '–'   = 0x2D
  sb  t5, 0(t4)      # D  = str1[N−1]                   = 0x2D
  la  t5, str2       # t5 = address of str2             = 0x140
  lb  t6, 8(t5)      # t6 = str2[8] = 'r'               = 0x72
  sb  t6, 0(t1)      # str1[0] = 'r'                    = 0x72
  jr  ra             # return
.section .rodata
str2: .string "Hello world!"
.end                 # end of assembly file
```

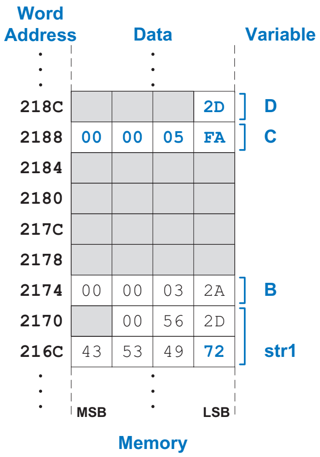


Figure 6.33 Final values of global variables `C`, `D`, and `str1`

The `main` function begins by loading the addresses of the global variables into `t0`–`t4` using the load address (`la`) pseudoinstruction (see Table B.7, located on the inside covers of the textbook). The program retrieves `A[5]` and `C` from memory, adds them together, and places the result (0x5FA) in `D`. Then it loads the value of `str1[4]` (which is '–' = ASCII code 0x2D) using instruction `lb t5, N−1(t1)` and places that value in global variable `B`. At the end, the program reads `str2[8]`, which is the character '`r`', and places that value in `str1[0]`. The `main` function finishes by returning to the operating system or boot code using `jr ra`. Figure 6.33 shows the final values of `C`, `D`, and `str1`. The `.end` assembly directive indicates the end of the assembly file.

6.5.3 Compiling

A compiler translates high-level code into assembly language, and an assembler then translates that assembly code into machine code. The examples in this section are based on GCC, a popular and widely used free compiler. GCC is part of a *toolchain* that includes other capabilities, some of which will be discussed in this section. Code Example 6.30 shows a simple high-level program with three global variables and two functions, along with the assembly code produced by GCC from SiFive's Freedom E SDK toolchain. See the Preface for instructions about using RISC-V compilers.

In Code Example 6.30, the `main` function starts by storing `ra` on the stack. It makes room for four words (16 bytes) but only uses one of the stack locations. Recall that `sp` must maintain 16-byte alignment for

Grace Hopper, 1906–1902
Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal. She has also documented the first computer "bug," which, in this case was an actual insect that interfered with a punchcard.

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y;

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b − 1));
}

void main() {
  f=2;
  g=3;
  y=func(f,g);

  return;
}
```

RISC-V Assembly Code

```
      .text
      .globl     func
      .type func, @function
func:
      addi  sp,sp,−16
      sw    ra,12(sp)
      sw    s0,8(sp)
      mv    s0,a0
      add   a0,a1,a0
      bge   a1,zero,.L5
.L1:
      lw    ra,12(sp)
      lw    s0,8(sp)
      addi  sp,sp,16
      jr    ra
.L5:
      addi  a1,a1,−1
      mv    a0,s0
      call  func
      add   a0,a0,s0
      j     .L1

      .globl        main
      .type main,   @function
main:
      addi  sp,sp,−16
      sw    ra,12(sp)
      lui   a5,%hi(f)
      li    a4,2
      sw    a4,%lo(f)(a5)
      lui   a5,%hi(g)
      li    a4,3
      sw    a4,%lo(g)(a5)
      li    a1,3
      li    a0,2
      call  func
      lui   a5,%hi(y)
      sw    a0,%lo(y)(a5)
      lw    ra,12(sp)
      addi  sp,sp,16
      jr    ra
      .comm y,4,4
      .comm g,4,4
      .comm f,4,4
```

compatibility with RV128I. main then writes the value 2 to global variable f and 3 to global variable g. The global variables are not yet placed in memory—they will be later, by the assembler. So, for now, the assembly code uses two instructions (lui followed by sw) instead of just one (sw) to store to each global variable in case it needs to specify a 32-bit address.

The program then puts f and g (i.e., 2 and 3) into the argument registers, a0 and a1, and calls func by using the pseudocode call func. The function (func) stores ra and s0 on the stack. It then places a0 (a) in s0 (because it will be needed after the recursive call to func) and calculates a0 = a0 + a1 (the return value = a + b). func then branches if a1 (b) is greater than or equal to zero. Otherwise, it restores ra, s0, and sp and returns using jr ra. If the branch is taken (b ≥ 0), func decrements a1 (b), and recursively calls func. After it returns from the recursive call, it adds the return value (a0) and s0 (a) and jumps to label .L1, where ra, s0, and sp are restored and the function returns. The main function then stores the returned result from func (a0) into global variable y, restores ra and sp, and returns y. At the bottom of the assembly code, the program indicates that it has three 4-byte-wide global variables f, g, and y, using .comm g, 4, 4, etc. The first 4 indicates 4-byte alignment and the second 4 indicates the size of the variable (4 bytes).

To compile, assemble, and link a C program named prog.c with GCC, use the command:

```
gcc -O1 -g prog.c -o prog
```

This command produces an executable output file called prog. The -O1 flag asks the compiler to perform basic optimizations rather than producing grossly inefficient code. The -g flag tells the compiler to include debugging information in the file.

To see the intermediate steps, we can use GCC's -S flag to compile but not assemble or link.

```
gcc -O1 -S prog.c -o prog.s
```

The output, prog.s, is rather verbose, but the interesting parts are shown in Code Example 6.30.

6.5.4 Assembling

An *assembler* turns the assembly language code into an *object file* containing machine language code. GCC can create the object file from either prog.s or directly from prog.c using:

```
gcc -c prog.s -o prog.o
```

or

```
gcc -O1 -g -c prog.c -o prog.o
```

The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all of the symbols, such as labels and global variable names. The names and addresses of the symbols are kept in a symbol table. On the second pass through the code, the assembler produces the machine language code. Addresses for labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

We can *disassemble* the object file using the `objdump` command to see the assembly language code beside the machine language code.

```
objdump -S prog.o
```

The following shows the disassembly of the .text section. If the code was originally compiled with `-g`, the disassembler also shows the corresponding lines of C code, as shown below, interspersed with the assembly code. Notice that the `call` pseudoinstruction was translated into two RISC-V instructions: `auipc ra,0x0` and `jalr ra` in case the function is far away, that is, farther away from the current PC than `jal`'s signed 21-bit offset can reach. The instructions for storing to the global variables are also just placeholders until the global variables are placed in memory. For example, the three instructions at addresses 0x48 to 0x50 are for storing the value 2 in global variable `f`. Once `f` is placed in memory in the linking stage, the instructions will be updated.

```
00000000 <func>:
int f, g, y;
int func(int a, int b) {
   0: ff010113            addi  sp,sp,-16
   4: 00112623            sw    ra,12(sp)
   8: 00812423            sw    s0,8(sp)
   c: 00050413            mv    s0,a0
   if (b<0) return (a+b);
  10: 00a58533            add   a0,a1,a0
  14: 0005da63            bgez  a1,28 <.L5>
00000018 <.L1>:
  else return(a + func(a, b-1));
}
  18: 00c12083            lw    ra,12(sp)
  1c: 00812403            lw    s0,8(sp)
  20: 01010113            addi  sp,sp,16
  24: 00008067            ret
00000028 <.L5>:
  else return(a + func(a, b-1));
  28: fff58593            addi  a1,a1,-1
  2c: 00040513            mv    a0,s0
  30: 00000097                    auipc ra,0x0
  34: 000080e7                    jalr  ra # 30 <.LVL5+0x4>
  38: 00850533                    add   a0,a0,s0
  3c: fddff06f                    j     18 <.L1>

00000040 <main>:
void main() {
  40: ff010113                    addi  sp,sp,-16
  44: 00112623                    sw    ra,12(sp)
  f=2;
  48: 000007b7                    lui   a5,0x0
  4c: 00200713                    li    a4,2
  50: 00e7a023                    sw    a4,0(a5) # 0 <func>
  g=3;
  54: 000007b7                    lui   a5,0x0
  58: 00300713                    li    a4,3
  5c: 00e7a023                    sw    a4,0(a5) # 0 <func>
  y=func(f,g);
  60: 00300593                    li    a1,3
  64: 00200513                    li    a0,2
  68: 00000097                    auipc ra,0x0
  6c: 000080e7                    jalr  ra # 68 <main+0x28>
  70: 000007b7                    lui   a5,0x0
  74: 00a7a023                    sw    a0,0(a5) # 0 <func>
  return;
}
  78: 00c12083                    lw    ra,12(sp)
  7c: 01010113                    addi  sp,sp,16
  80: 00008067                    ret
```

We can view the symbol table from the object file using objdump with the -t flag. The interesting parts are shown below. We added labels for the three columns of interest: the symbol's memory address, size, and name. Because the program has not yet been placed in memory (it has not been linked), the addresses are only placeholders for now. The .text indicates the code (text) segment and .data the data (global data) segment. The size of those two symbols is currently 0 because the program has not yet been linked. The size of the two functions, func and main, are listed: func is 0x40 (64) bytes = 16 instructions, and main is 0x44 (68) bytes = 17 instructions, as shown in the code above. The global variable symbols f, g, and y are listed and are 4 bytes each, but their addresses are listed as a placeholder value, 0x00000004, because they have not yet been assigned addresses.

```
objdump -t prog.o
```

```
SYMBOL TABLE:
  Address                 Size        Symbol Name
  00000000 l d .text      00000000    .text
  00000000 l d .data      00000000    .data
  00000000 g F .text      00000040    func
  00000040 g F .text      00000044    main
  00000004   O *COM*      00000004    f
  00000004   O *COM*      00000004    g
  00000004   O *COM*      00000004    y
```

We will largely ignore the unlabeled columns in this symbol table. They show the flags associated with the symbols (`l` for local or `g` for global, and `d` for debug, `F` for function, or `O` for object) and the section in which the symbol is located (`.text`, `.data`, or `*COM*` (common) when it is not located in a section).

6.5.5 Linking

Most large programs contain more than one file. If the programmer edits only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated. Also, a program typically involves some start-up code (to initialize the stack, heap, and so forth) that must be executed before calling the `main` function.

The job of the linker is to combine all of the object files and the start-up code into one machine language file called the *executable* and assign addresses for global variables. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the code based on the new label and global variable addresses. Invoke GCC to link the object file using:

```
gcc prog.o –o prog
```

We can again disassemble the executable using:

```
objdump –S –t prog
```

The start-up code is too lengthy to show, but the updated symbol table and program code disassembled from the executable are shown below. We have again added labels to the columns of interest. The functions and global variables are now relocated to actual addresses. According to the symbol table, the overall text and data segments (which include start-up code and system data) begin at 0x10074 and 0x115e0, respectively. `func` starts at address 0x10144 and is 0x3c bytes (15 instructions). `main` starts at 0x10180 and is 0x34 bytes (13 instructions). The global variables are each 4 bytes; `f` is located at memory address 0x11a30, `g` at 0x11a34, and `y` at 0x11a38.

Notice that now that the global variables, `f`, `g`, and `y`, are allocated memory addresses, they are listed as global symbols (as indicated by the `g` flag) and are located in the `.bss` segment, where uninitialized global variables are placed.

```
SYMBOL TABLE:
Address                 Size       Symbol Name
00010074 l d .text      00000000   .text
000115e0 l d .data      00000000   .data
00010144 g F .text      0000003c   func
00010180 g F .text      00000034   main
00011a30 g O .bss       00000004   f
00011a34 g O .bss       00000004   g
00011a38 g O .bss       00000004   y
```

Notice that the size of `func`, as shown below, is now only 15 instructions instead of 16. The call to `func` was nearby; so, only one instruction, `jalr`, was needed to make the call. Likewise, the `main` code reduced from 17 to 13 instructions because of near calls and stores near to the global pointer, `gp`. The program stores to `f` using a single instruction: `sw  a4,−944(gp)`. From this instruction, we can also determine the value of the global pointer, `gp`, that was initialized by the start-up code. We know that `f` is at address 0x11a30; so, `gp` is 0x11a30 + 944 = 0x11DE0.

```
00010144 <func>:
int f, g, y;

int func(int a, int b) {
   10144: ff010113          addi  sp,sp,-16
   10148: 00112623          sw    ra,12(sp)
   1014c: 00812423          sw    s0,8(sp)
   10150: 00050413          mv    s0,a0
  if (b<0) return (a+b);
   10154: 00a58533          add   a0,a1,a0
   10158: 0005da63          bgez  a1,1016c <func+0x28>
  else return(a + func(a, b-1));
}
   1015c: 00c12083          lw    ra,12(sp)
   10160: 00812403          lw    s0,8(sp)
   10164: 01010113          addi  sp,sp,16
   10168: 00008067          ret
  else return(a + func(a, b-1));
   1016c: fff58593          addi  a1,a1,-1
   10170: 00040513          mv    a0,s0
   10174: fd1ff0ef          jal   ra,10144 <func>
   10178: 00850533          add   a0,a0,s0
   1017c: fe1ff06f          j     1015c <func+0x18>

00010180 <main>:
```

```
void main() {
   10180: ff010113            addi  sp,sp,-16
   10184: 00112623            sw    ra,12(sp)
  f=2;
   10188: 00200713            li    a4,2
   1018c: c4e1a823            sw    a4,-944(gp) # 11a30 <f>
  g=3;
   10190: 00300713            li    a4,3
   10194: c4e1aa23            sw    a4,-940(gp) # 11a34 <g>
  y=func(f,g);
   10198: 00300593            li    a1,3
   1019c: 00200513            li    a0,2
   101a0: fa5ff0ef            jal   ra,10144 <func>
   101a4: c4a1ac23            sw    a0,-936(gp) # 11a38 <y>

  return;
}
   101a8: 00c12083            lw    ra,12(sp)
   101ac: 01010113            addi  sp,sp,16
   101b0: 00008067            ret
```

6.5.6 Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk or flash storage) into the text segment of memory. The operating system jumps to the beginning of the program to begin executing. Figure 6.34 shows the memory map at the beginning of program execution.

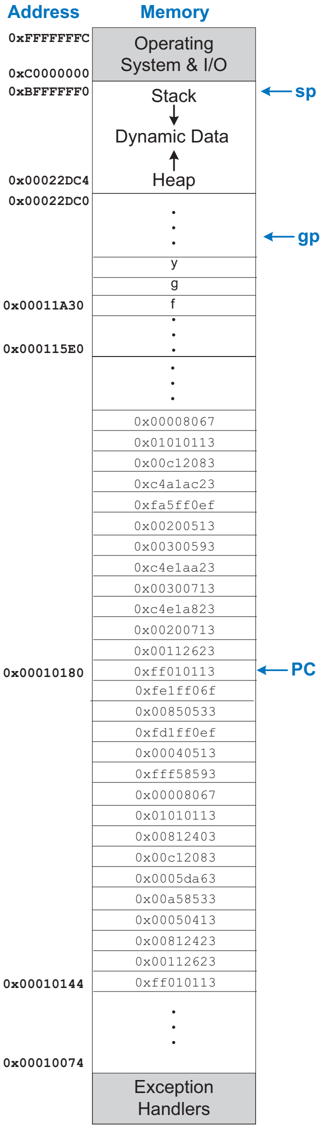


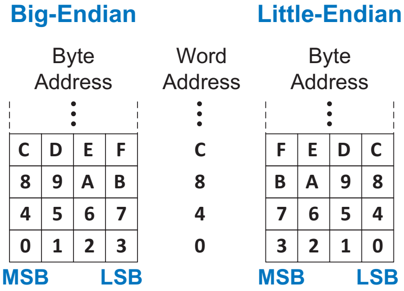
Figure 6.34 prog **loaded into memory**

6.6 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include endianness, exceptions, signed and unsigned arithmetic instructions, floating-point instructions, and compressed (16-bit) instructions.

6.6.1 Endianness

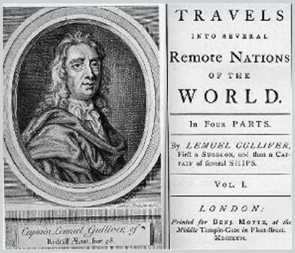
Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.35. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ (see Figure 6.35). In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.



Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0

Figure 6.35 **Big- and little-endian memory addressing**

RISC-V is typically little-endian, although a big-endian variant has been defined. IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big- and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.


TRAVELS
INTO SEVERAL
Remote NATIONS
OF THE
WORLD.
In Four PARTS.
By LEMUEL GULLIVER,
First a SURGEON, and then a CAPTAIN of several SHIPS.
Vol. I.
LONDON:
Printed for BENJ. MOTTE, at the Middle Temple-Gate in Fleet-street.
MDCCXXVI.

Captain Lemuel Gulliver, of Redriff Ætat. suæ 58.

The terms *big-endian* and *little-endian* come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories, the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

These terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools' Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, Leeds University Library.)

6.6.2 Exceptions

An *exception* is like an unscheduled function call caused by an event in hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, and then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition caused by the software, such as an undefined instruction. Software exceptions are sometimes called *traps*. Other causes of exceptions include reset and attempts to read nonexistent memory. Like any other function call, an exception must save the return address, jump to some address, do its work, clean up after itself, and return to the program where it left off.

Execution Modes and Privilege Levels

A RISC-V processor can operate in one of several execution modes with different privilege levels. Privilege levels dictate what instructions can be executed and what memory can be accessed. The three main RISC-V privilege levels are *user mode*, *supervisor mode*, and *machine mode*, in order of increasing privilege. Machine mode (M-mode) is the highest privilege level; a program running in this mode can access all registers and memory locations. M-mode is the only required privilege mode and the only mode used

A fourth privilege level exists called *hypervisor mode* (*H-mode*) that supports *machine virtualization*, that is, the appearance of multiple machines (potentially with multiple operating systems) running on a single physical machine. H-mode has higher privilege than S-mode but not as much privilege as M-mode.

in processors without an operating system (OS), including many embedded systems. User applications that run on top of an OS typically run in user mode (U-mode) and the OS runs in supervisor mode (S-mode); user programs do not have access to privileged registers or memory locations reserved for the OS. The different modes keep the key state from being corrupted. We discuss exceptions when running in M-mode. Exceptions that occur at other levels are similar but use registers associated with that mode.

Exception Handlers

Exception handlers use four special-purpose registers, called *control and status registers* (*CSRs*), to handle an exception: `mtvec`, `mcause`, `mepc`, and `mscratch`. The machine trap-vector base-address register, `mtvec`, holds the address of the exception handler code. When an exception occurs, the processor records the cause of an exception in `mcause` (see Table 6.6), stores the PC of the excepting instruction in `mepc`, the machine exception PC register, and jumps to the exception handler at the address preconfigured in `mtvec`.

After jumping to the address in `mtvec`, the exception handler reads the `mcause` register to examine what caused the exception and responds appropriately (e.g., by reading the keyboard on a hardware interrupt).

RISC-V defines a whole slew of CSRs, all of which must be initialized at start-up.

The value of `mcause` can be classified as either an interrupt or an exception, as indicated by the left-most column in Table 6.6, which is bit 31 of `mcause`. Bits [30:0] of `mcause` hold the *exception code*, that indicates the cause of the interrupt or exception.

Table 6.6 Common exception cause encodings

Interrupt	Exception Code	Description
1	3	Machine software interrupt
1	7	Machine timer interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-Mode
0	9	Environment call from S-Mode
0	11	Environment call from M-Mode

Exceptions can use one of two exception handling modes: direct or vectored. RISC-V typically uses the direct mode described here, where all exceptions branch to same address, that is, the base address encoded in bits 31:2 of `mtvec`. In vectored mode, exceptions branch to an *offset* from the base address, depending on the cause of the exception. Each offset is separated by a small number of addresses—for example, 32 bytes—so the exception handler code may need to jump to a larger exception handler to deal with the exception. The exception mode is encoded in bits 1:0 of `mtvec`; 00_2 is for direct mode and 01_2 for vectored.

Exception-related registers are specific to the operating mode. M-mode registers are mtvec, mepc, mcause, and mscratch, and S-mode registers are sepc, scause, and sscratch. H-mode also has its own registers. Separate exception registers dedicated to each mode provide hardware support for multiple privilege levels.

It then either aborts the program or returns to the program by executing the mret, machine exception return instruction, that jumps to the address in mepc. Holding the PC of the excepting instruction in mepc is analogous to using ra to store the return address during a jal instruction. Exception handlers must use program registers (x1−x31) to handle exceptions, so they use the memory pointed to by mscratch to store and restore these registers.

Exception-Related Instructions

Exception handlers use special instructions to deal with exceptions. These instructions are called *privileged instructions* because they access CSRs. They are part of the base RV32I instruction set (see Appendix B, Table B.8). The mepc and mcause registers are not part of the RISC-V program registers (x1−x31), so the exception handler must move these special-purpose (CSR) registers into the program registers to read and operate on them. RISC-V uses three instructions to read, write, or both read and write CSRs: csrr (read CSR), csrw (write CSR), and csrrw (read/write CSR). For example, csrr t1, mcause reads the value in mcause into t1; csrw mepc, t2 writes the value in t2 into mepc; and csrrw t1, mscratch, t0 simultaneously reads the value in mscratch into t1 and writes the value in t0 into mscratch.

csrrw is an actual RISC-V instruction (see Table B.8 in Appendix B), but csrr and csrw are pseudoinstructions. csrr is implemented as csrrs rd, CSR, x0 and csrw as csrrw x0, CSR, rs1.

Exception Handling Summary

In summary, when a processor detects an exception, it:

1. Jumps to the exception handler address held in mtvec.
2. The exception handler saves registers on a small stack pointed to by mscratch and then uses csrr (read CSR) to look at the cause of the exception (encoded in mcause) and respond accordingly.
3. When the handler is finished, it optionally increments mepc by 4, restores registers from memory and either aborts the program or returns to the user code using the mret instruction, which jumps to the address held in mepc.

Example 6.7 EXCEPTION HANDLER CODE

Write an exception handler for dealing with the following two exceptions: illegal instruction (mcause = 2) and load address misaligned (mcause = 4). If an illegal instruction occurs, the program should simply continue executing after the illegal instruction. Upon a load address misaligned exception, the program should abort. If any other exception occurs, the program should attempt to re-execute the instruction.

Solution The exception handler begins by preserving program registers that will be overwritten. It then checks for each exception cause and (1) continues executing just past the excepting instruction (i.e., at mepc + 4) upon an illegal instruction exception, (2) aborts upon a misaligned load address, or (3) attempts to re-execute the excepting instruction (i.e., returns to mepc) upon any other exception. Before returning to the program, the handler restores any registers that were overwritten. To abort the program, the handler jumps to exit code located at the exit label (not shown). For programs running on top of an OS, the j exit instruction may be replaced by an environment call (ecall) with the return code stored in a program register such as a0.

```
# save registers that will be overwritten
  csrrw t0, mscratch, t0    # swap t0 and mscratch
  sw    t1, 0(t0)           # save t1 on mscratch stack
  sw    t2, 4(t0)           # save t2 on mscratch stack

# check cause of exception
  csrr  t1, mcause          # t1 = mcause
  addi  t2, x0, 2           # t2 = 2 (illegal instruction exception code)

illegalinstr:
  bne   t1, t2, checkother  # branch if not an illegal instruction
  csrr  t2, mepc            # t2 = exception PC
  addi  t2, t2, 4           # increment exception PC by 4
  csrw  mepc, t2            # mepc = mepc + 4
  j     done                # restore registers and return

checkother:
  addi  t2, x0, 4           # t2 = 4 (misaligned load exception code)
  bne   t1, t2, done        # branch if not a misaligned load
  j     exit                # exit program
# restore registers and return from the exception
done:
  lw    t1, 0(t0)           # restore t1 from mscratch stack
  lw    t2, 4(t0)           # restore t2 from mscratch stack
  csrrw t0, mscratch, t0    # swap t0 and mscratch
  mret                      # return to program (PC = mepc)
  ...
exit:
  ...
```

At start-up, the processor jumps to the *reset exception vector*, a hardwired memory address—for example, 0x200—which is the starting address of the *boot loader* code, also called *boot code*. Although reset is not a typical exception that occurs during program execution, it is called such because reset is an exceptional state of the processor. The boot code configures the memory system, initializes the CSRs and stack pointer, and reads part of the OS from disk. Then, it begins a much longer boot process in the OS. The OS eventually will load a program, change to unprivileged user mode, and jump to the start of the program. In *bare metal* systems—that is, systems with no OS—user code (potentially with a lightweight boot code for setting up the stack pointer, etc.) is typically placed directly at the reset vector.

A particularly important exception is a *system call*, also called an *environment call*. Programs use these to call a function in the OS, which runs at a higher privilege level than user code. This exception is initiated by the user program executing the ecall instruction. Like a function call, the program may set up argument registers before making the system call.

Unlike other architectures, such as MIPS and ARM, RISC-V does not include instructions (or exceptions) for detecting overflow because it can be detected using a series of existing instructions. For example, the following code detects unsigned overflow when adding t1 and t2.

```
add  t0, t1, t2
bltu t0, t1, overflow
```

In other words, if the result (t0) is less than either of the operands (in this case, t1), overflow occurred.

The following code detects overflow when adding two signed numbers, t1 and t2:

```
add  t0, t1, t2
slti t3, t2, 0
slt  t4, t0, t1
bne  t3, t4, overflow
```

In equation form, overflow = $(\texttt{t2} < 0) \,\&\, (\texttt{t0} \geq \texttt{t1}) \mid (\texttt{t2} \geq 0) \,\&\, (\texttt{t0} < \texttt{t1})$

In words, overflow occurs when one operand is negative ($\texttt{t3} = 1$) and the result is not less than the other operand ($\texttt{t4} = 0$), or when one operand is greater than or equal to 0 ($\texttt{t3} = 0$), and the result is less than the other operand ($\texttt{t4} = 1$).

6.6.3 Signed and Unsigned Instructions

Recall that a binary number may be signed or unsigned. As with most architectures, RISC-V uses two's complement representation for signed numbers. RISC-V has certain instructions that come in signed and unsigned flavors, including multiplication, division, set less than, branches, and partial word loads.

Multiplication and Division

Multiplication and division behave differently for signed and unsigned numbers. For example, as an unsigned number, 0xFFFFFFFF represents a large number, but as a signed number it represents –1. Hence, 0xFFFFFFFF × 0xFFFFFFFF would equal 0xFFFFFFFE00000001 if the numbers were unsigned but 0x0000000000000001 if the numbers were signed. (Notice that the lower 32 bits are the same for both signed and unsigned multiplication.) Therefore, multiplication and division come in both signed and unsigned flavors. mulh and div treat the operands as signed numbers. multhu and divu treat the operands as unsigned numbers. mulhsu treats the first operand as signed and the second operand as unsigned. All multiply high instructions (mulh, mulhu, and mulhsu) put the most significant 32 bits in the destination register **rd**. The lower 32 bits of the result are the same for unsigned or signed multiplication, so mul puts the lower 32 bits of the multiplication result in **rd** for both unsigned and signed multiplication.

Set Less Than

Set less than instructions compare either two registers (slt) or a register and an immediate (slti). Set less than also comes in signed (slt and slti) and unsigned (sltu and sltiu) versions. In a signed comparison, 0x80000000 is less than any other number, because it is the most negative two's complement number. In an unsigned comparison, 0x80000000 is greater than 0x7FFFFFFF but less than 0x80000001, because all numbers are positive. Beware that sltiu sign-extends the 12-bit immediate before treating it as an unsigned number. For example, sltiu s0, s1, −1273 compares s1 to 0xFFFFFB07, treating the immediate as a large positive number.

Branches

The *branch if less than* and *branch if greater than or equal to* instructions also come in signed (blt and bge) and unsigned (bltu and bgeu) versions. The signed versions treat the two source operands as two's complement numbers and the unsigned versions treat the source operands as unsigned numbers.

Loads

As described in Section 6.3.6, byte loads come in signed (lb) and unsigned (lbu) versions. lb sign-extends the byte, and lbu zero-extends the byte to fill the entire 32-bit register. Similarly, signed and unsigned half-word loads (lh and lhu) load two bytes into the lower half and sign- or zero-extend the upper half of the word.

6.6.4 Floating-Point Instructions

The RISC-V architecture defines optional floating-point extensions called RVF, RVD, and RVQ for operating on single-, double-, and quad-precision floating-point numbers, respectively. RVF/D/Q define 32 floating-point registers, f0 to f31, with a width of 32, 64, or 128 bits, respectively. When a processor implements multiple floating-point extensions, it uses the lower part of the floating-point register for lower-precision instructions. f0 to f31 are separate from the program (also called *integer*) registers, x0 to x31. As with program registers, floating-point registers are reserved for certain purposes by convention, as given in Table 6.7.

Table B.3 in Appendix B lists all of the floating-point instructions. Computation and comparison instructions use the same mnemonics for all precisions, with .s, .d, or .q appended at the end to indicate precision. For example, fadd.s, fadd.d, and fadd.q perform single-, double-, and quad-precision addition, respectively. Other floating-point instructions include fsub, fmul, fdiv, fsqrt, fmadd (multiply-add), and fmin. Memory accesses use separate instructions for each precision. Loads are flw, fld, and flq, and stores are fsw, fsd, and fsq.

Floating-point instructions use R-, I-, and S-type formats, as well as a new format, the R4-type instruction format (see Figure B.1 in Appendix B).

Table 6.7 RISC-V floating-point register set

Name	Register Number	Use
ft0–7	f0–7	Temporary variables
fs0–1	f8–9	Saved variables
fa0–1	f10–11	Function arguments/Return values
fa2–7	f12–17	Function arguments
fs2–11	f18–27	Saved variables
ft8–11	f28–31	Temporary variables

Code Example 6.31 USING A FOR LOOP TO ACCESS AN ARRAY OF FLOATS

High-Level Code

```
int i;
float scores[200];

for (i = 0; i < 200; i = i + 1)



 scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i


  addi     s1, zero, 0       # i = 0
  addi     t2, zero, 200     # t2 = 200
  addi     t3, zero, 10      # t3 = 10
  fcvt.s.w ft0, t3           # ft0 = 10.0


for:
  bge      s1, t2, done      # if i >= 200 then done
  slli     t3, s1, 2         # t3 = i * 4
  add      t3, t3, s0        # address of scores[i]
  flw      ft1, 0(t3)        # ft1 = scores[i]
  fadd.s   ft1, ft1, ft0     # ft1 = scores[i] + 10
  fsw      ft1, 0(t3)        # scores[i] = t1
  addi     s1, s1, 1         # i = i + 1
  j        for               # repeat
done:
```

This format is needed for multiply-add instructions, which use four register operands. Code Example 6.31 modifies Code Example 6.21 to operate on an array of single-precision floating-point scores. The changes are in bold.

6.6.5 Compressed Instructions

RISC-V's compressed instruction extension (RVC) reduces the size of common integer and floating-point instructions to 16 bits by reducing the sizes of the control, immediate, and register fields and by taking advantage of redundant or implied registers. This reduced instruction size decreases cost, power, and required memory—all of which can be crucial for handheld and mobile applications. According to the *RISC-V Instruction Set Manual*, typically 50% to 60% of a program's instructions can be replaced with RVC instructions. 16-bit instructions still operate on the base data size (32, 64, or 128 bits), as determined by the base instruction set. Assembly programs may use a mix of compressed and 32-bit instructions if the processor can handle both.

Most RV32I instructions have a compressed counterpart starting with `c.`, as listed in Table B.6 of Appendix B. To reduce their size, most compressed instructions only specify two registers; the first source is also the destination. Most use 3-bit register codes to specify one of 8 registers from `x8` to `x15`. `x8` is encoded as 000_2, `x9` as 001_2, and so on. Immediates are also shorter (6–11 bits), and fewer bits are available for opcodes. Figure B.2, located on the inside back cover of this book, shows the compressed instruction formats.

Code Example 6.32 modifies Code Example 6.21 to use compressed instructions. Notice that the constant 200 is too large to fit in a compressed immediate, so s0 is initialized with an uncompressed addi. There is no compressed c.bge, so we also use a noncompressed bge. We also increment s0 as a pointer to scores[i] because shifting and adding is cumbersome with 2-operand compressed instructions. The program is shrunk from 40 to 22 bytes.

Many RISC-V assemblers generate code using a mix of compressed and uncompressed instructions, using compressed instructions wherever possible to minimize code size.

Code Example 6.32 USING COMPRESSED INSTRUCTIONS

High-Level Code

```
int i;
int scores[200];

for (i = 0; i < 200; i = i + 1)



  scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i

  c.li s1, 0           # i = 0
  addi t2, zero, 200   # t2 = 200

for:
  bge    s1, t2, done  # if i >= 200 then done
  c.lw   a3, 0(s0)     # a3 = scores[i]
  c.addi a3, 10        # a3 = scores[i] + 10
  c.sw   a3, 0(s0)     # scores[i] = a3
  c.addi s0, 4         # next element of
                         scores
  c.addi s1, 1         # i = i + 1
  c.j    for           # repeat
done:
```

6.7 EVOLUTION OF THE RISC-V ARCHITECTURE

RISC-V was designed to be a commercially viable, open-source computer architecture, that is robust, efficient, and flexible. RISC-V differentiates itself from other architectures because it is open source, uses base instruction sets to ease compatibility, supports the full range of microarchitectures, from embedded systems to high-performance computers, offers both defined and customizable extensions, and provides features, such as compressed instructions and RV128I, that optimize hardware and support both existing and future designs, ensuring the architecture's longevity.

RISC-V has also created a community of both industry and academic partners by forming RISC-V International (see riscv.org), thereby accelerating innovation, commercialization, and collaboration. This consortium of developers also helps design and ratify the RISC-V architecture. RISC-V International has grown to have more than 500 industrial and academic members as of 2021, including Western Digital, NVIDIA, Microchip, and Samsung.

The RISC-V architecture is described in *The RISC-V Instruction Set Manual* (riscv.org/specifications). Early versions of the manual, through version 2.2, are a gem of a specification, succinct, readable, and interspersed with the rationale behind the design decisions embodied in the architecture.

6.7.1 RISC-V Base Instruction Sets and Extensions

RISC-V includes various base instruction sets and extensions so that it can support a broad range of processors—from small, inexpensive embedded processors, such as those in handheld devices, to high-performance, multicore, multithreaded systems. RISC-V has 32-, 64-, and 128-bit base instruction sets: RV32I/E, RV64I, and RV128I. The 32-bit base instruction set comes in the standard version RV32I discussed in this chapter and in an embedded version RV32E with only 16 registers, intended for very low-cost processors. As of 2021, only the RV32I and RV64I instruction sets are frozen; RV32E and RV128I are still being defined. Along with these base architectures, RISC-V also defines the extensions listed in Table 6.8. The most commonly used extensions—those for floating-point operations (RVF/D/Q), compressed instructions (RVC), and atomic instructions (RVA)—are fully specified and frozen to enable development and commercialization. The remaining extensions are still being developed.

All RISC-V processors must support one of the base architectures—RV32/64/128I or RV32E—and may optionally support extensions, such as the compressed or floating-point extensions. By using extensions, instead of new architecture versions, RISC-V alleviates the burden of backward or forward compatibility between microarchitectures. All

Table 6.8 RISC-V extensions

Extension	Description	Status
M	Integer multiplication and division	Frozen
F	Single-precision floating-point	Frozen
D	Double-precision floating-point	Frozen
Q	Quad-precision floating-point	Frozen
C	Compressed instructions	Frozen
A	Atomic instructions	Frozen
B	Bit manipulations	Open
L	Decimal floating-point	Open
J	Dynamically translated languages	Open
T	Transactional memory	Open
P	Packed-SIMD instructions	Open
V	Vector operations	Open

processors must support at least the base architecture. However, a processor need not support all (or even any) of the extensions.

To understand the evolution of the RISC-V architecture, it is important to understand other architectures that preceded RISC-V and, notably, the MIPS architecture. RISC-V follows many principles from the MIPS architecture but also benefits from the perspective of modern architectures and applications to include features that support embedded, multicore, and multithreaded systems and extensibility, among other features. The next section compares the RISC-V and MIPS architectures.

6.7.2 Comparison of RISC-V and MIPS Architectures

The RISC-V architecture has many similarities to the MIPS architecture developed by John Hennessy in the 1980's, but it eliminates some unnecessary complexity—and introduces some, in the case of immediates! Similarities include assembly and machine code formats, instruction mnemonics, register naming, and stack and calling conventions. Differences include RISC-V's immediate sizes and encodings, branching relative to PC (instead of PC+4), both branches and jumps being PC-relative, removal of the branch delay slot from MIPS, a strict definition of source and destination register instruction fields, different numbers of temporary, saved, and argument registers, and more extensibility by including more control bits in the instruction. By keeping **rs1**, **rs2**, and **rd** in the same bitfields of every instruction type that uses them, RISC-V simplifies the decoder hardware relative to MIPS. Similarly, RISC-V's immediate encodings simplify the immediate extension hardware.

6.7.3 Comparison of RISC-V and ARM Architectures

ARM is a RISC architecture that was developed around the same time as the MIPS architecture in the 1980's. Over the past decade or more, ARM processors have dominated the mobile devices arena and are also found in other applications, such as robots, pinball machines, and servers. ARM's similarities to RISC-V include its small number of machine code formats and assembly instructions and similar stack and calling conventions. ARM differs from RISC-V by including conditional execution, complex indexing modes for accessing memory, its ability to push and pop multiple registers onto/off the stack using a single instruction, optionally shifted source registers, and unconventional immediate encodings. Immediates are encoded as an 8-bit value and a 4-bit rotation, and they encode only positive immediates (subtraction is determined by the control bits). These features—particularly, conditional execution, shifted registers, and indexing modes—are typically only found in CISC architectures, but ARM includes them to reduce program and, thus, memory size, which is critical for embedded and handheld devices. However, these design decisions also result in more complex hardware.

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

Almost all personal computers today use x86 architecture microprocessors. x86, also called IA-32, is a 32-bit architecture originally developed by Intel. AMD also sells x86-compatible microprocessors.

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than RISC-V. However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

x86 is an example of a complex instruction set computer (CISC) architecture. In contrast to RISC architectures such as RISC-V, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact to save memory when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer but rather to illustrate some of the similarities and differences between x86 and RISC-V. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and RISC-V (RV32I) are summarized in Table 6.9.

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to

Table 6.9 Major differences between RISC-V (RV32I) and x86

Feature	RISC-V	x86
# of registers	32 general-purpose	8, some restrictions on purpose
# of operands	3 (2 sources, 1 destination)	2 (1 source, 1 source/destination)
Operand locations	Registers or immediates	Registers, immediates, or memory
Operand size	32 bits	8, 16, or 32 bits
Condition flags	No	Yes
Instruction types	Simple	Simple and complicated
Instruction encoding	Fixed: 4 bytes	Variable: 1–15 bytes

32 bits. These registers are called EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.36

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like `sp` in RISC-V, ESP is normally reserved for the stack pointer.

The x86 program counter is called the EIP (the *extended instruction pointer*). Like the RISC-V PC, it advances from one instruction to the next or can be changed with branch and function call instructions.

6.8.2 x86 Operands

RISC-V instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, x86 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

RISC-V instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. Table 6.10 lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Like RISC-V (RV32I), x86 has a 32-bit memory space that is byte-addressable. However, unlike RISC-V, x86 supports a wider variety of memory indexing modes. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.11 illustrates these combinations. The displacement can be an

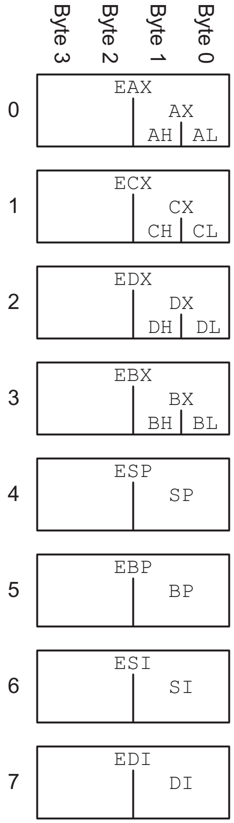


Figure 6.36 x86 registers

Table 6.10 Operand locations

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <– EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <– EAX + 42`
register	memory	`add EAX, [20]`	`EAX <– EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <– Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <– Mem[20] + 42`

Table 6.11 Memory addressing modes

Example	Meaning	Comment
`add EAX, [20]`	`EAX <– EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <– EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <– EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <– EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <– EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.12 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <– AH + BL`	8-bit
`add AX, −1`	`AX <– AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <– EAX + EDX`	32-bit

8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the RISC-V base addressing mode for loads and stores, but RISC-V instructions do not allow for scaling. x86 also provides a scaled index. In x86, the scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address. While RISC-V always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. Table 6.12 illustrates these variations.

Table 6.13 Selected EFLAGS

Name	Meaning
CF (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic
ZF (Zero Flag)	Result of last operation was zero
SF (Sign Flag)	Result of last operation was negative (msb = 1)
OF (Overflow Flag)	Overflow of two's complement arithmetic

6.8.3 Status Flags

x86, like many CISC architectures, uses condition flags (also called *status flags*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called EFLAGS, that stores the status flags. Some of the bits of the EFLAGS register are given in Table 6.13. Other bits are used by the operating system. The architectural state of an x86 processor includes EFLAGS as well as the eight registers and the EIP.

6.8.4 x86 Instructions

x86 has a larger set of instructions than RISC-V. Table 6.14 describes some of the general-purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. D indicates the destination (a register or memory location), and S indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32×32-bit multiplication always takes one of the sources from EAX and always puts the 64-bit result in EDX and EAX. LOOP always stores the loop counter in ECX. PUSH, POP, CALL, and RET use the stack pointer, ESP.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, JZ jumps if the zero flag (*ZF*) is 1. JNZ jumps if the zero flag is 0. The jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.15 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation. Unlike RISC-V, conditional jumps (called conditional branches in RISC-V) usually require two instructions instead of one.

Table 6.14 **Selected x86 instructions**

Instruction	Meaning	Function
ADD/SUB	add/subtract	D = D + S / D = D − S
ADDC	add with carry	D = D + S + CF
INC/DEC	increment/decrement	D = D + 1 / D = D − 1
CMP	compare	set flags based on D − S
NEG	negate	D = − D
AND/OR/XOR	logical AND/OR/XOR	D = D op S
NOT	logical NOT	D = $\overline{\text{D}}$
IMUL/MUL	signed/unsigned multiply	EDX:EAX = EAX × D
IDIV/DIV	signed/unsigned divide	EDX:EAX/D EAX = quotient; EDX = remainder
SAR/SHR	arithmetic/logical shift right	D = D >>> S / D = D >> S
SAL/SHL	left shift	D = D << S
ROR/ROL	rotate right/left	rotate D by S
RCR/RCL	rotate right/left with carry	rotate CF and D by S
BT	bit test	CF = D[S] (the S*th* bit of D)
BTR/BTS	bit test and reset/set	CF = D[S]; D[S] = 0 / 1
TEST	set flags based on masked bits	set flags based on D AND S
MOV	move	D = S
PUSH	push onto stack	ESP = ESP − 4; Mem[ESP] = S
POP	pop off stack	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	clear/set carry flag	CF = 0 / 1
JMP	unconditional jump	relative jump: EIP = EIP + S absolute jump: EIP = S
Jcc	conditional jump	if (flag) EIP = EIP + S

LOOP	loop	ECX = ECX −1
		if (ECX ≠ 0) EIP = EIP + imm

CALL	function call	ESP = ESP − 4;
		MEM[ESP] = EIP; EIP = S

RET	function return	EIP = MEM[ESP]; ESP = ESP + 4

Table 6.15 **Selected branch conditions**

Instruction	Meaning	Function after CMP D, S
JZ/JE	jump if ZF = 1	jump if D = S
JNZ/JNE	jump if ZF = 0	jump if D ≠ S
JGE	jump if SF = OF	jump if D ≥ S
JG	jump if SF = OF and ZF = 0	jump if D > S
JLE	jump if SF ≠ OF or ZF = 1	jump if D ≤ S
JL	jump if SF ≠ OF	jump if D < S
JC/JB	jump if CF = 1	
JNC	jump if CF = 0	
JO	jump if OF = 1	
JNO	jump if OF = 0	
JS	jump if SF = 1	
JNS	jump if SF = 0	

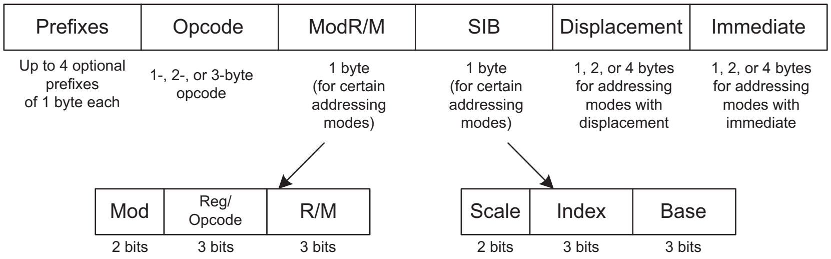

Prefixes
Opcode
ModR/M
SIB
Displacement
Immediate
Up to 4 optional prefixes of 1 byte each
1-, 2-, or 3-byte opcode
1 byte (for certain addressing modes)
1 byte (for certain addressing modes)
1, 2, or 4 bytes for addressing modes with displacement
1, 2, or 4 bytes for addressing modes with immediate
Mod
Reg/ Opcode
R/M
Scale
Index
Base
2 bits
3 bits
3 bits
2 bits
3 bits
3 bits

Figure 6.37 **x86 instruction encodings**

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike RISC-V, whose instructions are uniformly 32 bits (or 16 bits for compressed instructions), x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.37.[3] The *opcode* may be 1, 2, or 3 bytes. It is followed by four optional fields: *ModR/M*, *SIB*, *Displacement*, and *Immediate*. *ModR/M* specifies an addressing mode. *SIB* specifies the scale, index, and base registers in certain addressing

[3] It is possible to construct 17-byte instructions if all of the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

modes. *Displacement* indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. *Immediate* is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The *ModR/M* byte uses the 2-bit *Mod* and 3-bit *R/M* field to specify the addressing mode for one of the operands. The operand can come from one of the eight registers or from one of 24 memory addressing modes. Due to artifacts in the encodings, the ESP and EBP registers are not available for use as the base or index register in certain addressing modes. The *Reg* field specifies the register used as the other operand. For certain instructions that do not require a second operand, the *Reg* field is used to specify three more bits of the *opcode*.

In addressing modes using a scaled index register, the *SIB* byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the *SIB* byte also specifies the base register.

RISC-V fully specifies the instruction in the **op**, **funct3**, and **funct7** fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `ADD AL, imm8` performs an 8-bit add of an immediate to AL. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The A register (AL, AX, or EAX) is called the *accumulator*. On the other hand, `ADD D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, *D* (memory or a register). It is represented with the 1-byte opcode 0x80 followed by one or more bytes specifying *D*, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the opcode specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form. Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specified which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the opcode to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the prefix 0x66 appears before the opcode, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

x86 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 prefix is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid-1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, the first IA-64 chip was too late to market and never became a commercial success. Most computers needing the large address space now use the 64-bit extensions of x86.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, x86 can access 4 GB of memory. This was far more than the largest computers had in 1985. However, by the early 2000's, it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those interested in examining x86 architecture in more detail, the *x86 Intel Architecture Software Developer's Manual* is freely available on Intel's website.

6.8.7 The Big Picture

This section has given a taste of some of the differences between the RISC-V architecture and the x86 CISC architecture. x86 tends to have shorter programs because a complex instruction is equivalent to a series of simple RISC-V instructions and because the instructions are encoded to minimize memory usage. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs.

It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are:

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

RISC-V (RV32I) is a 32-bit architecture because it operates on 32-bit data. The RISC-V architecture has 32 general-purpose registers. In principle, almost any register can be used for any purpose. However, by convention, certain registers are reserved for certain purposes for ease of programming and so that functions written by different programmers can communicate easily. For example, register 0 (`zero`) always holds the constant 0, `ra` holds the return address after a `jal` instruction, and `a0` to `a7` hold the arguments of a function. `a0` to `a1` hold a function's return value. RISC-V has a byte-addressable memory system with 32-bit addresses. Instructions are 32 bits long and are word-aligned for efficient access. This chapter discussed the most commonly used RISC-V instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel Pentium processor in 1993 will generally still run (and run much faster) on the Intel i9 or AMD Ryzen processors in 2021.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture.

Microarchitecture is the link between hardware and software. We believe it is one of the most exciting topics in all of engineering: You will learn to build your own microprocessor!

Exercises

Exercise 6.1 Give three examples from the RISC-V architecture of each of the architecture design principles: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Exercise 6.2 The RISC-V architecture has a register set that consists of 32 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are advantages and disadvantages of this architecture over the RISC-V architecture?

Exercise 6.3 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) hello there

(b) bag o' chips

(c) To the rescue!

Exercise 6.4 Repeat Exercise 6.3 for the following strings.

(a) Cool

(b) RISC-V

(c) boo!

Exercise 6.5 Show how the strings in Exercise 6.3 are stored in a byte-addressable memory starting at memory address 0x004F05BC. The first character of the string is stored at the lowest byte address (in this case, 0x004F05BC). Clearly indicate the memory address of each byte.

Exercise 6.6 Repeat Exercise 6.5 for the strings in Exercise 6.4.

Exercise 6.7 The `nor` instruction is not part of the RISC-V instruction set because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `s3` = `s4` NOR `s5`. Use as few instructions as possible.

Exercise 6.8 The `nand` instruction is not part of the RISC-V instruction set because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `s3` = `s4` NAND `s5`. Use as few instructions as possible.

Exercise 6.9 Convert the following high-level code snippets into RISC-V assembly language. Assume that the (signed) integer variables g and h are in registers a0 and a1, respectively. Clearly comment your code.

a)
```
if (g > h)
  g = g + 1;
else
  h = h − 1;
```

b)
```
if (g <= h)
  g = 0;
else
  h = 0;
```

Exercise 6.10 Repeat Exercise 6.9 for the following code snippets.

a)
```
if (g >= h)
  g = g + h;
else
  g = g − h;
```

b)
```
if (g < h)
  h = h + 1;
else
  h = h * 2;
```

Exercise 6.11 Convert the following high-level code snippet into RISC-V assembly. Assume that the base addresses of array1 and array2 are held in t1 and t2 and that the array2 array is initialized before it is used. Use as few instructions as possible. Clearly comment your code.

```
int i;
int array1[100];
int array2[100];
...
for (i = 0; i < 100; i = i + 1)
   array1[i] = array2[i];
```

Exercise 6.12 Repeat Exercise 6.11 for the following high-level code snippet. Assume that the temp array is initialized before it is used and that t3 holds the base address of temp.

```
int i;
int temp[100];
...
for (i = 0; i < 100; i = i + 1)
 temp[i] = temp[i] * 128;
```

Exercise 6.13 Write RISC-V assembly code for placing the following immediates (constants) in s7. Use a minimum number of instructions.

(a) 29

(b) –214

(c) –2999

(d) 0xABCDE000

(e) 0xEDCBA123

(f) 0xEEEEEFAB

Exercise 6.14 Repeat Exercise 6.13 for the following immediates.

(a) 47

(b) -349

(c) 5328

(d) 0xBBCCD000

(e) 0xFEEBC789

(f) 0xCCAAB9AB

Exercise 6.15 Write a function in a high-level language for int find42(int array[], int size). size specifies the number of elements in array, and array[] specifies the base address of the array. The function should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value –1. Clearly comment your code.

Exercise 6.16 The high-level function strcpy (string copy) copies the character string src to the character string dst.

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Implement the strcpy function in RISC-V assembly code. Use t0 for i.

(b) Draw a picture of the stack before, during, and after the strcpy function call. Assume sp = 0xFFC000 just before strcpy is called.

This simple string copy function has a serious flaw: it has no way of knowing that dst has enough space to receive src. If a malicious programmer were able to execute strcpy with a long string src, the programmer might be able to write bytes all over memory, possibly even modifying code stored in subsequent memory locations. With some cleverness, the modified code might take over the machine. This is called a *buffer overflow attack*; it is employed by several nasty programs, including the infamous Blaster worm, which caused an estimated $525 million in damages in 2003.

Exercise 6.17 Convert the high-level function from Exercise 6.15 into RISC-V assembly code. Clearly comment your code.

Exercise 6.18 Consider the RISC-V assembly code below. `func1`, `func2`, and `func3` are nonleaf functions. `func4` is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function. You may assume that the functions do not need to save any nonpreserved registers on their stacks.

```
0x00091000 func1: ...    # func1 uses t2–t3, s4–s10
...
0x00091020   jal func2
...
0x00091100 func2: ...    # func2 uses a0–a2, s0–s5
...
0x0009117C   jal func3
...
0x00091400 func3: ...    # func3 uses t3, s7–s9
...
0x00091704   jal func4
...
0x00093008 func4: ...    # func4 uses s10–s12
...
0x00093118   jr  ra
```

(a) How many words are the stack frames of each function?

(b) Sketch the stack after `func4` is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Give values where possible. Assume that `sp` = 0xABC124 just before `func1` is called.

Exercise 6.19 Each number in the Fibonacci series is the sum of the previous two numbers. Table 6.16 lists the first few numbers in the series, *fib*(*n*).

(a) What is *fib*(*n*) for $n = 0$ and $n = -1$?

(b) Write a function called `fib` in a high-level language that returns the Fibonacci number for any nonnegative value of `n`. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into RISC-V assembly code. Add comments after every line of code that explain clearly what it does. Use a simulator to test your code on *fib*(9). (See the Preface for links to a RISC-V simulator.)

Table 6.16 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	...
fib(n)	1	1	2	3	5	8	13	21	34	55	89	...

Exercise 6.20 Consider Code Example 6.28. For this exercise, assume *factorial*(*n*) is called with input argument $n = 5$.

(a) What value is in `a0` when *factorial* returns to the calling function?

(b) Suppose you replace the instructions at addresses 0x8508 and 0x852C with nops. Will the program:
   (1) enter an infinite loop but not crash;
   (2) crash (cause the stack to grow or shrink beyond the dynamic data segment or the PC to jump to a location outside the program);
   (3) produce an incorrect value in `a0` when the program returns to loop (if so, what value?); or
   (4) run correctly despite the deleted lines?

(c) Repeat part (b) with the following instruction modifications:
   (1) Replace the instructions at addresses 0x8504 and 0x8528 with nops.
   (2) Replace the instruction at address 0x8518 with a nop.
   (3) Replace the instruction at address 0x8530 with a nop.

Exercise 6.21 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative b. He goes overboard in the use of function calls and recursion and produces the following high-level code for functions `f` and `g`.

```
// high-level code for functions f and g
int f(int a, int b) {
  int j;

  j = a;
  return j + a + g(b);
}

int g(int x) {
  int k;

  k = 3;
  if (x == 0) return 0;
  else return k + g(x − 1);
}
```

Ben then translates the two functions into RISC-V assembly language as follows. He also writes a function, `test`, that calls the function `f(5,3)`.

```
# RISC-V assembly code
# f: a0 = a, a1 = b, s4 = j;
# g: a0 = x, s4 = k

0x8000 test:  addi  a0, zero, 5    # a = 5
0x8004        addi  a1, zero, 3    # b = 3
0x8008        jal   f              # call f(5, 3)
0x800C loop:  j     loop           # and loop forever
0x8010 f:     addi  sp, sp, −16    # make room on stack
0x8014        sw    a0, 0xC(sp)    # save a0
0x8018        sw    a1, 0x8(sp)    # save a1
```

```
0x801C        sw    ra, 0x4(sp)     # save ra
0x8020        sw    s4, 0x0(sp)     # save s4
0x8024        addi  s4, a0, 0       # j = a
0x8028        addi  a0, a1, 0       # place b as argument for g()
0x802C        jal   g               # call g
0x8030        lw    t0, 0xC(sp)     # restore a into t0
0x8034        add   a0, a0, t0      # a0 = g(b) + a
0x8038        add   a0, a0, s4      # a0 = (g(b) + a) + j
0x803C        lw    s4, 0x0(sp)     # restore registers
0x8040        lw    ra, 0x4(sp)
0x8044        addi  sp, sp, 16
0x8048        jr    ra              # return
0x804C g:     addi  sp, sp, −8      # make room on stack
0x8050        sw    ra, 4(sp)       # save registers
0x8054        sw    s4, 0(sp)
0x8058        addi  s4, zero, 3     # k = 3
0x805C        bne   a0, zero, else  # if (x != 0), goto else
0x8060        addi  a0, zero, 0     # return 0
0x8064        j     done            # clean up and return
0x8068 else:  addi  a0, a0, −1      # decrement x
0x806C        jal   g               # call g(x−1)
0x8070        add   a0, s4, a0      # return k + g(x−1)
0x8074 done:  lw    s4, 0(sp)       # restore registers
0x8078        lw    ra, 4(sp)
0x807C        addi  sp, sp, 8
0x8080        jr    ra              # return
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.10 to help you answer the following questions.

(a) If the code runs starting at test, what value is in a0 when the program gets to loop? Does his program correctly compute $2a + 3b$?

(b) Suppose Ben replaces the instruction at address 0x8014 with a nop. Will the program
   (1) enter an infinite loop but not crash;
   (2) crash (cause the stack to grow or shrink beyond the dynamic data segment or the PC to jump to a location outside the program);
   (3) produce an incorrect value in a0 when the program returns to loop (if so, what value?); or
   (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the following instructions are changed. Note that labels aren't changed, only instructions.
   (i) Instructions at 0x8014 and 0x8030 are replaced with nops.
   (ii) Instructions at 0x803C and 0x8040 are replaced with nops.
   (iii) Instruction at 0x803C is replaced with a nop.
   (iv) Instruction at 0x8030 is replaced with a nop.
   (v) Instructions at 0x8054 and 0x8074 are replaced with nops.

(vi) Instructions at 0x8020 and 0x803C are replaced with nops.

(vii) Instructions at 0x8050 and 0x8078 are replaced with nops.

Exercise 6.22 Convert the following RISC-V assembly code into machine language. Write the instructions in hexadecimal.

```
addi s3, s4, 28
sll  t1, t2, t3
srli s3, s1, 14
sw   s9, 16(t4)
```

Exercise 6.23 Repeat Exercise 6.22 for the following RISC-V assembly code:

```
add  s7, s8, s9
srai t0, t1, 0xC
ori  s3, s1, 0xABC
lw   s4, 0x5C(t3)
```

Exercise 6.24 Consider all instructions with an immediate field.

(a) Which instructions from Exercise 6.22 use an immediate field in their machine code format?

(b) What is the instruction type (I-, S-, B-, U-, or J-type) for the instructions from part (a)?

(c) Write out the 5- to 21-bit immediates of each instruction from part (a) in hexadecimal. If the number is extended, also write them as 32-bit extended immediates. Otherwise, indicate that they are not extended.

Exercise 6.25 Repeat Exercise 6.24 for the instructions in Exercise 6.23.

Exercise 6.26 Consider the RISC-V machine code snippet below. The first instruction is listed at the top.

(a) Convert the machine code snippet into RISC-V assembly language.

(b) Reverse engineer a high-level program that would compile into this assembly language routine and write it. Clearly comment your code.

(c) Explain in words what the program does. `a0` and `a1` are the inputs, and they initially contain positive numbers, `A` and `B`. At the end of the program, register `a0` holds the output (i.e., return value).

```
0x01800513
0x00300593
0x00000393
0x00058E33
0x01C54863
0x00138393
0x00BE0E33
0xFF5FF06F
0x00038533
```

Exercise 6.27 Repeat Exercise 6.26 for the following machine code. a0 and a1 are the inputs. a0 contains a 32-bit number and a1 is the address of a 32-element array of characters (char).

```
0x01F00393
0x00755E33
0x001E7E13
0x01C580A3
0x00158593
0xFFF38393
0xFE03D6E3
0x00008067
```

Exercise 6.28 Convert the following branch instructions into machine code. Instruction addresses are given to the left of each instruction.

a)
```
0x0000A000       beq t4, zero, Loop
0x0000A004       ...
0x0000A008       ...
0x0000A00C Loop: ...
```

b)
```
0x00801000       bne s5, a1, L1
...              ...
0x0080174C L1:   ...
```

c)
```
0x0000C10C Back: ...
...              ...
0x0000D000       blt s1, s2, Back
```

d)
```
0x01030AAC       bge t4, t6, L2
...              ...
0x01031AA4 L2:   ...
```

e)
```
0x0BC08004 L3:   ...
...              ...
0x0BC09000       beq s3, s7, L3
```

Exercise 6.29 Convert the following branch instructions into machine code. Instruction addresses are given to the left of each instruction.

a)
```
0xAA00E124       blt t4, s3, Loop
0xAA00E128       ...
0xAA00E12C       ...
0xAA00E130 Loop: ...
```

b)
```
0xC0901000       bge t1, t2, L1
...              ...
0xC090174C L1:   ...
```

c)
```
0x1230D10C Back: ...
...              ...
0x1230D908       bne s10, s11, Back
```

d)
```
0xAB0C99A8       beq a0, s1, L2
...              ...
0xAB0CA0FC L2:   ...
```

e)
```
0xFFABCF04 L3:   ...
...              ...
0xFFABD640       blt s1, t3, L3
```

Exercise 6.30 Convert the following jump instructions into machine code. Instruction addresses are given to the left of each instruction.

a)
```
0x1234ABC0       j Loop
...              ...
0x123CABBC Loop: ...
```

b)
```
0x12345678 Back: ...
...              ...
0x123B8760       jal s0, Back
```

c)
```
0xAABBCCD0       jal L1
...              ...
0xAABDCD98 L1:   ...
```

d)
```
0x11223344       j L2
...              ...
0x1127BCDC L2:   ...
```

e)
```
0x9876543C L3:   ...
...              ...
0x9886543C       jal L3
```

Exercise 6.31 Convert the following jump instructions into machine code. Instruction addresses are given to the left of each instruction.

a)
```
0x0000ABC0       jal Loop
...              ...
0x0000EEEC Loop: ...
```

b)
```
0x0000C10C Back: ...
...              ...
0x000F1230       jal Back
```

c)
```
0x00801000       jal s1, L1
...              ...
0x008FFFDC L1:   ...
```

d)
```
0xA1234560       j L2
...              ...
0xA131347C L2:   ...
```

e)
```
0xF0BBCCD4 L3:   ...
...              ...
0xF0CBCCD4       j L3
```

Exercise 6.32 Consider the following RISC-V assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0xA0028 Func1: addi t4, a1, 0
0xA002C        ori  a0, a0, 32
0xA0030        sub  a1, a1, a0
0xA0034        jal  Func2
...            ...
0xA0058 Func2: lw   t2, 4(a0)
0xA005C        sw   t2, 16(a1)
0xA0060        srli t3, t2, 8
0xA0064        beq  t2, t3, Else
0xA0068        jr   ra
0xA006C Else:  addi a0, a0, 4
0xA0070        j    Func2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

(b) List the instruction type and addressing mode used at each line of code.

Exercise 6.33 Consider the following C code snippet.

```
// C code
void setArray(int num) {
  int i;
  int array[10];

for (i = 0; i < 10; i = i + 1)
  array[i] = compare(num, i);
}

int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
 else
   return 0;
}
int sub(int a, int b) {
   return a − b;
}
```

(a) Implement the C code snippet in RISC-V assembly language. Use s4 to hold the variable i. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the setArray function (see the end of Section 6.3.7). Clearly comment your code.

(b) Assume that setArray is the first function called. Draw the status of the stack before calling setArray and during each function call. Indicate stack addresses and the names of registers and variables stored on the stack; mark the location of sp; and clearly mark each stack frame. Assume that sp starts at 0x8000.

(c) How would your code function if you failed to store ra on the stack?

Exercise 6.34 Consider the following high-level function.

```
// C code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0)
    b = 10;
  else
    b = b + (n * n) + f(n − 1, k + 1);
  return b * k;
}
```

(a) Translate the high-level function f into RISC-V assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the RISC-V preserved register conventions. Clearly comment your code. Assume that the function starts at instruction address 0x8100. Keep local variable b in s4. Clearly comment your code.

(b) Step through your function from part (a) by hand for the case of f(2,4). Draw a picture of the stack similar to the one in Figure 6.10, and assume that sp is equal to 0xBFF00100 when f is called. Write the stack addresses and the register name and data value stored at each location in the stack and keep track of the stack pointer value (sp). Clearly mark each stack frame. You might also find it useful to keep track of the values in a0, a1, and s4 throughout execution. Assume that when f is called, s4 = 0xABCD and ra = 0x8010.

(c) What is the final value of a0 when f(2,4) is called?

Exercise 6.35 What is the maximum number of instructions that a branch instruction (like beq) can branch forward (i.e., to higher instruction addresses)?

Exercise 6.36 What is the maximum number of instructions that a branch instruction (like beq) can branch backward (i.e., to lower instruction addresses)?

Exercise 6.37 Write assembly code that conditionally branches to an instruction 32 Minstructions forward from a given instruction. Recall that 1 Minstruction = 2^{20} instructions = 1,048,576 instructions. Assume that your code begins at address 0x8000. Use a minimum number of instructions.

Exercise 6.38 Explain why it is advantageous to have a large immediate field in the machine format for the jump and link instruction, `jal`.

Exercise 6.39 Consider a function that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format.

(a) Write this function in high-level code.

(b) Convert this function to RISC-V assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.40 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a function called `concat` that concatenates (joins together) the two strings: `void concat(char string1[], char string2[], char stringconcat[])`. The function does not return a value. It concatenates `string1` and `string2` and places the resulting string in `stringconcat`. You may assume that the character array `stringconcat` is large enough to accommodate the concatenated string. Clearly comment your code.

(b) Convert the function from part (a) into RISC-V assembly language. Clearly comment your code.

Exercise 6.41 Write a RISC-V assembly program that adds two positive single-precision floating-point numbers held in `a0` and `a1`. Do not use any of the RISC-V floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, etc.) or numbers that overflow or underflow. Use a simulator to test your code. (See the Preface for links to a RISC-V simulator.) You will need to manually set the values of `a0` and `a1` to test your code. Demonstrate that your code functions reliably. Clearly comment your code.

Exercise 6.42 Expand the RISC-V assembly program from Exercise 6.41 to handle both positive and negative single-precision floating-point numbers. Clearly comment your code.

Exercise 6.43 Consider a function that sorts a 10-element array called `scores` from lowest to highest. After the function completes, `scores[0]` holds the smallest value and `scores[9]` holds the highest value.

(a) Write a high-level `sort` function that performs the function above. `sort` receives a single argument, the address of the `scores` array. Clearly comment your code.

(b) Convert the `sort` function into RISC-V assembly language. Clearly comment your code.

Exercise 6.44 Consider the following RISC-V program. Assume that the instructions are placed starting at memory address 0x8400 and that global variables x and y are at memory addresses 0x10024 and 0x10028, respectively.

```
# RISC-V assembly code
main:
  addi sp, sp, -4    # make room on stack
  sw   ra, 0(sp)     # save ra on stack
  lw   a0, -940(gp)  # a0 = x
  lw   a1, -936(gp)  # a1 = y
  jal  diff          # call diff()
  lw   ra, 0(sp)     # restore registers
  addi sp, sp, 4
  jr   ra            # return
diff:
  sub  a0, a0, a1    # return (a0-a1)
  jr   ra
```

(a) First, show the instruction address next to each assembly instruction.

(b) Describe the symbol table: that is, list the name, address, and size of each symbol (i.e., function label and global variable).

(c) Convert all instructions into machine code.

(d) How big (how many bytes) are the data and text segments?

(e) Sketch a memory map showing where data and instructions are stored, similar to Figure 6.34. Be sure to label the values of PC and gp at the beginning of the program.

Exercise 6.45 Repeat Exercise 6.44 for the following RISC-V code. Assume that the instructions are placed starting at memory address 0x8534 and that global variables g and h are at memory addresses 0x1305C and 0x13060.

```
# RISC-V assembly code
main:       addi sp, sp, -8
            sw   ra, 4(sp)
            sw   s4, 0(sp)
            addi s4, zero, 15
            sw   s4, -300(gp)  # g = 15
            addi a1, zero, 27  # arg1 = 27
            sw   a1, -296(gp)  # h = 27
            lw   a0, -300(gp)  # arg0 = g = 15
            jal  greater
            lw   s4, 0(sp)
            lw   ra, 4(sp)
            addi sp, sp, 8
            jr   ra
greater:    blt  a1, a0, isGreater
            addi a0, zero, 0
            jr   ra
isGreater:  addi a0, zero, 1
            jr   ra
```

Exercise 6.46 Explain the advantages and disadvantages of the bit-swizzling found in the encodings of RISC-V immediates.

Exercise 6.47 Consider sign-extension of the immediates in RISC-V instructions. Design a sign-extension unit for RISC-V immediates using the steps below. Minimize the amount of hardware needed.

(a) Sketch the schematic for a sign-extension unit that sign-extends the 12-bit immediate in I-type instructions. The circuit's input is the upper 12 bits of the instruction, $Instr_{31:20}$, which encodes the 12-bit signed immediate. The output is a 32-bit sign-extended immediate, $ImmExt_{31:0}$.

(b) Expand the sign-extension unit from part (a) to also sign-extend the 12-bit immediate found in S-type instructions. Modify the inputs as needed, and reuse hardware when possible.

(c) Expand the sign-extension unit from part (b) to also sign-extend the 13-bit immediate found in B-type instructions.

(d) Expand the sign-extension unit from part (c) to also sign-extend the 21-bit immediate found in J-type instructions.

Exercise 6.48 In this exercise, you will design an alternate extension unit for RISC-V immediates using a minimum amount of hardware. Suppose that the RISC-V architects had chosen to use immediate encodings that were more straightforward to humans, as shown in Figure 6.38. This figure shows all instruction fields except **op**. The immediate encodings in Figure 6.38 use no bit-swizzling (but still split the immediate field in S/B-type instructions across two instruction fields). Bits that differ from the actual RISC-V immediate encodings (shown in Figure 6.27) are highlighted in blue. Specifically, these hypothetical (straightforward) immediate encodings differ from actual RISC-V immediate encodings for the B- and J-type formats.

(a) Sketch the schematic for a sign-extension unit that sign-extends the 12-bit immediate in I-type instructions. The circuit's input is the upper 12 bits of

11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd					I
11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0	S
12	11	10	9	8	7	6	rs2					rs1					funct3			5	4	3	2	1	B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd					U
20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	rd					J
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	

Figure 6.38 Alternate immediate encodings

the instruction, $Instr_{31:20}$, which encodes the 12-bit signed immediate. The output is a 32-bit sign-extended immediate, $ImmExt_{31:0}$.

(b) Expand the sign-extension unit from part (a) to also sign-extend the 12-bit immediate found in S-type instructions. Modify the inputs as needed, and reuse hardware when possible.

(c) Expand the sign-extension unit from part (b) to also sign-extend the 13-bit immediate found in (the modified) B-type instructions (see Figure 6.38).

(d) Expand the sign-extension unit from part (c) to also sign-extend the 21-bit immediate found in (the modified) J-type instructions (see Figure 6.38).

(e) If you completed Exercise 6.47, compare this to the hardware in the actual RISC-V extension unit.

Exercise 6.49 Consider how far `jal` instructions can jump.

(a) How many instructions can a `jal` instruction jump forward (i.e., to higher addresses)?

(b) How many instructions can a `jal` instruction jump backward (i.e., to lower addresses)?

Exercise 6.50 Consider memory storage of a 32-bit word stored at the 42nd memory word in a byte-addressable memory. Remember, the 0th word is stored at memory address 0, the first word at address 4, and so on.

(a) What is the byte address of the 42nd word stored in memory?

(b) What are the byte addresses that the 42nd word spans?

(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Clearly label the byte address corresponding to each data byte value.

Exercise 6.51 Repeat Exercise 6.50 for memory storage of a 32-bit word stored at the 15th word in a byte-addressable memory.

Exercise 6.52 Explain how the following RISC-V program can be used to determine whether a computer is big-endian or little-endian:

```
addi s7, 100
lui  s3, 0xABCD8     # s3 = 0xABCD8000
addi s3, s3, 0x765   # s3 = 0xABCD8765
sw   s3, 0(s7)
lb   s2, 1(s7)
```

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 6.1 Write a RISC-V assembly language program for swapping the contents of two registers, `a0` and `a1`. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write RISC-V assembly code that finds the subset of the array with the largest sum. Assume that the array's base address and the number of array elements are in `a0` and `a1`, respectively. Your code should place the resulting subset of the array starting at the base address in `a2`. Write code that runs as fast as possible.

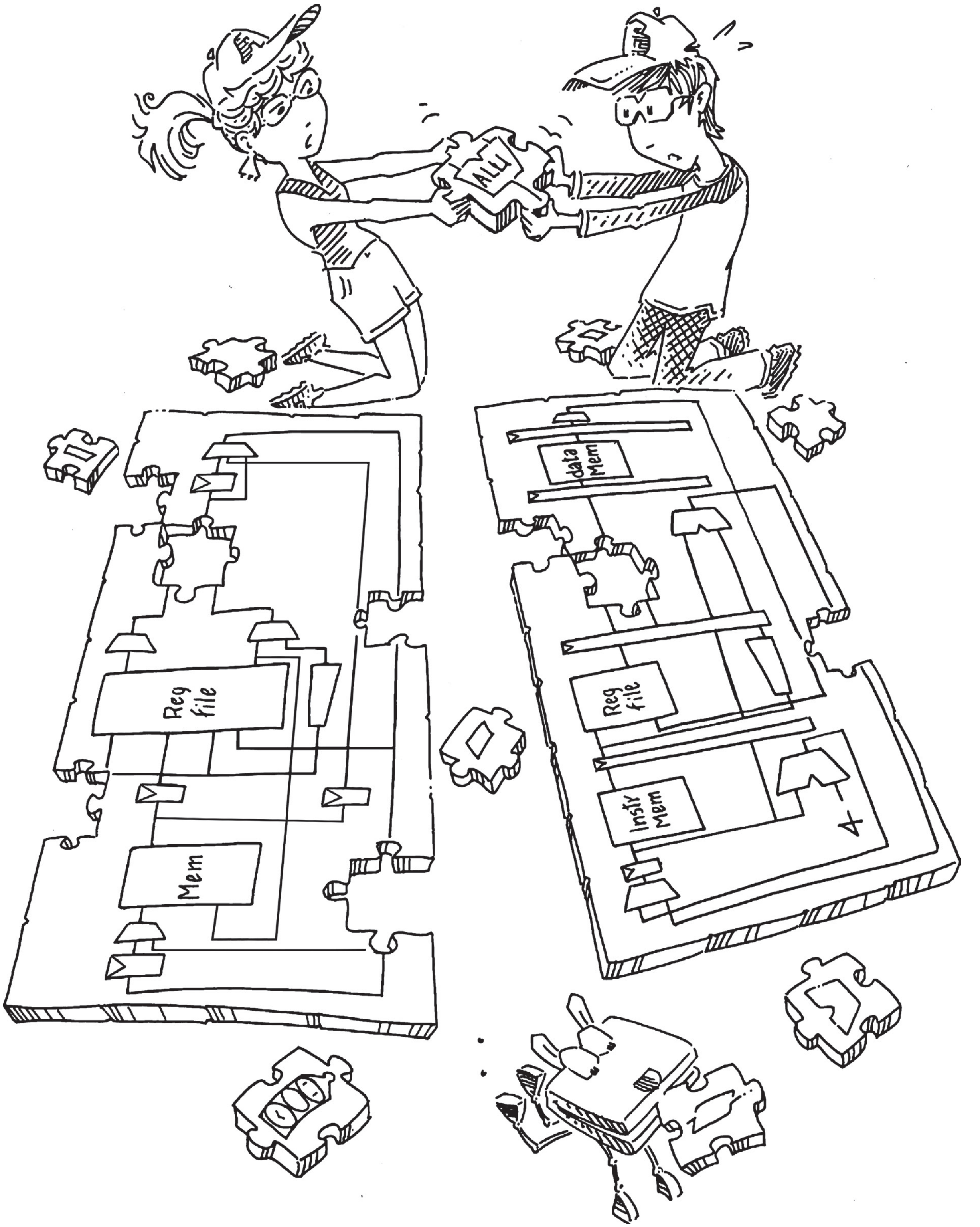
Question 6.3 You are given an array that holds a sentence in a null-terminated string. Write a RISC-V assembly language program to reverse the words in the sentence and store the new sentence back in the array.

Question 6.4 Write a RISC-V assembly language program to count the number of ones in a 32-bit number.

Question 6.5 Write a RISC-V assembly language program to reverse the bits in a register. Use as few instructions as possible.

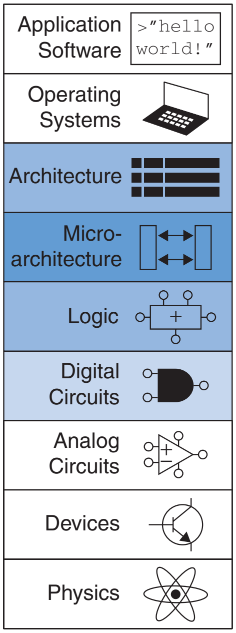
Question 6.6 Write a succinct RISC-V assembly language program to test whether overflow occurs when `a2` is subtracted from `a3`.

Question 6.7 Write a RISC-V assembly language program for testing whether a given string is a palindrome. (Recall that a palindrome is a word that is the same forward and backward. For example, the words "wow" and "racecar" are palindromes.)


ALU
data Mem
Reg file
Reg file
Mem
Instr Mem
4

Microarchitecture

7



7.1 INTRODUCTION

In this chapter, you will learn how to piece together a microprocessor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward and, by this point, you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory. And you have learned about the RISC-V architecture, which specifies the programmer's view of the RISC-V processor in terms of registers, instructions, and memory.

This chapter covers *microarchitecture*, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, arithmetic logic units (ALUs), finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as RISC-V, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We design three different microarchitectures in this chapter to illustrate the trade-offs.

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and architectural state. The *architectural state* for the RISC-V processor consists of the program counter and the 32 32-bit registers. Any RISC-V microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00007-6

microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance; we point this out as it arises.

To keep the microarchitectures easy to understand, we focus on a subset of the RISC-V instruction set. Specifically, we handle the following instructions:

- R-type instructions: `add`, `sub`, `and`, `or`, `slt`
- Memory instructions: `lw`, `sw`
- Branches: `beq`

These particular instructions were chosen because they are sufficient to write useful programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

The *architectural state* is the information necessary to define what a computer is doing. If one were to save a copy of the architectural state and contents of memory, then turn off a computer, then turn it back on and restore the architectural state and memory, the computer would resume the program it was running, unaware that it had been powered off and back on. Think of a science fiction novel in which the protagonist's brain is frozen, then thawed years later to wake up in a new world.

7.1.2 Design Process

We divide our microarchitectures into two interacting parts: the *datapath* and the *control unit*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. We are implementing the 32-bit RISC-V (RV32I) architecture, so we use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.

In this chapter, heavy lines indicate 32-bit data busses. Medium lines indicate narrower busses, such as the 5-bit address busses on the register file. Narrow lines indicate 1-bit wires, and blue lines are used for control signals, such as the register file write enable. Registers usually have a reset input to put them into a known state at start-up, but reset is not shown to reduce clutter.

The *program counter* (PC) points to the current instruction. Its input, *PCNext*, indicates the address of the next instruction.

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. Upon reset, RISC-V processors normally initialize the PC to a low address in memory, such as 0x00001000, and we start our programs there.

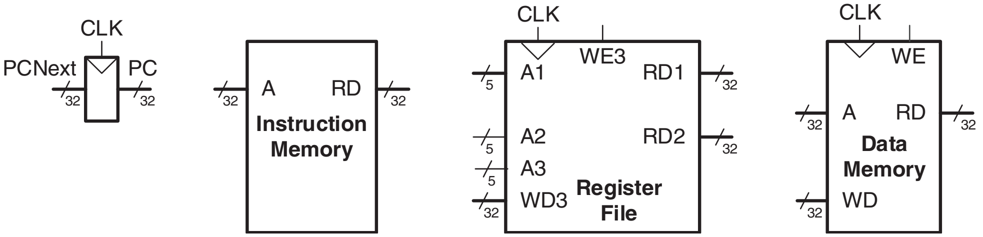


Figure 7.1 State elements of a RISC-V processor

The *instruction memory* has a single read port.[1] It takes a 32-bit instruction address input, *A*, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.

The 32-element × 32-bit register file holds registers `x0`-`x31`. Recall that `x0` is hardwired to 0. The register file has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of the $2^5 = 32$ registers as source operands. The register file places the 32-bit register values onto read data outputs *RD1* and *RD*2. The write port, port 3, takes a 5-bit address input, *A3*; a 32-bit write data input, *WD3*; a write enable input, *WE3*; and a clock. If its write enable (*WE3*) is asserted, then the register file writes the data (*WD3*) into the specified register (*A3*) on the rising edge of the clock.

The *data memory* has a single read/write port. If its write enable, *WE*, is asserted, then it writes data *WD* into address *A* on the rising edge of the clock. If its write enable is 0, then it reads from address *A* onto the read data bus, *RD*.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, then the new data appears at *RD* after some propagation delay; no clock is involved. The clock controls writing only. These memories are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must set up before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. A microprocessor

[1] This is an oversimplification used to treat the instruction memory as a ROM. In most real processors, the instruction memory must be writable so that the operating system (OS) can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a single memory that contains both instructions and data and that can be both read and written.

is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, a processor can be viewed as a giant finite state machine or as a collection of simpler interacting state machines.

7.1.3 Microarchitectures

In this chapter, we develop three microarchitectures for the RISC-V architecture: single-cycle, multicycle, and pipelined. They differ in how the state elements are connected and in the amount of nonarchitectural state needed.

The *single-cycle microarchitecture* executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction. Moreover, the processor requires separate instruction and data memories, which is generally unrealistic.

Examples of classic multicycle processors include the 1947 MIT Whirlwind, the IBM System/360, the Digital Equipment Corporation VAX, the 6502 used in the Apple II, and the 8088 used in the IBM PC. Multicycle microarchitectures are still used in inexpensive microcontrollers such as the 8051, the 68HC11, and the PIC16-series found in appliances, toys, and gadgets.

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks, such as adders and memories. For example, the adder may be used on different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by introducing several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. This processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data. Because they use less hardware than single-cycle processors, multicycle processors were the historical choice for inexpensive systems.

Intel processors have been pipelined since the 80486 was introduced in 1989. Nearly all RISC microprocessors are also pipelined, and all commercial RISC-V processors have been pipelined. Because of the decreasing cost of transistors, pipelined processors now cost fractions of a penny, and the entire system, with memory and peripherals, costs 10's of cents. Thus, pipelined processors are replacing their slower multicycle siblings in even the most cost-sensitive applications.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. Pipelined processors must access instructions and data in the same cycle; they generally use separate instruction and data caches for this purpose, as discussed in Chapter 8. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to achieve even more speed in modern high-performance microprocessors.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Precise cost calculations require detailed knowledge of the implementation technology but, in general, more gates and more memory mean more dollars.

This section lays the foundation for analyzing performance. There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real-world performance. For example, microprocessor makers often market their products based on the clock frequency and the number of cores. However, they gloss over the complications that some processors accomplish more work than others in a clock cycle and that this varies from program to program. What is a buyer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run. This may be necessary if you have not written your program yet or if somebody else who does not have your program is making the measurements. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

Dhrystone, CoreMark, and SPEC are three popular benchmarks. The first two are *synthetic benchmarks* composed of important common pieces of programs. Dhrystone was developed in 1984 and remains commonly used for embedded processors, although the code is somewhat unrepresentative of real-life programs. CoreMark is an improvement over Dhrystone and involves matrix multiplications that exercise the multiplier and adder, linked lists to exercise the memory system, state machines to exercise the branch logic, and cyclical redundancy checks that involve many parts of the processor. Both benchmarks are less than 16 KB in size and do not stress the instruction cache.

The SPECspeed 2017 Integer benchmark from the Standard Performance Evaluation Corporation (SPEC) is composed of real programs, including x264 (video compression), deepsjeng (an artificial intelligence chess player), omnetpp (simulation), and GCC (a C compiler). The benchmark is widely used for high-performance processors because it stresses the entire system in a representative way.

When customers buy computers based on benchmarks, they must be careful because computer makers have strong incentive to bias the benchmark. For example, Dhrystone involves extensive string copying, but the strings are of known constant length and word alignment. Thus, a smart compiler may replace the usual code involving loops and byte accesses with a series of word loads and stores, improving Dhrystone scores by more than 30% but not speeding up real-world applications. The SPEC89 benchmark contained a Matrix 300 program in which 99% of the execution time was in one line. IBM sped up the program by a factor of 9 using a compiler technique called *blocking*. Benchmarking multicore computing is even harder because there are many ways to write programs, some of which speed up in proportion to the number of cores available but are inefficient on a single core. Others are fast on a single core but scarcely benefit from extra cores.

Equation 7.1 gives the execution time of a program, measured in seconds.

$$ExecutionTime = (\#\,instructions)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \tag{7.1}$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we assume that we are executing known programs on a RISC-V processor, so the number of instructions for each program is constant, independent of the microarchitecture. The *cycles per instruction* (*CPI*) is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (*instructions per cycle*, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we assume we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic in the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances also improve transistor speed, so a microprocessor built today will be faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

Many other factors affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world does not help surfing the Internet on a poor connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

We first design a microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is performed by the datapath at any given time. The control unit contains combinational logic that generates the appropriate control signals based on the current instruction. Finally, we analyze the performance of the single-cycle processor.

7.3.1 Sample Program

For the sake of concreteness, we will have the single-cycle processor run the short program from Figure 7.2 that exercises loads, stores, an R-type instruction (`or`), and a branch (`beq`). Suppose that the program is stored in memory starting at address 0x1000. The figure indicates the address of each instruction, the instruction type, the instruction fields, and the hexadecimal machine language code for the instruction.

Assume that register `x5` initially contains the value 6 and `x9` contains 0x2004. Memory location 0x2000 contains the value 10. The program counter begins at 0x1000. The `lw` reads 10 from address (0x2004 – 4) = 0x2000 and puts it in `x6`. The `sw` writes 10 to address (0x2004 + 8) = 0x200C. The `or` computes `x4` = 6 | 10 = 0110_2 | 1010_2 = 1110_2 = 14. Then, `beq` goes back to label `L7`, so the program repeats forever.

7.3.2 Single-Cycle Datapath

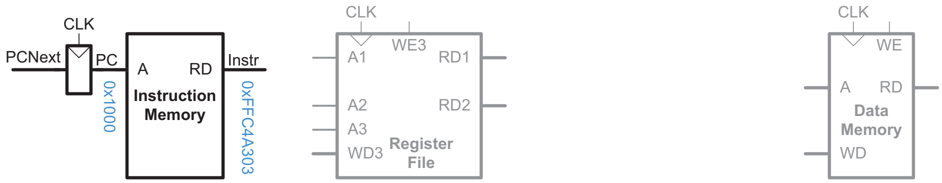
This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray. The example instruction being executed is shown at the bottom of each figure.

The program counter contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.3 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*. In our sample program from Figure 7.2, PC is 0x1000. (Note that this is a 32-bit processor, so PC is really 0x00001000, but we omit leading zeros to avoid cluttering the figure.)

We italicize signal names in the text but not the names of hardware modules. For example, *PC* is the signal coming out of the PC register, or simply, the PC.

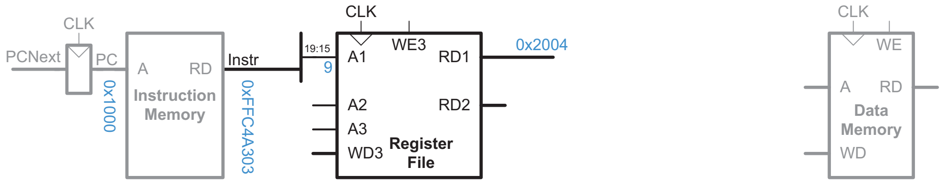
Address	Instruction	Type	Fields						Machine Language
0x1000	L7: lw x6, -4(x9)	I	$imm_{11:0}$ 111111111100		rs1 01001	f3 010	rd 00110	op 0000011	FFC4A303
0x1004	sw x6, 8(x9)	S	$imm_{11:5}$ 0000000	rs2 00110	rs1 01001	f3 010	$imm_{4:0}$ 01000	op 0100011	0064A423
0x1008	or x4, x5, x6	R	funct7 0000000	rs2 00110	rs1 00101	f3 110	rd 00100	op 0110011	0062E233
0x100C	beq x4, x4, L7	B	$imm_{12,10:5}$ 1111111	rs2 00100	rs1 00100	f3 000	$imm_{4:1,11}$ 10101	op 1100011	FE420AE3

Figure 7.2 Sample program exercising different types of instructions



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.3 Fetch instruction from memory



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.4 Read source operand from register file

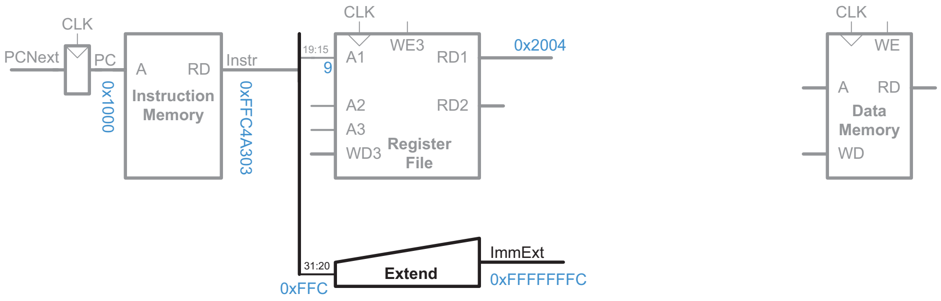
Instr is `lw`, 0xFFC4A303, as also shown at the bottom of Figure 7.3. These sample values are annotated in light blue on the diagram.

The processor's actions depend on the specific instruction that was fetched. First, we will work out the datapath connections for the `lw` instruction. Then, we will consider how to generalize the datapath to handle other instructions.

`lw`

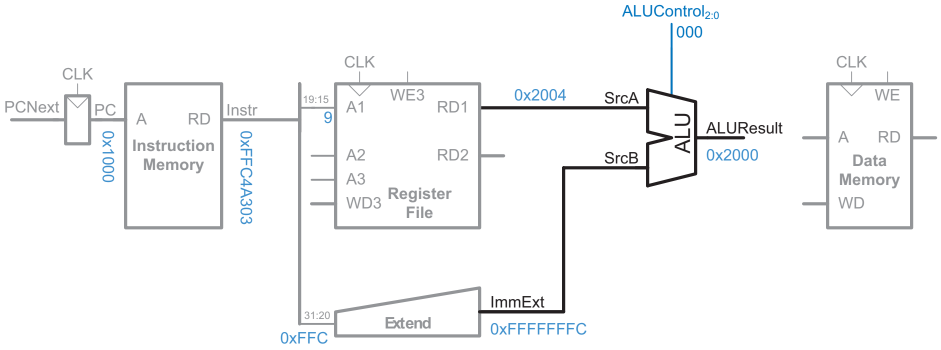
For the `lw` instruction, the next step is to read the source register containing the base address. Recall that `lw` is an I-type instruction, and the base register is specified in the **rs1** field of the instruction, $Instr_{19:15}$. These bits of the instruction connect to the *A1* address input of the register file, as shown in Figure 7.4. The register file reads the register value onto *RD1*. In our example, the register file reads 0x2004 from `x9`.

The `lw` instruction also requires an offset. The offset is stored in the 12-bit immediate field of the instruction, $Instr_{31:20}$. It is a signed value, so it must be sign-extended to 32 bits. Sign extension simply means copying the sign bit into the most significant bits: $ImmExt_{31:12} = Instr_{31}$, and $ImmExt_{11:0} = Instr_{31:20}$. Sign-extension is performed by



Address	Instruction	Type	Fields					Machine Language
			imm$_{11:0}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.5 Sign-extend the immediate

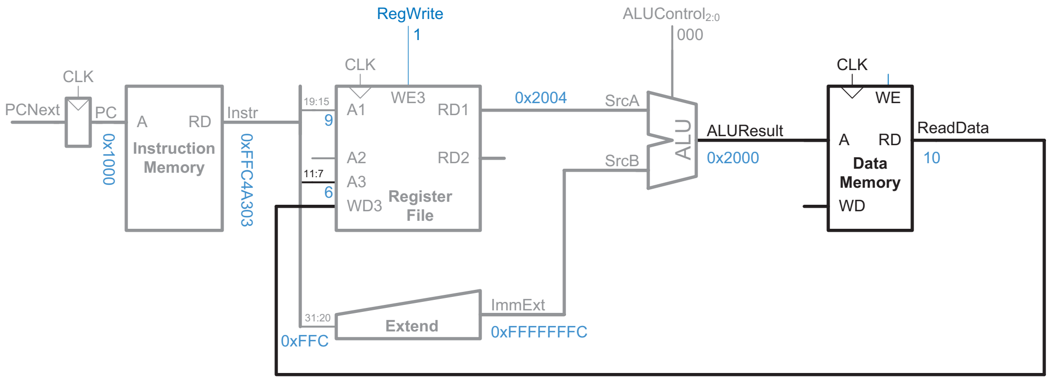


Address	Instruction	Type	Fields					Machine Language
			imm$_{11:0}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.6 Compute memory address

an Extend unit, as shown in Figure 7.5, which receives the 12-bit signed immediate in $Instr_{31:20}$ and produces the 32-bit sign-extended immediate, *ImmExt*. In our example, the two's complement immediate −4 is extended from its 12-bit representation 0xFFC to a 32-bit representation 0xFFFFFFFC.

The processor adds the base address to the offset to find the address to read from memory. Figure 7.6 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* is the base address from the register file, and *SrcB* is the offset from the sign-extended immediate, *ImmExt*. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal



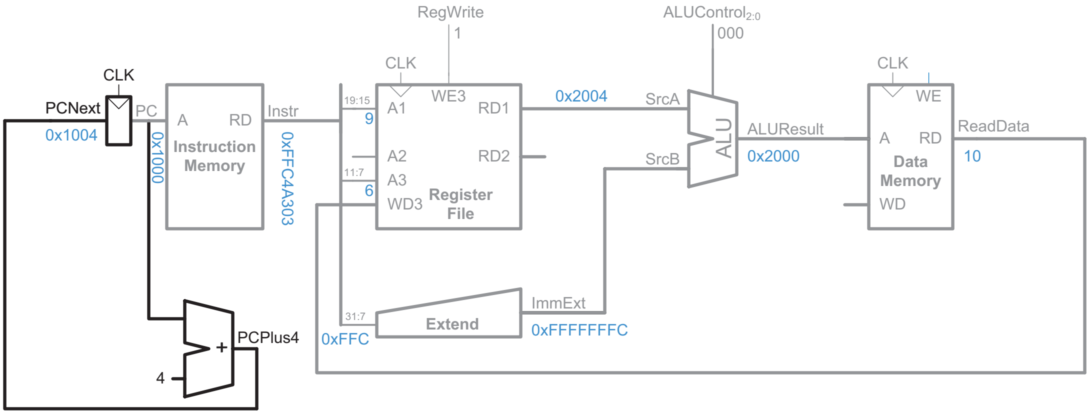
Address	Instruction	Type	Fields					Machine Language
			$\text{imm}_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.7 Read memory and write result back to register file

specifies the operation (see Table 5.3 on page 250). The ALU receives 32-bit operands and generates a 32-bit *ALUResult*. For the `lw` instruction, *ALUControl* should be set to 000 to perform addition. *ALUResult* is sent to the data memory as the address to read, as shown in Figure 7.6. In our example, the ALU computes 0x2004 + 0xFFFFFFFC = 0x2000. Again, this is a 32-bit value, but we omit the leading zeros to avoid cluttering the figure.

This memory address from the ALU is provided to the address (*A*) port of the data memory. The data is read from the data memory onto the *ReadData* bus and then written back to the destination register at the end of the cycle, as shown in Figure 7.7. Port 3 of the register file is the write port. `lw`'s destination register, indicated by the **rd** field ($Instr_{11:7}$), is connected to *A3*, port 3's address input. The *ReadData* bus is connected to *WD3*, port 3's write data input. A control signal called *RegWrite* (register write) is connected to *WE3*, port 3's write enable input, and is asserted during the `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle. In our example, the processor reads 10 from address 0x2000 in the data memory and puts that value (10) into `x6` in the register file.

While the instruction is being executed, the processor must also compute the address of the next instruction, *PCNext*. Because instructions are 32 bits (4 bytes), the next instruction is at PC+4. Figure 7.8 uses an adder to increment the PC by 4. In our example, *PCNext* = 0x1000 + 4 = 0x1004. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `lw` instruction.



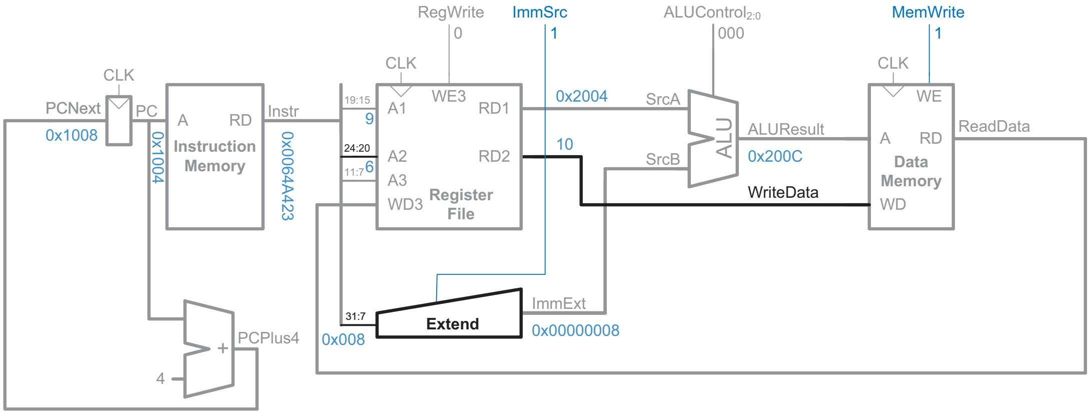
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.8 Increment program counter

sw

Next, let us extend the datapath to handle sw, which is an S-type instruction. Like lw, sw reads a base address from port 1 of the register file and sign-extends the immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported in the datapath, but the 12-bit signed immediate is stored in $Instr_{31:25,11:7}$ (instead of $Instr_{31:20}$, as it was for lw). Thus, the Extend unit must be modified to also receive these additional bits, $Instr_{11:7}$. For simplicity (and for future instructions such as jal), the Extend unit receives all the bits of $Instr_{31:7}$. A control signal, *ImmSrc*, decides which instruction bits to use as the immediate. When *ImmSrc* = 0 (for lw), the Extend unit chooses $Instr_{31:20}$ as the 12-bit signed immediate; when *ImmSrc* = 1 (for sw), it chooses $Instr_{31:25,11:7}$.

The sw instruction also reads a second register from the register file and writes its contents to the data memory. Figure 7.9 shows the new connections for this added functionality. The register is specified in the **rs2** field, $Instr_{24:20}$, which is connected to the address 2 (*A2*) input of the register file. The register's contents are read onto the read data 2 (*RD2*) output, which, in turn, is connected to the write data (*WD*) input of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For an sw instruction: *MemWrite* = 1 to write the data to memory; *ALUControl* = 000 to add the base address and offset; and *RegWrite* = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but this *ReadData* is ignored because *RegWrite* = 0.



Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

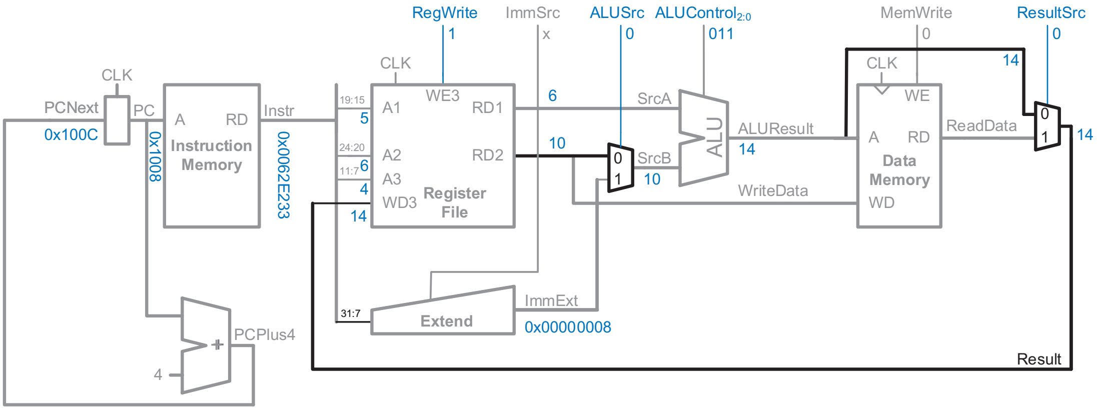
Figure 7.9 Write data to memory for sw instruction

In our example, the PC is 0x1004. Thus, the instruction memory reads out the sw instruction, 0x0064A423. The register file reads 0x2004 (the base address) from x9 and 10 from x6 while the Extend unit extends the immediate offset 8 from 12 to 32 bits. The ALU computes 0x2004 + 8 = 0x200C. The data memory writes 10 to address 0x200C. Meanwhile, the PC is incremented to 0x1008.

R-Type Instructions

Next, consider extending the datapath to handle the R-type instructions, add, sub, and, or, and slt. All of these instructions read two source registers from the register file, perform some ALU operation on them, and write the result back to the destination register. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware but with different *ALUControl* signals. Recall from Section 5.2.4 that *ALUControl* is 000 for addition, 001 for subtraction, 010 for AND, 011 for OR, and 101 for set less than.

Figure 7.10 shows the enhanced datapath handling these R-type instructions. The datapath reads **rs1** and **rs2** from ports 1 and 2 of the register file and performs an ALU operation on them. We introduce a multiplexer and a new select signal, *ALUSrc*, to select between *ImmExt* and *RD2* as the second ALU source, *SrcB*. For lw and sw, *ALUSrc* is 1 to select *ImmExt*; for R-type instructions, *ALUSrc* is 0 to select the register file output *RD2* as *SrcB*.



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

Figure 7.10 Datapath enhancements for R-type instructions

Let us name the value to be written back to the register file *Result*. For lw, *Result* comes from the *ReadData* output of the memory. However, for R-type instructions, *Result* comes from the *ALUResult* output of the ALU. We add the Result multiplexer to choose the proper *Result* based on the type of instruction. The multiplexer select signal *ResultSrc* is 0 for R-type instructions to choose *ALUResult* as *Result*; *ResultSrc* is 1 for lw to choose *ReadData*. We do not care about the value of *ResultSrc* for sw because it does not write the register file.

In our example, the PC is 0x1008. Thus, the instruction memory reads out the or instruction 0x0062E233. The register file reads source operands 6 from x5 and 10 from x6. *ALUControl* is 011, so the ALU computes 6 | 10 = 0110_2 | 1010_2 = 1110_2 = 14. The result is written back to x4. Meanwhile, the PC is incremented to 0x100C.

beq

Finally, we extend the datapath to handle the branch if equal (beq) instruction. beq compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter (PC).

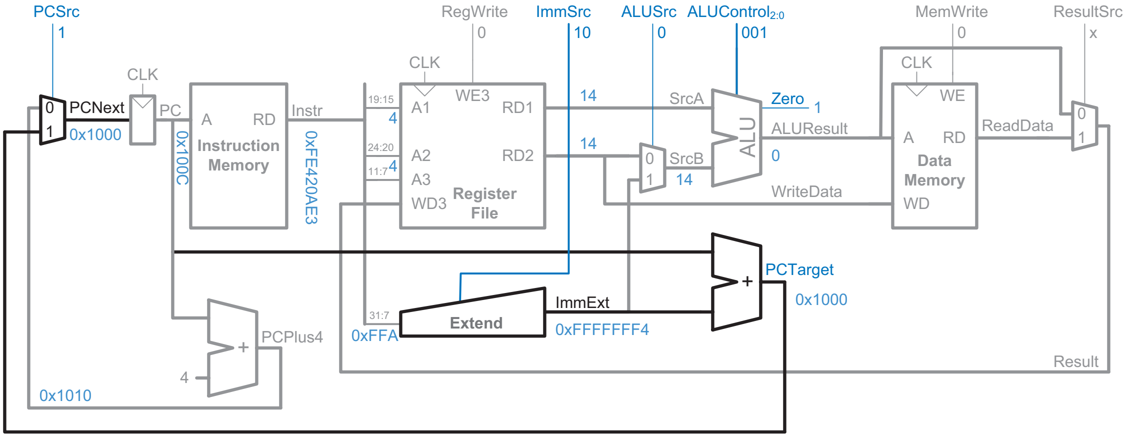
The branch offset is a 13-bit signed immediate stored in the 12-bit immediate field of the B-type instruction. Thus, the Extend logic needs yet another mode to choose the proper immediate. *ImmSrc* is increased to 2 bits, using the encoding from Table 7.1. *ImmExt* is now either the

Observe that our hardware computes all the possible answers needed by different instructions (e.g., *ALUResult* and *ReadData*) and then uses a multiplexer to choose the appropriate one based on the instruction. This is an important design strategy. Throughout the rest of this chapter, we will add multiplexers to choose the desired answer.

One of the major differences between software and hardware is that software operates sequentially, so we can compute just the answer we need. Hardware operates in parallel; therefore, we often compute all the possible answers and then pick the one we need. For example, while executing an R-type instruction with the ALU, the memory still receives an address and reads data from this address even though we don't care what that data might be.

Table 7.1 ***ImmSrc* encoding**

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1’b0}	B	13-bit signed immediate



Address	Instruction	Type	Fields						Machine Language
			$\text{imm}_{12,10:5}$	rs2	rs1	f3	$\text{imm}_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Figure 7.11 Datapath enhancements for `beq`

sign-extended immediate (when *ImmSrc* = 00 or 01) or the branch offset (when *ImmSrc* = 10).

Figure 7.11 shows the modifications to the datapath. We need another adder to compute the branch target address, *PCTarget* = *PC* + *ImmExt*. The two source registers are compared by computing (*SrcA* – *SrcB*) using the ALU. If *ALUResult* is 0, as indicated by the ALU’s *Zero* flag, the registers are equal. We add a multiplexer to choose *PCNext* from either *PCPlus4* or *PCTarget*. *PCTarget* is selected if the instruction is a branch and the *Zero* flag is asserted. For `beq`, *ALUControl* = 001, so that the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don’t care about the value of *ResultSrc*, because the register file is not written.

In our example, the PC is 0x100C, so the instruction memory reads out the `beq` instruction 0xFE420AE3. Both source registers are `x4`, so the

Logically, we can build the Extend unit from a 32-bit 3:1 multiplexer choosing one of three possible inputs based on *ImmSrc* and the various bitfields of the instruction. In practice, the upper bits of the sign-extended immediate always come from bit 31 of the instruction, $Instr_{31}$, so we can optimize the design and only use a multiplexer to select the lower bits.

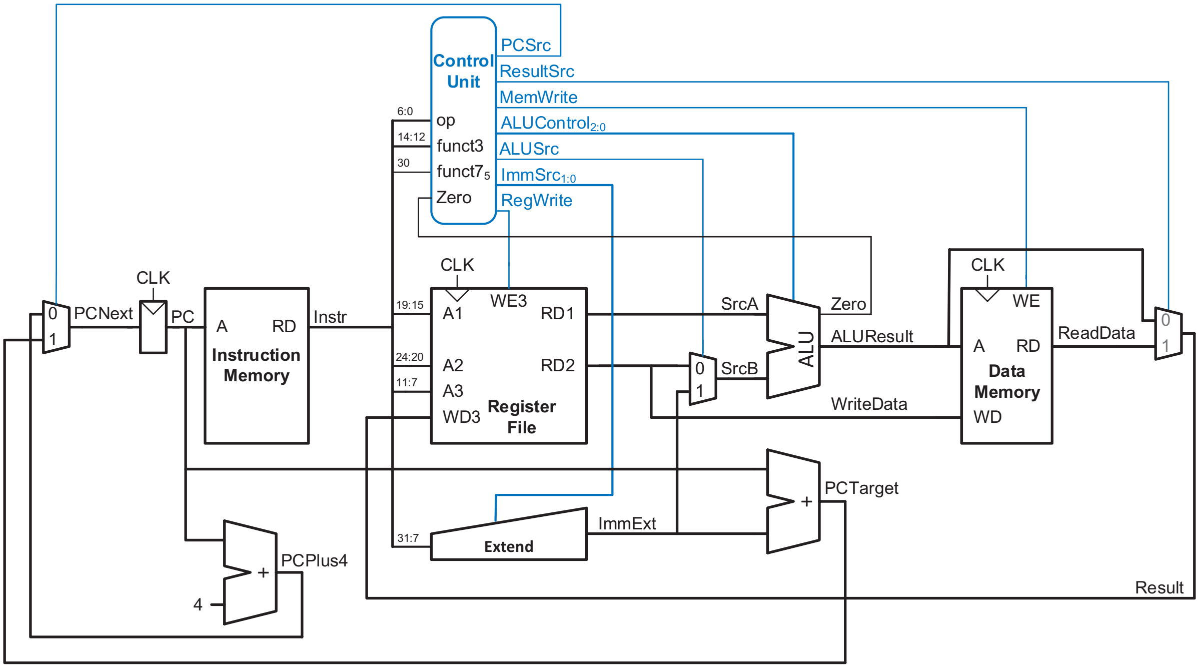


Figure 7.12 Complete single-cycle processor

register file reads 14 on both ports. The ALU computes 14 – 14 = 0, and the *Zero* flag is asserted. Meanwhile, the Extend unit produces 0xFFFFFFF4 (i.e., –12), which is added to *PC* to obtain *PCTarget* = 0x1000. Note that we show the unswizzled upper 12 bits of the 13-bit immediate on the input of the Extend unit (0xFFA). The PCNext mux chooses *PCTarget* as the next PC and branches back to the start of the code at the next clock edge.

This completes the design of the single-cycle processor datapath. We have illustrated not only the design itself but also the design process in which the state elements are identified and the combinational logic is systematically added to connect the state elements. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

We name the multiplexers (muxes) by the signals they produce. For example, the PCNext mux produces the *PCNext* signal, and the Result mux produces the *Result* signal.

7.3.3 Single-Cycle Control

The single-cycle processor's control unit computes the control signals based on **op**, **funct3**, and **funct7**. For the RV32I instruction set, only bit 5 of **funct7** is used, so we just need to consider **op** ($Instr_{6:0}$), **funct3** ($Instr_{14:12}$), and **funct7**$_5$ ($Instr_{30}$). Figure 7.12 shows the entire single-cycle processor with the control unit attached to the datapath.

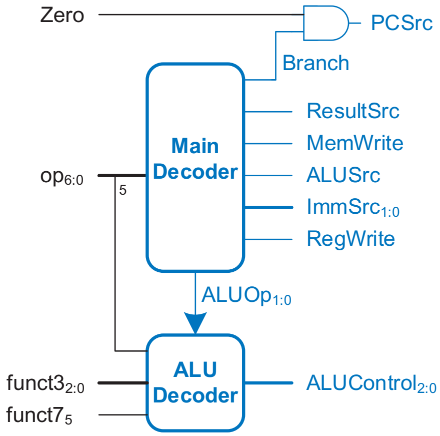


Figure 7.13 **Single-cycle processor control unit**

Table 7.2 **Main Decoder truth table**

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01

Figure 7.13 hierarchically decomposes the control unit, which is also referred to as the *controller* or the *decoder*, because it decodes what the instruction should do. We partition it into two major parts: the Main Decoder, which produces most of the control signals, and the ALU Decoder, which determines what operation the ALU performs.

Table 7.2 shows the control signals that the Main Decoder produces, as we determined while designing the datapath. The Main Decoder determines the instruction type from the opcode and then produces the appropriate control signals for the datapath. The Main Decoder generates most of the control signals for the datapath. It also produces internal signals *Branch* and *ALUOp*, signals used within the controller. The logic for the Main Decoder can be developed from the truth table using your favorite techniques for combinational logic design.

The ALU Decoder produces *ALUControl* based on *ALUOp* and **funct3**. In the case of the `sub` and `add` instructions, the ALU Decoder also uses **funct7**$_5$ and **op**$_5$ to determine *ALUControl*, as given in in Table 7.3.

Table 7.3 **ALU Decoder truth table**

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw`, `sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

ALUOp of 00 indicates add (e.g., to find the address for loads or stores). *ALUOp* of 01 indicates subtract (e.g., to compare two numbers for branches). *ALUOp* of 10 indicates an R-type ALU instruction where the ALU Decoder must look at the **funct3** field (and sometimes also the $\mathbf{op_5}$ and $\mathbf{funct7_5}$ bits) to determine which ALU operation to perform (e.g., `add`, `sub`, `and`, `or`, `slt`).

According to Table B.1 in the inside covers of the book, `add`, `sub`, and `addi` all have **funct3** = 000. `add` has **funct7** = 0000000 while `sub` has **funct7** = 0100000, so $\mathbf{funct7_5}$ is sufficient to distinguish these two. But we will soon consider supporting `addi`, which doesn't have a **funct7** field but has an **op** of 0010011. With a bit of thought, we can see that an ALU instruction with **funct3** = 000 is `sub` if $\mathbf{op_5}$ and $\mathbf{funct7_5}$ are both 1, or `add` or `addi` otherwise.

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an `and` instruction.

Solution Figure 7.14 illustrates the control signals and flow of data during execution of an `and` instruction. The PC points to the memory location holding the instruction; the instruction memory outputs this instruction. The main flow of data through the register file and ALU is represented with a heavy blue line. The register file reads the two source operands specified by *Instr*. *SrcB* should come from the second port of the register file (not *ImmExt*), so *ALUSrc* must be 0. The ALU performs a bitwise AND operation, so *ALUControl* must be 010. The result comes from the ALU, so *ResultSrc* is 0, and the result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* is 0.

The updating of *PC* with *PCPlus4* is shown by a heavy gray line. *PCSrc* is 0 to select the incremented PC. Note that data does flow through the nonhighlighted paths, but the value of that data is disregarded. For example, the immediate is extended and a value is read from memory, but these values do not influence the next state of the system.

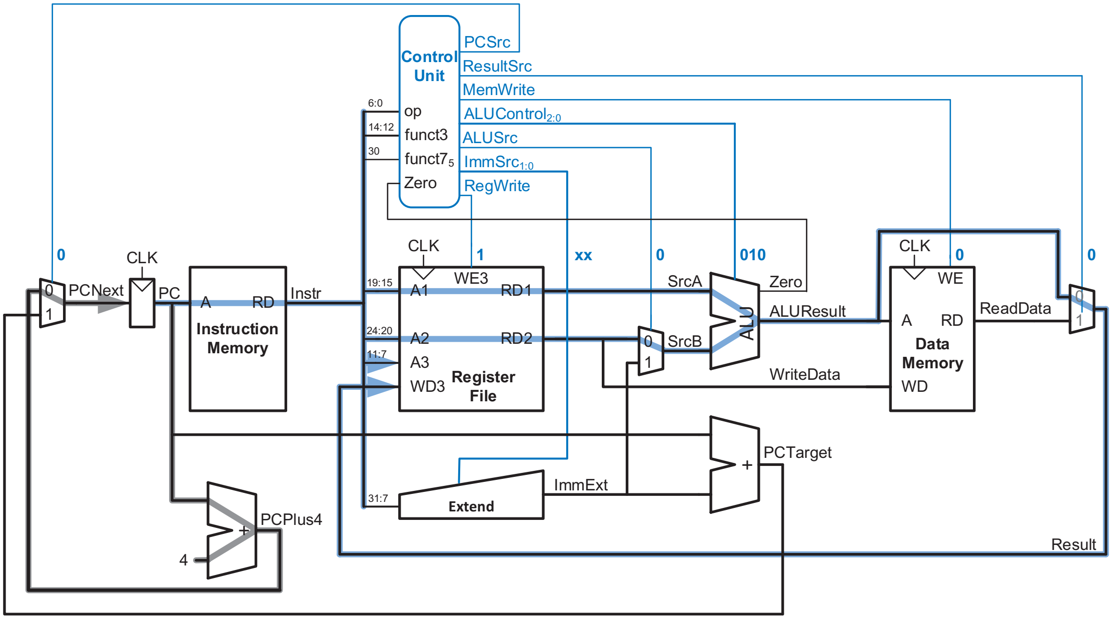


Figure 7.14 **Control signals and data flow while executing an** `and` **instruction**

7.3.4 More Instructions

So far, we have considered only a small subset of the RISC-V instruction set. In this section, we enhance the datapath and controller to support the `addi` (add immediate) and `jal` (jump and link) instructions. These examples illustrate the principle of how to handle new instructions, and they give us a sufficiently rich instruction set to write many interesting programs. With enough effort, you could extend the single-cycle processor to handle every RISC-V instruction. Moreover, we will see that supporting some instructions simply requires enhancing the decoders, whereas supporting others also requires new hardware in the datapath.

Example 7.2 `addi` INSTRUCTION

Recall that `addi rd,rs1,imm` is an I-type instruction that adds the value in **rs1** to a sign-extended immediate and writes the result to **rd**. The datapath already is capable of this task. Determine the necessary changes to the controller to support `addi`.

Solution All we need to do is add a new row to the Main Decoder truth table showing the control signal values for `addi`, as given in Table 7.4. The result should be written to the register file, so *RegWrite* = 1. The 12-bit immediate in $Instr_{31:20}$ is sign-extended as it was with `lw`, another I-type instruction, so

Table 7.4 **Main Decoder truth table enhanced to support** `addi`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01
`addi`	**0010011**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

ImmSrc is 00 (see Table 7.1). *SrcB* comes from the immediate, so *ALUSrc* = 1. The instruction does not write memory nor is it a branch, so *MemWrite* = *Branch* = 0. The result comes from the ALU, not memory, so *ResultSrc* = 0. Finally, the ALU should add, so *ALUOp* = 10; the ALU Decoder makes *ALUControl* = 000 because **funct3** = 000 and $\mathbf{op}_5$ = 0.

The astute reader may note that this change also provides the other I-type ALU instructions: `andi`, `ori`, and `slti`. These other instructions share the same **op** value of 0010011, need the same control signals, and only differ in the **funct3** field, which the ALU Decoder already uses to determine *ALUControl* and, thus, the ALU operation.

Example 7.3 `jal` INSTRUCTION

Show how to change the RISC-V single-cycle processor to support the jump and link (`jal`) instruction. `jal` writes PC+4 to **rd** and changes PC to the jump target address, PC + **imm**.

Solution The processor calculates the jump target address, the value of *PCNext*, by adding PC to the 21-bit signed immediate encoded in the instruction. The least significant bit of the immediate is always 0 and the next 20 most significant bits come from $Instr_{31:12}$. This 21-bit immediate is then sign-extended. The datapath already has hardware for adding *PC* to a sign-extended immediate, selecting this as the next PC, computing PC+4, and writing a value to the register file. Hence, in the datapath, we must only modify the Extend unit to sign-extend the 21-bit immediate and expand the Result multiplexer to choose PC+4 (i.e., *PCPlus4*) as shown in Figure 7.15. Table 7.5 shows the new encoding for *ImmSrc* to support the long immediate for `jal`.

The control unit needs to set *PCSrc* = 1 for the jump. To do this, we add an OR gate and another control signal, *Jump*, as shown in Figure 7.16. When *Jump* asserts, *PCSrc* = 1 and *PCTarget* (the jump target address) is selected as the next PC.

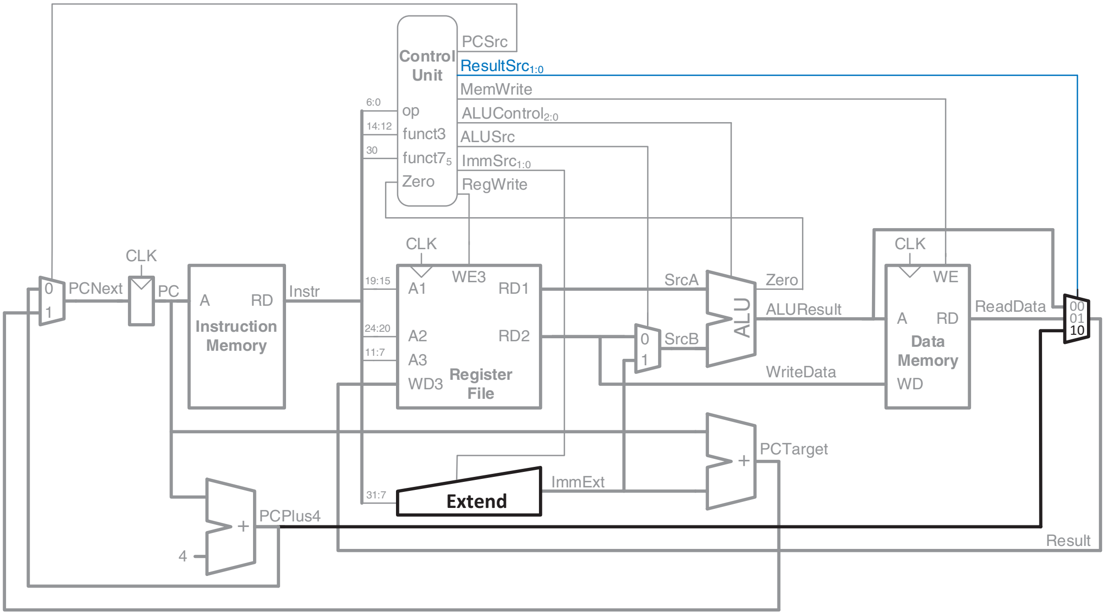


Figure 7.15 Enhanced datapath for `jal`

Table 7.5 *ImmSrc* encoding.

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

Table 7.6 shows the updated Main Decoder table with a new row for `jal`. *RegWrite* = 1 and *ResultSrc* = 10 to write PC+4 into **rd**. *ImmSrc* = 11 to select the 21-bit jump offset. *ALUSrc* and *ALUOp* don't matter because the ALU is not used. *MemWrite* = 0 because the instruction isn't a store, and *Branch* = 0 because the instruction isn't a branch. The new *Jump* signal is 1 to pick the jump target address as the next PC.

7.3.5 Performance Analysis

Recall from Equation 7.1 that the execution time of a program is the product of the number of instructions, the cycles per instruction, and

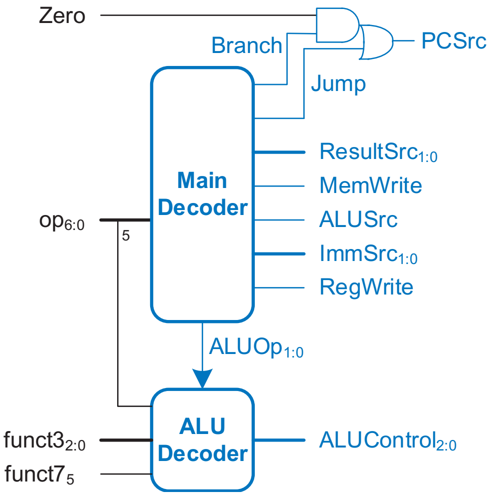


Figure 7.16 **Enhanced control unit for `jal`**

Table 7.6 **Main Decoder truth table enhanced to support `jal`**

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	01	0	00	0
`sw`	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
`beq`	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
`jal`	**1101111**	**1**	**11**	**x**	**0**	**10**	**0**	**xx**	**1**

the cycle time. Each instruction in the single-cycle processor takes one clock cycle, so the clock cycles per instruction (CPI) is 1. The cycle time is set by the critical path. In our processor, the `lw` instruction is the most time-consuming and involves the critical path shown in Figure 7.17. As indicated by heavy blue lines, the critical path starts with the PC loading a new address on the rising edge of the clock. The instruction memory then reads the new instruction, and the register file reads **rs1** as *SrcA*. While the register file is reading, the immediate field is sign-extended based on *ImmSrc* and selected at the SrcB multiplexer (path highlighted in gray). The ALU adds *SrcA* and *SrcB* to find the memory address. The data memory reads from this address, and the Result multiplexer selects *ReadData* as *Result*. Finally, *Result* must set up at the register file before

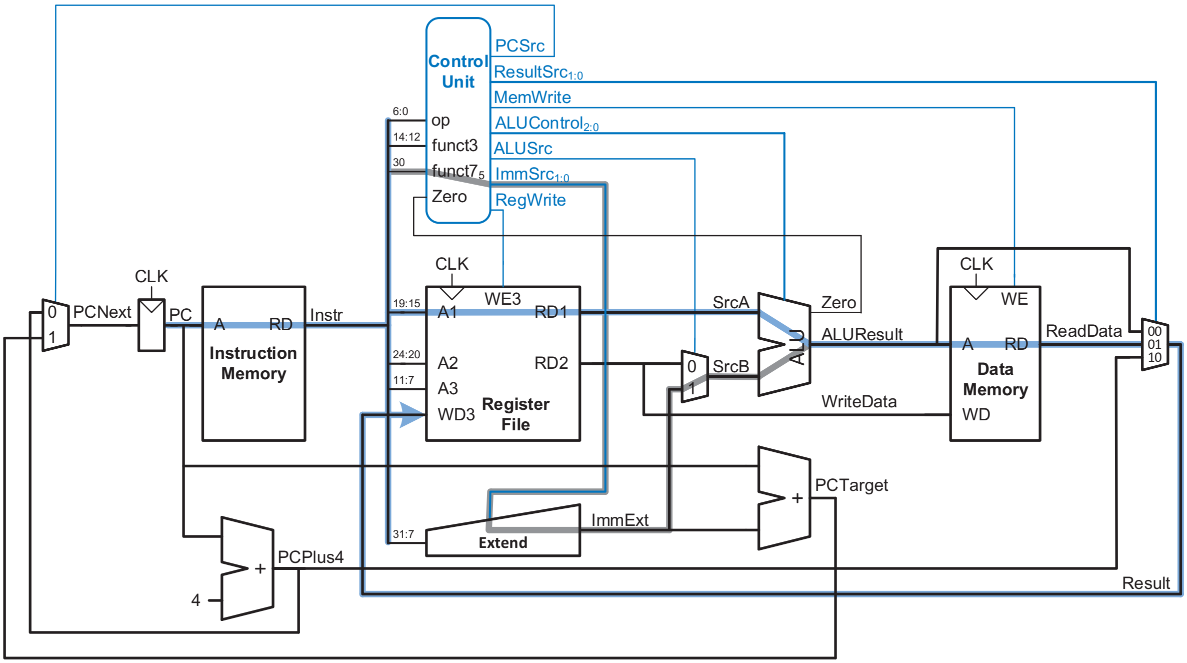


Figure 7.17 **Critical path for** `lw`

the next rising clock edge so that it can be properly written. Hence, the cycle time of the single-cycle processor is:

$$T_{c\_single} = t_{pcq\_PC} + t_{mem} + \max[t_{RFread},\ \ t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \tag{7.2}$$

Remember that `lw` does not use the second read port (A2/RD2) of the register file.

In most implementation technologies, the ALU, memory, and register file are substantially slower than other combinational blocks. Therefore, the critical path is through the register file—not through the decoder (controller), Extend unit, and multiplexer—and is the path highlighted in blue in Figure 7.17. The cycle time simplifies to:

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup} \tag{7.3}$$

The numerical values of these times will depend on the specific implementation technology.

Other instructions have shorter critical paths. For example, R-type instructions do not need to access data memory. However, we are disciplining ourselves to synchronous sequential design, so the clock period is constant and must be long enough to accommodate the slowest instruction.

Table 7.7 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle processor in a 7-nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.7. Help him compute the execution time for a program with 100 billion instructions.

Solution According to Equation 7.3, the cycle time of the single-cycle processor is $T_{c\_single} = 40 + 2(200) + 100 + 120 + 30 + 60 = 750\,\text{ps}$. According to Equation 7.1, the total execution time is $T_{\text{single}} = (100 \times 10^9 \text{ instruction}) (1 \text{ cycle/instruction}) (750 \times 10^{-12} \text{ s/cycle}) = 75$ seconds.

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three notable weaknesses. First, it requires separate memories for instructions and data, whereas most processors have only a single external memory holding both instructions and data. Second, it requires a clock cycle long enough to support the slowest instruction (`lw`) even though most instructions could be faster. Finally, it requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast.

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps. The memory, ALU, and register file have the longest delays, so to keep the delay for each short step approximately equal, the processor can use *only one* of those units in each step. The processor uses a single memory because the instruction is read in one step and data is read or written in a later step. And the processor needs only one adder, which is reused for different purposes on different steps. Various instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones.

We design a multicycle processor following the same procedure we used for the single-cycle processor. First, we construct a datapath by connecting the architectural state elements and memories with combinational logic. But, this time, we also add nonarchitectural state elements to hold intermediate results between the steps. Then, we design the controller. During the execution of a single instruction, the controller produces different signals on each step, so now the controller uses a finite state machine rather than combinational logic. Finally, we analyze the performance of the multicycle processor and compare it with the single-cycle processor.

7.4.1 Multicycle Datapath

Again, we begin our design with the memory and architectural state of the processor, as shown in Figure 7.18. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic and is feasible because we can read the instruction in one cycle, then read or write the data in another cycle. The PC and register file remain unchanged.

As with the single-cycle processor, we gradually build the datapath by adding components to handle each step of each instruction. The PC contains the address of the instruction to execute. The first step is to read this instruction from memory. Figure 7.19 shows that the PC is

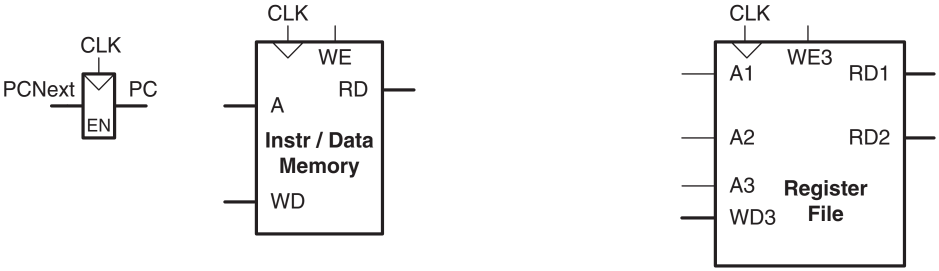


Figure 7.18 State elements with unified instruction/data memory

simply connected to the address input of the memory. The instruction is read and stored in a new nonarchitectural instruction register (IR) so that it is available for future cycles. The IR receives an enable signal, called *IRWrite*, which is asserted when the IR should be loaded with a new instruction.

Like in the single-cycle processor, we name the multiplexers and nonarchitectural registers by the signals they produce. For example, the instruction register produces the instruction signal (*Instr*), and the Result multiplexer produces the *Result* signal.

`lw`

As we did with the single-cycle processor, we first work out the datapath connections for the `lw` instruction. After fetching `lw`, the second step is to read the source register containing the base address. This register is specified in the **rs1** field, $Instr_{19:15}$. These bits of the instruction are connected to address input *A1* of the register file, as shown in Figure 7.20. The register file reads the register onto *RD1*, and this value is stored in another nonarchitectural register, *A*.

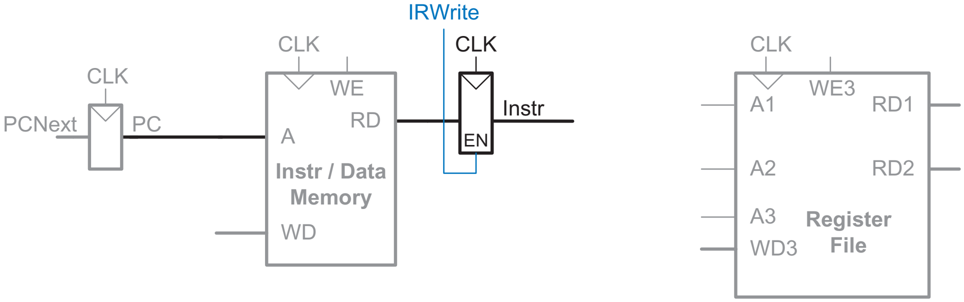


Figure 7.19 Fetch instruction from memory

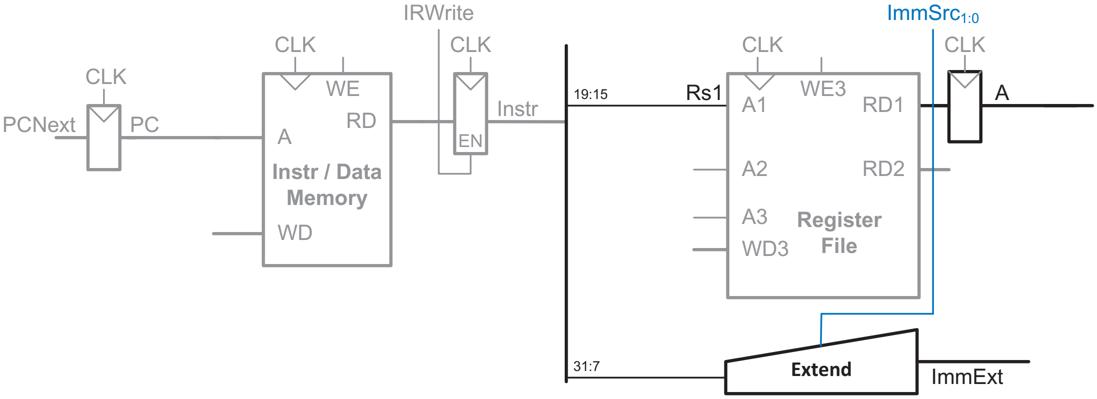


Figure 7.20 Read one source from register file and extend second source from immediate field

The `lw` instruction also requires a 12-bit offset found in the immediate field of the instruction, *Instr*$_{31:20}$, which must be sign-extended to 32 bits, as shown in Figure 7.20. As in the single-cycle processor, the Extend unit takes a 2-bit *ImmSrc* control signal to specify a 12-, 13-, or 21-bit immediate to extend for various types of instructions. The 32-bit extended immediate is called *ImmExt*. To be consistent, we might store *ImmExt* in another nonarchitectural register. However, *ImmExt* is a combinational function of *Instr* and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load is the sum of the base address and offset. In the third step, we use an ALU to compute this sum, as shown in Figure 7.21. *ALUControl* should be set to 000 to perform the addition. *ALUResult* is stored in a nonarchitectural register called ALUOut.

The fourth step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the memory address, *Adr*, from either the PC or *ALUOut* based on the *AdrSrc* select signal, as shown in Figure 7.22. The data read from memory is stored in another nonarchitectural register, called *Data*. Note that the address (Adr) multiplexer permits us to reuse the memory unit during the `lw` instruction. On the first step, the address is taken from the PC to fetch the instruction. On the fourth step, the address is taken from *ALUOut* to load the data. Hence, *AdrSrc* must have different values during different steps of a single instruction. In Section 7.4.2, we develop the FSM controller that generates these sequences of control signals.

Finally, the data is written back to the register file, as shown in Figure 7.23. The destination register is specified by the **rd** field of the instruction, *Instr*$_{11:7}$. The result comes from the Data register. Instead of connecting the Data register directly to the register file's *WD3* write port, let us add a multiplexer on the *Result* bus to choose either *ALUOut* or *Data* before feeding *Result* back to the register file's

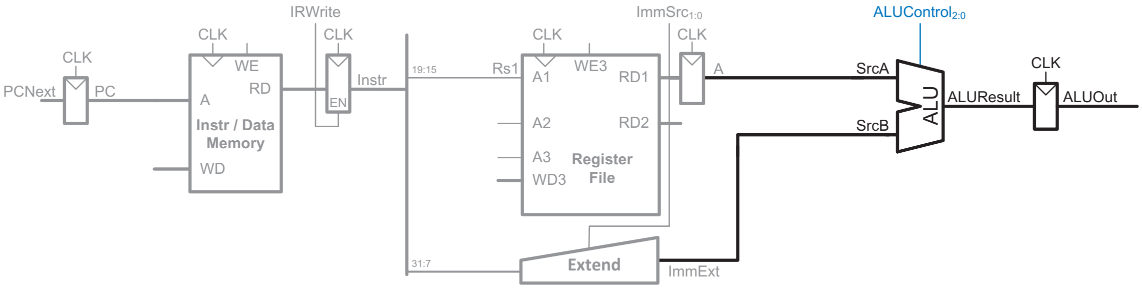


Figure 7.21 Add base address to offset

writedata port (*WD3*). This will be helpful because other instructions will need to write a result from the ALU to the register file. The *RegWrite* signal is 1 to indicate that the register file should be updated.

While all this is happening, the processor must update the program counter by adding 4 to the PC. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU during the instruction fetch step because it is not busy. To do so, we must insert source multiplexers to choose *PC* and the constant 4 as ALU inputs, as shown in Figure 7.24. A multiplexer controlled by *ALUSrcA* chooses either *PC* or *A* as *SrcA*. Another multiplexer chooses either 4 or *ImmExt* as *SrcB*. We also show additional multiplexer inputs that will be used when we implement more instructions. To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter. The Result multiplexer chooses this sum from *ALUResult* rather than *ALUOut*; this requires a third multiplexer input. The *PCWrite* control signal enables the PC to be written only on certain cycles. This completes the datapath for the `lw` instruction.

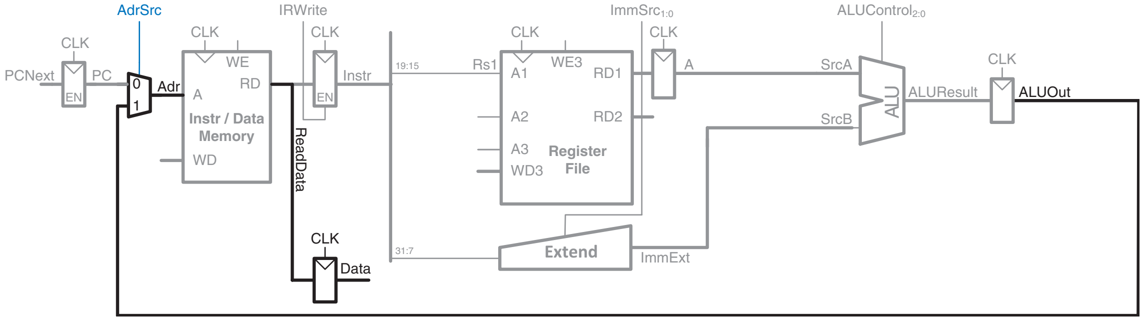


Figure 7.22 Load data from memory

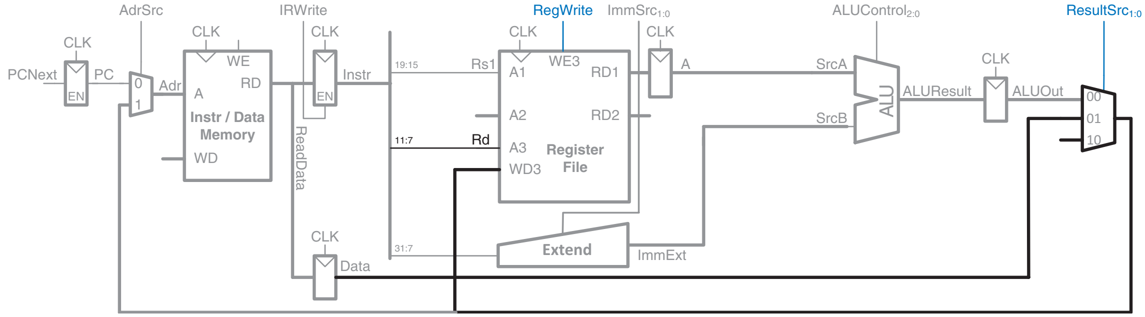


Figure 7.23 Write data back to register file

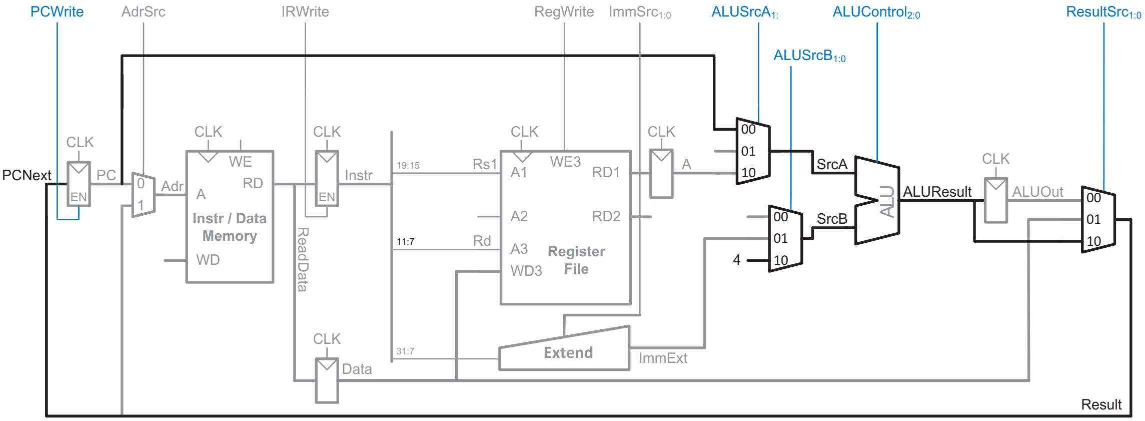


Figure 7.24 Increment PC by 4

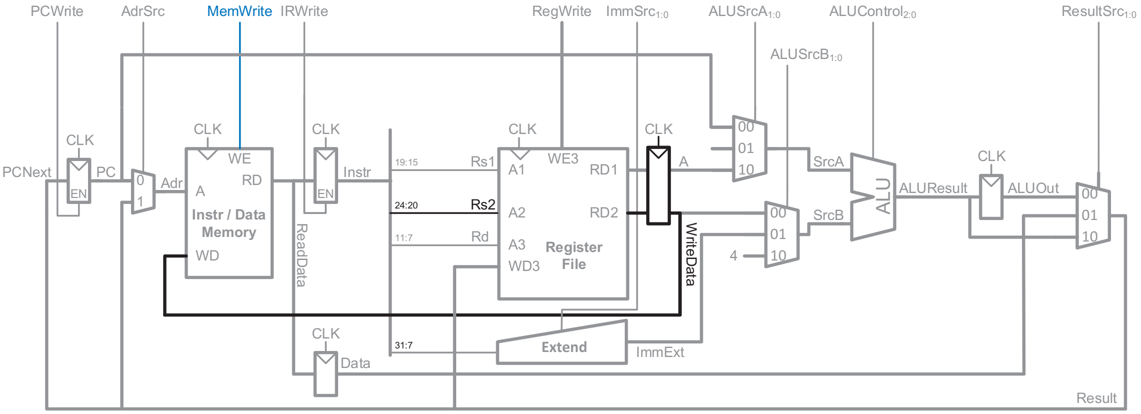


Figure 7.25 Enhanced datapath for `sw` instruction

`sw`

Next, let us extend the datapath to handle the `sw` instruction. Like `lw`, `sw` reads a base address from port 1 of the register file and extends the immediate on the second step. Then, the ALU adds the base address to the immediate to find the memory address on the third step. The only new feature of `sw` is that we must read a second register from the register file and write its contents into memory, as shown in Figure 7.25. The register is specified in the **rs2** field of the instruction, $Instr_{24:20}$, which is connected to the second port of the register file (*A2*). After it is read

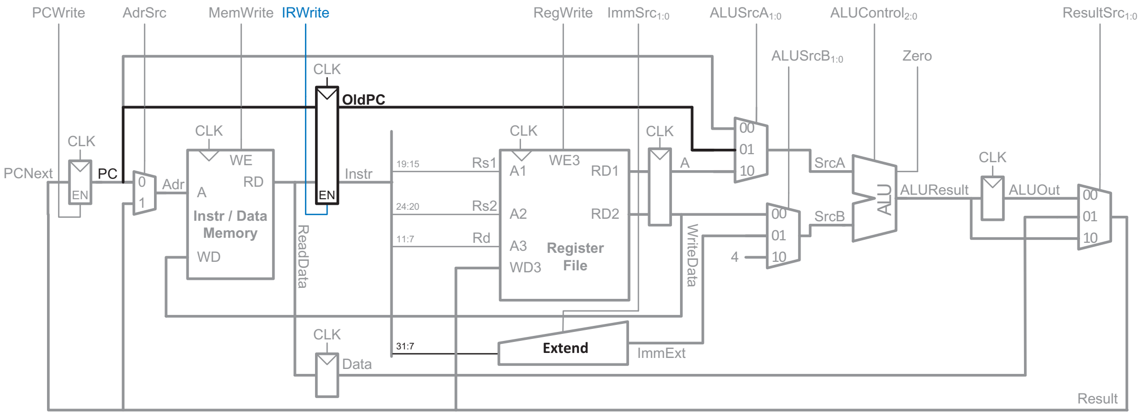


Figure 7.26 **Enhanced datapath for `beq` target address calculation**

on the second step, the register's contents are then stored in a nonarchitectural register, the WriteData register, just below the A register. It is then sent to the write data port (*WD*) of the data memory to be written on the fourth step. The memory receives the *MemWrite* control signal, which is asserted when memory should be written.

R-Type Instructions

R-type instructions operate on two source registers and write the result back to the register file. The datapath already contains all the connections necessary for these steps.

`beq`

`beq` checks whether two register contents are equal and computes the branch target address by adding the current PC to a 13-bit signed branch offset. The hardware to compare the registers using subtraction is already present in the datapath.

The ALU is not being used during the second step of instruction execution, so we use it then to calculate the branch target address *PCTarget* = *PC* + *ImmExt*. In this step, the instruction has been fetched from memory and PC has already been updated to PC+4. Thus, in the first step, the PC of the current instruction, *OldPC*, must be stored in a nonarchitectural register. In the second step, as the registers are also fetched, the ALU calculates *PC* + *ImmExt* by selecting *OldPC* for *SrcA* and *ImmExt* for *SrcB* and making *ALUControl* = 000 so that it performs addition. The processor stores this sum in the *ALUOut* register. Figure 7.26 shows the updated datapath for `beq`.

In the third step, the ALU subtracts the source registers and asserts the *Zero* output if they are equal. If they are, the control unit asserts

PCWrite and the Result multiplexer selects *ALUOut* (that contains the target address) to feed to the PC. No new hardware is needed.

This completes the design of the multicycle datapath. The design process is much like that of the single-cycle processor in that hardware is systematically connected between the state elements to handle each instruction. The main difference is that the instruction is executed in several steps. Nonarchitectural registers are inserted to hold the results of each step. In this way, the memory can be shared for instructions and data and the ALU can be reused several times, thus reducing hardware costs. In the next section, we develop an FSM controller to deliver the appropriate sequence of control signals to the datapath on each step of each instruction.

7.4.2 Multicycle Control

As in the single-cycle processor, the control unit computes the control signals based on the **op**, **funct3**, and **funct7**$_5$ fields of the instruction ($Instr_{6:0}$, $Instr_{14:12}$, and $Instr_{30}$). Figure 7.27 shows the entire multicycle processor with the control unit attached to the datapath. The datapath is shown in black and the control unit in blue.

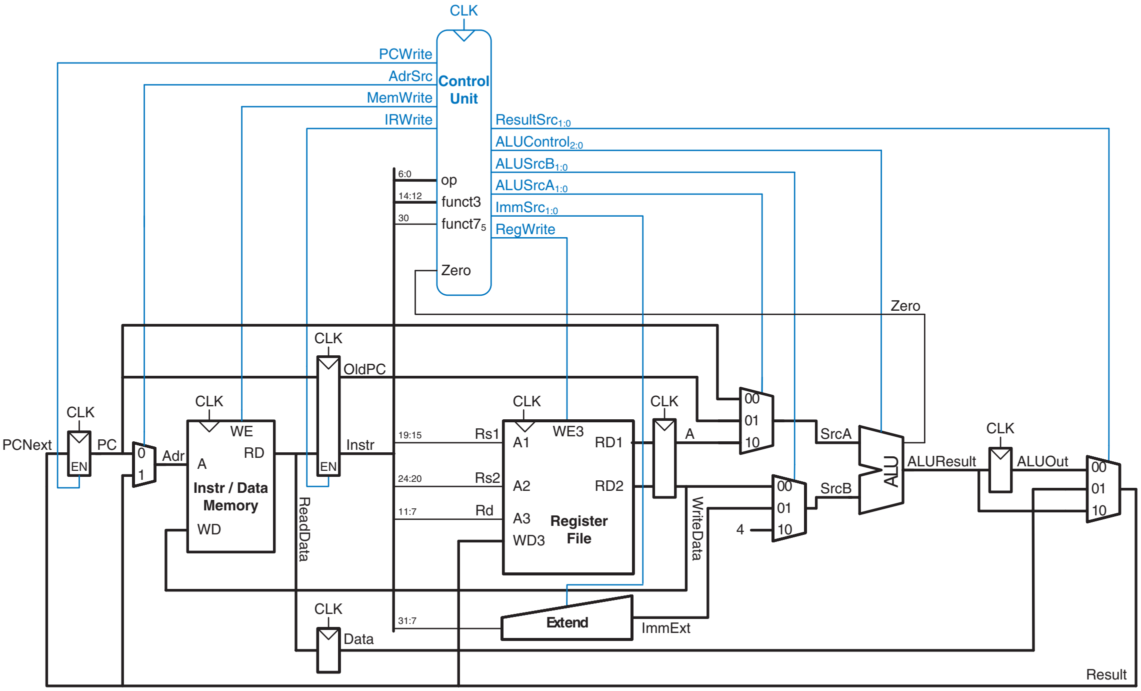


Figure 7.27 Complete multicycle processor

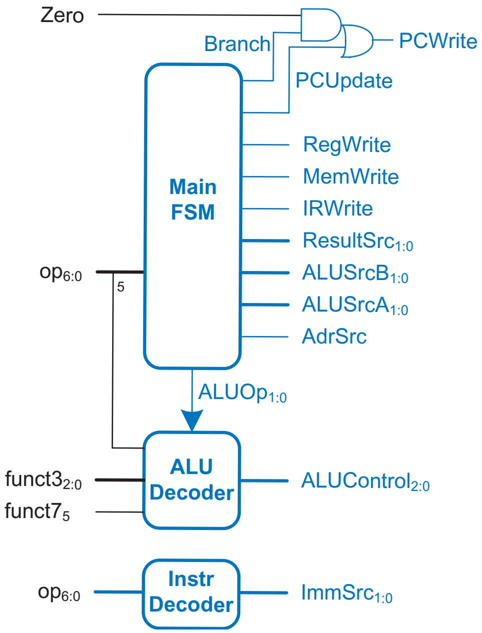


Figure 7.28 **Multicycle control unit**

The control unit consists of a Main FSM, ALU Decoder, and Instruction Decoder (Instr Decoder) as shown in Figure 7.28. The ALU Decoder is the same as in the single-cycle processor (see Table 7.3), but the combinational Main Decoder of the single-cycle processor is replaced with the Main FSM in the multicycle processor to produce a sequence of control signals on the appropriate cycles. A small Instruction Decoder combinationally produces the *ImmSrc* select signal based on the opcode using the *ImmSrc* column of Table 7.6. We design the Main FSM as a Moore machine so that the outputs are only a function of the current state. The remainder of this section develops the state transition diagram for the Main FSM.

The Main FSM produces multiplexer select, register enable, and memory write enable signals for the datapath. To keep the following state transition diagrams readable, only the relevant control signals are listed. Multiplexer select signals are listed only when their value matters; otherwise, they are don't care. Enable signals (*RegWrite*, *MemWrite*, *IRWrite*, *PCUpdate*, and *Branch*) are listed only when they are asserted; otherwise, they are 0.

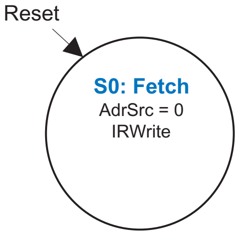


Figure 7.29 **Fetch**

Fetch

The first step for any instruction is to fetch the instruction from memory at the address held in the PC. The FSM enters this Fetch state on reset. The control signals are shown in Figure 7.29. To read the instruction from memory, *AdrSrc* = 0, so the address is taken from the PC. *IRWrite* is asserted to write the instruction into the instruction register, IR. At the same time, the current PC is written into the *OldPC* register. The data flow through the datapath for this and the next two steps of the `lw` instruction is shown in Figure 7.32, with the flow during the Fetch stage highlighted in gray.

Decode

The second step is to read the register file and decode the instructions. The control unit *decodes* the instruction, that is, figures out what operation should be performed based on **op**, **funct3**, and **funct7**$_5$. In this state, the processor also reads the source registers, **rs1** and **rs2**, and puts the values read into the A and WriteData nonarchitectural registers. No control signals are necessary for these tasks. Figure 7.30 shows the Decode state in the Main FSM and Figure 7.32 shows the flow through the datapath during this state in medium blue lines. After this step, the processor can differentiate its actions based on the instruction because the instruction has been fetched and decoded. We will first show the remaining steps for `lw`, then continue with the steps for the other RISC-V instructions.

MemAdr

The third step for `lw` is to calculate the memory address. The ALU adds the base address and the offset, so *ALUSrcA* = 10 to select *A* (the value read from **rs1**) as *SrcA*, and *ALUSrcB* = 01 to select *ImmExt* as *SrcB*. *ImmSrc*, as determined by the Instruction Decoder, is 00 to sign-extend the I-type immediate, and *ALUOp* is 00 to add *SrcA* and *SrcB*. At the end of this state, the ALU result (i.e., the address calculation) is stored in the ALUOut register. Figure 7.31 shows this MemAdr state added to the Main FSM, and Figure 7.32 shows the datapath flow during this state highlighted in dark-blue lines.

MemRead

The memory read (MemRead) step sends the calculated address to memory by sending *ALUOut* to the address port of the memory, *Adr*.

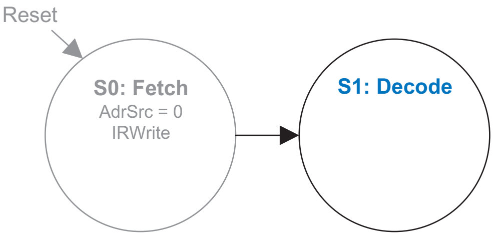


Figure 7.30 Decode

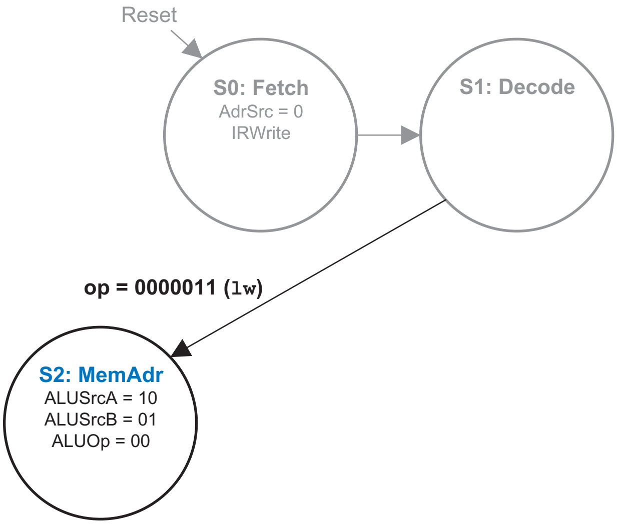


Figure 7.31 Memory address computation

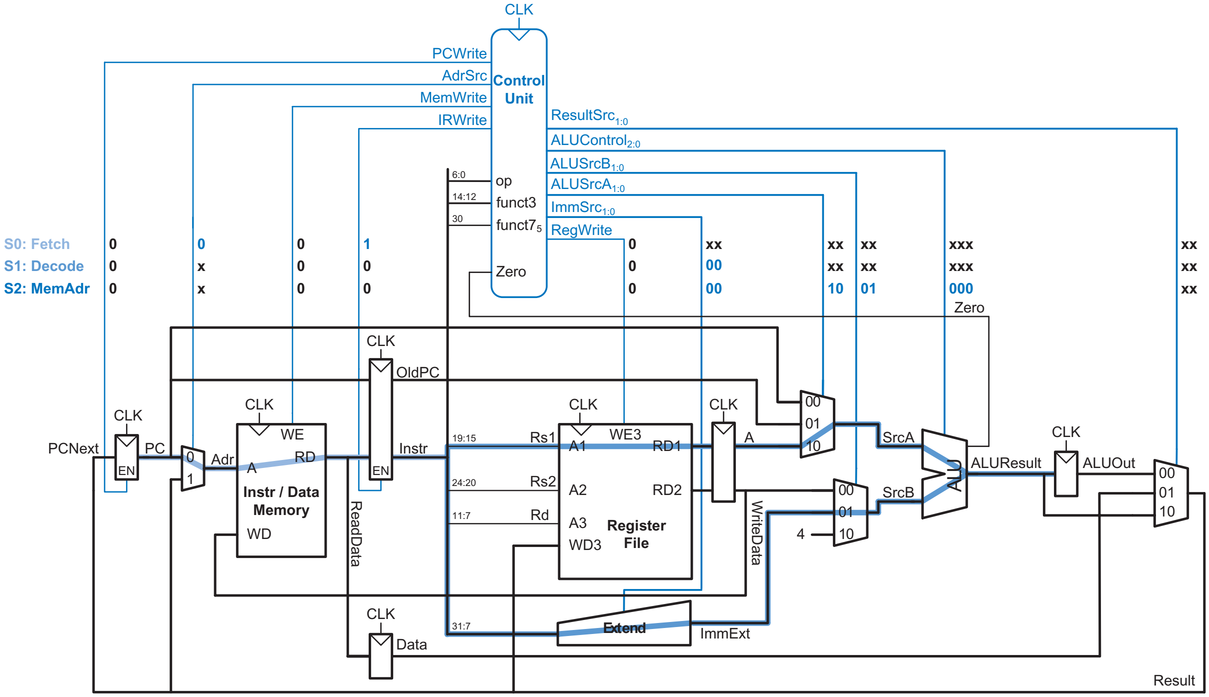


Figure 7.32 Data flow during Fetch, Decode, and MemAdr states

ResultSrc = 00 and *AdrSrc* = 1 to route *ALUOut* through the Result and Adr multiplexers to the memory address input. *ReadData* gets the value read from the desired address in memory. At the end of this state, *ReadData* is written into the Data register.

MemWB

In the memory writeback (MemWB) step, the data read from memory, *Data*, is written to the destination register. *ResultSrc* is 01 to select *Data* as the *Result*, and *RegWrite* is asserted to write the data to the register file. The register file's address and write data inputs for port 3 (*A3* and *WD3*) are already connected to **rd** ($Instr_{11:7}$) and *Result*, respectively. Figures 7.33 and 7.34 show the MemRead and MemWB states and the flow through the datapath for both steps. MemWB is the final step in the `lw` instruction. Figure 7.33 also shows the transition from MemWB back

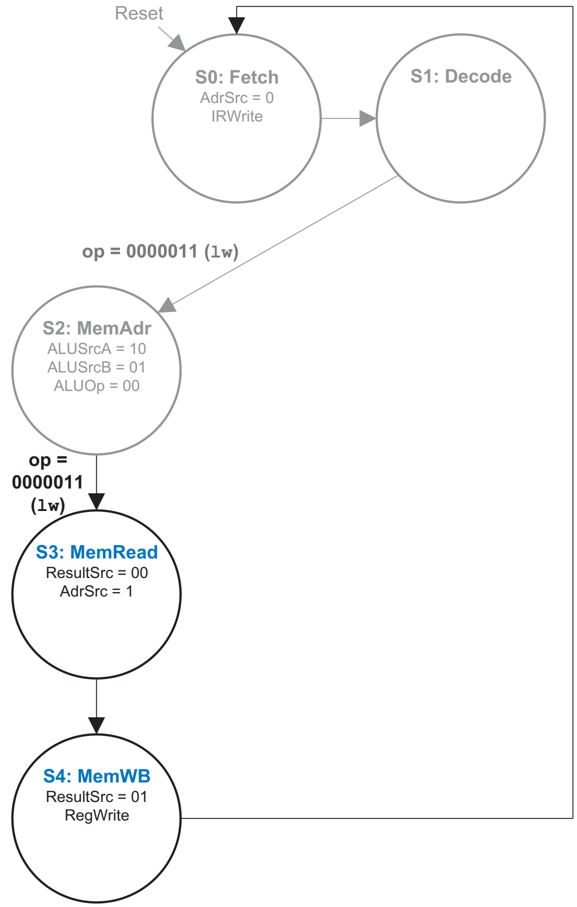


Figure 7.33 Memory read (MemRead) and memory write back (MemWB) states

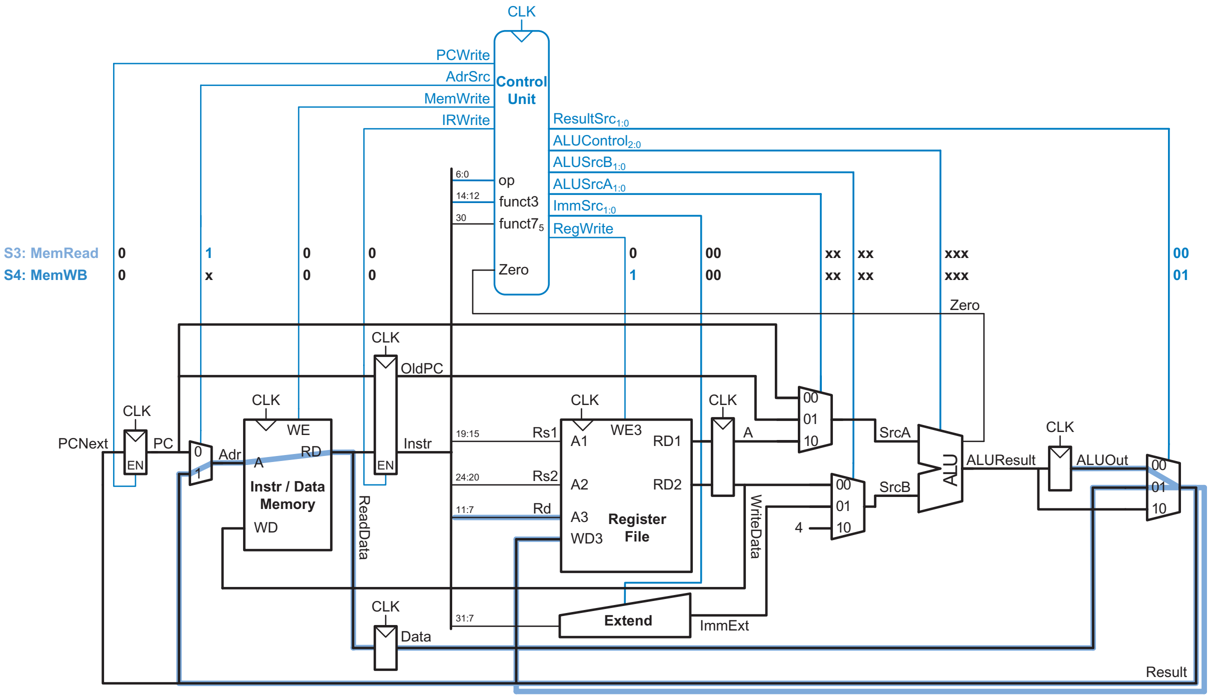


Figure 7.34 **Data flow during MemRead and MemWB**

to the Fetch state so that the next instruction can be fetched. However, the PC has not yet been incremented. We address this issue next.

Before finishing the `lw` instruction, the processor must increment the PC so that it can fetch the next instruction. We could add another state for doing this but, instead, we save a cycle by noticing that the ALU is not being used in the Fetch step, so the processor can use that state to calculate PC+4 at the same time that it fetches the instruction. *ALUSrcA* = 00 to get *SrcA* from the PC (i.e., from signal *OldPC*). *ALUSrcB* = 10 to get the constant 4 for *SrcB*. *ALUOp* = 00, so that the ALU adds PC to 4. To update the PC with PC+4, *ResultSrc* = 10 to choose *ALUResult* as the *Result*, and *PCUpdate* = 1 to force *PCWrite* high (see Figure 7.28). Figure 7.35 shows the modified Fetch state. The rest of the diagram remains the same as in Figure 7.33. Figure 7.36 highlights in blue the data flow for computing PC+4. The instruction fetch, which is occurring simultaneously, is highlighted in gray.

We started this section stating that only one of the time-consuming units (the memory, ALU, or register file) could be used in each step. However, here, we use *both* the register file and ALU. As long as the units are used at the same time—that is, in parallel—more than one unit can be used in a single step.

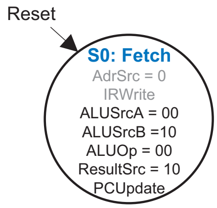


Figure 7.35 **Incrementing PC in the Fetch state**

`sw`

Now, we expand the Main FSM to handle more RISC-V instructions. All instructions pass through the first two states, Fetch and Decode.

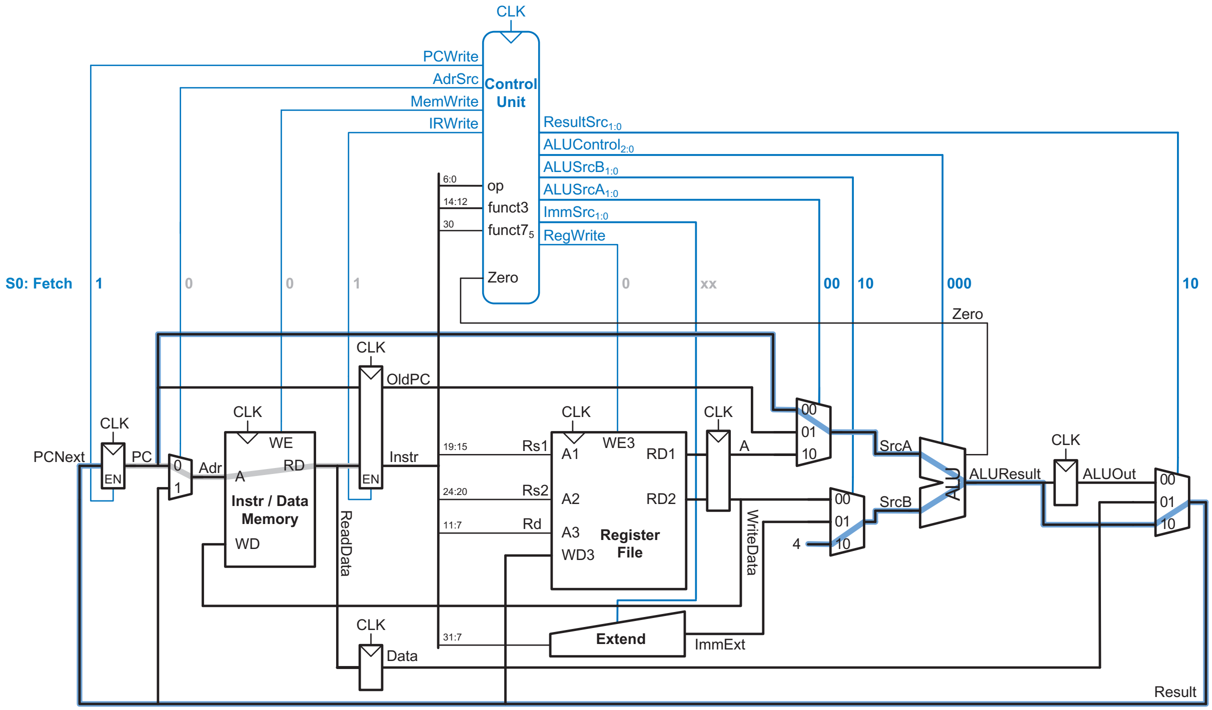


Figure 7.36 Data flow while incrementing PC in the Fetch state

The *ImmSrc* signal differs for `lw` and `sw` in the MemAdr state. But recall that *ImmSrc* is produced combinationally by the Instruction Decoder (see Figure 7.28).

The `sw` instruction uses the same MemAdr state as `lw` to calculate the memory address but then proceeds to the memory write (MemWrite) state, where *WriteData*, the value from **rs2**, is written to memory. *WriteData* is hardwired to the memory's write data port (*WD*). The memory's address port, *Adr*, is set to the calculated address in *ALUOut* by making *ResultSrc* = 00 and *AdrSrc* = 1. *MemWrite* is asserted to write the memory. This completes the `sw` instruction, so the Main FSM returns to the Fetch state to begin the next instruction. Figures 7.37 and 7.38 show the expanded Main FSM and the datapath flow for the MemWrite state. Note that the first two states of the FSM (Fetch and Decode), which are not shown in Figure 7.37, are the same as in Figure 7.33.

R-Type Instructions

After the Decode state, R-type ALU instructions proceed to the execute (ExecuteR) state, which performs the desired ALU computation. Namely, *ALUSrcA* = 10 and *ALUSrcB* = 00 to choose the contents of **rs1** as *SrcA* and **rs2** as *SrcB*. *ALUOp* = 10 so that the ALU Decoder uses the instruction's control fields to determine what operation to perform.

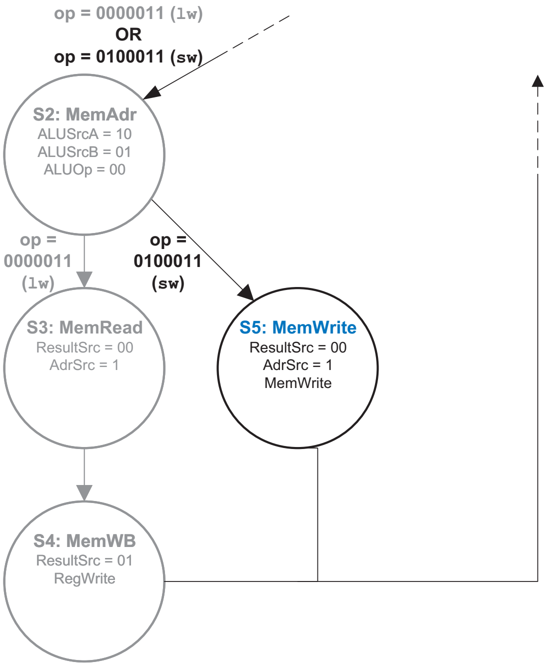


Figure 7.37 Memory write

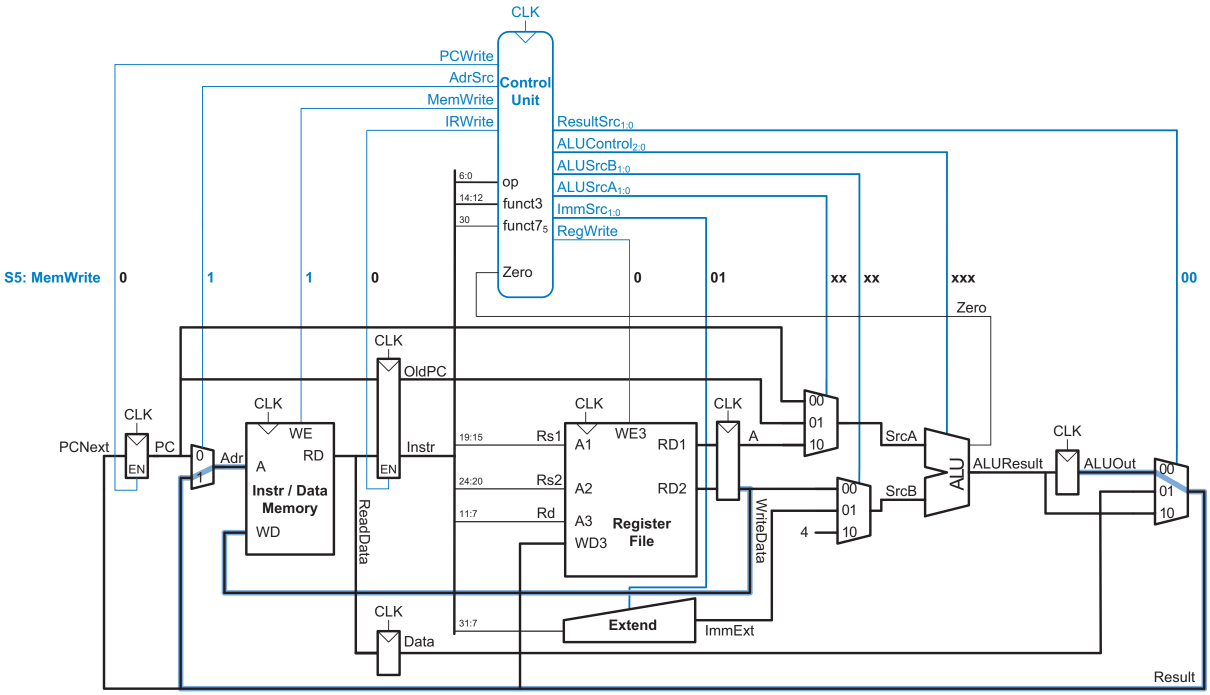


Figure 7.38 Data flow during the memory write (MemWrite) state

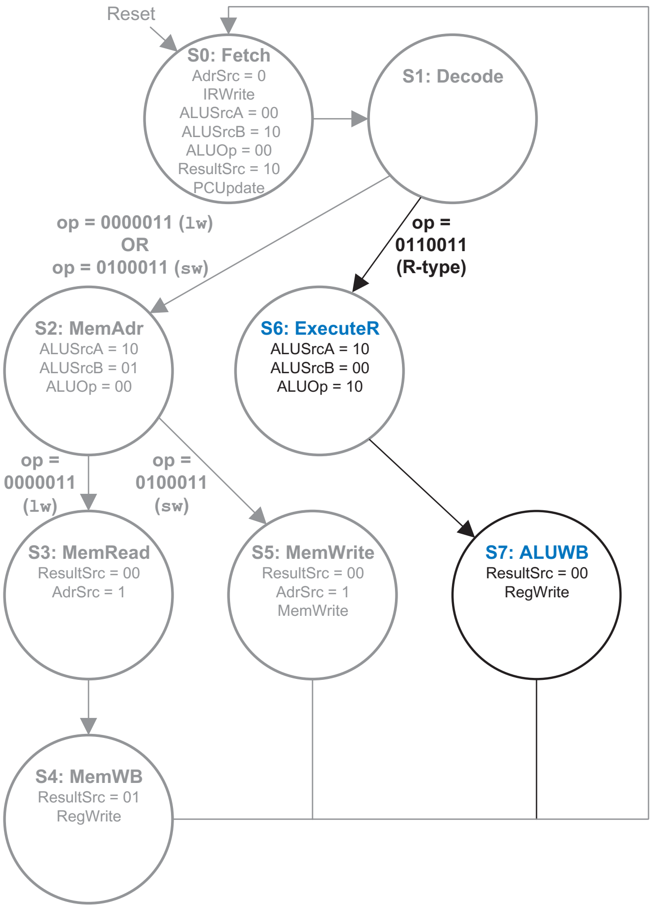


Figure 7.39 Execute R-type (ExecuteR) and ALU writeback (ALUWB) states

ALUResult is written to the ALUOut register at the end of the cycle. R-type instructions then go to the ALU writeback (ALUWB) state where the computation result, *ALUOut*, is written back to the register file. In the ALUWB state, *ResultSrc* = 00 to select *ALUOut* as *Result*, and *RegWrite* = 1 so that **rd** is written with the result. Figure 7.39 shows the ExecuteR and ALUWB states added to the Main FSM. Figure 7.40 shows the data flow during both states, with ExecuteR data flow shown in thick light-blue lines and ALUWB data flow in thick dark-blue lines.

beq

The final instruction, beq, compares two registers and computes the branch target address. Thus far, the ALU is idle during the Decode state, so we can use the ALU during that state to calculate the branch target

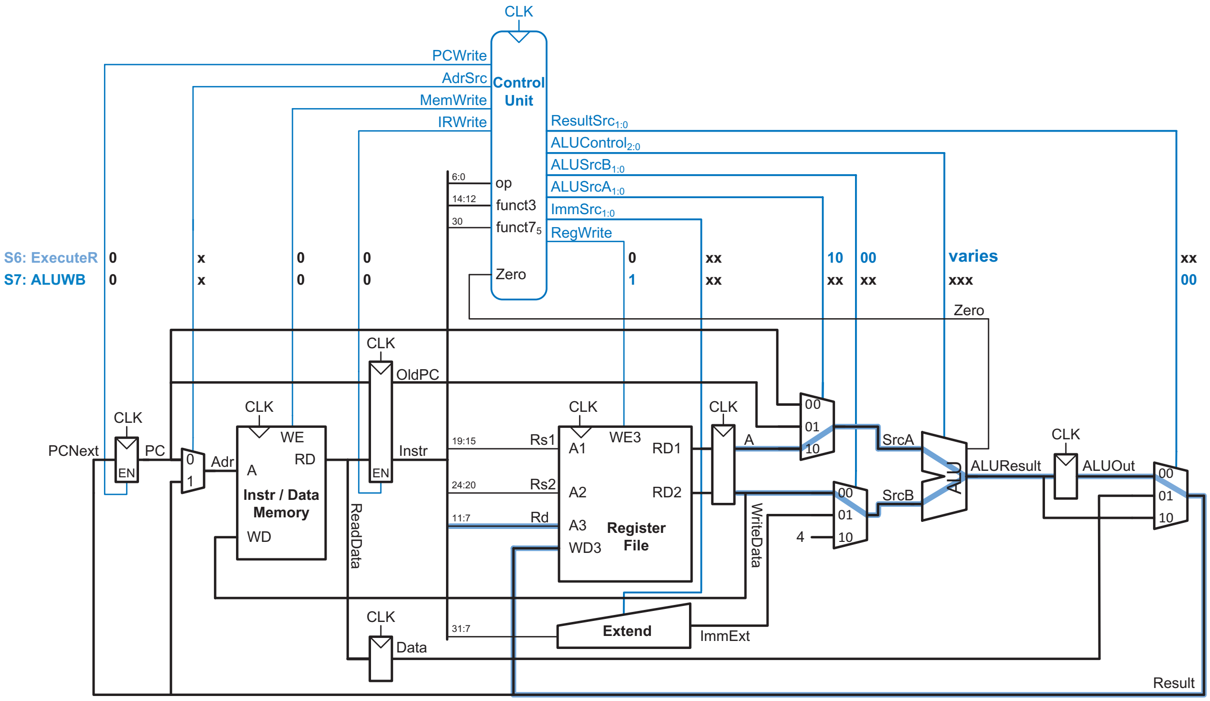


Figure 7.40 Data flow during the ExecuteR and ALUWB states

address, *OldPC* + *ImmExt*. *ALUSrcA* and *ALUSrcB* are both 01 so that *OldPC* is *SrcA* and the branch offset (*ImmExt*) is *SrcB*. *ALUOp* = 00 to make the ALU add. The target address is stored in the ALUOut register at the end of the Decode state. Figure 7.41 shows the enhanced Decode state as well as the subsequent BEQ state, which is discussed next. In Figure 7.42, the data flow during the Decode state is shown in light-blue and gray lines. The branch target address calculation is highlighted in light blue, and the register read and immediate extension is highlighted with thick gray lines.

After the Decode state, `beq` proceeds to the BEQ state, where it compares the source registers. *ALUSrcA* = 10 and *ALUSrcB* = 00 to select the values read from the register file as *SrcA* and *SrcB*. *ALUOp* = 01 so that the ALU performs subtraction. If the source registers are equal, the ALU's *Zero* output asserts (because **rs1** − **rs2** = 0). *Branch* = 1 in this state so that if *Zero* is also set, *PCWrite* is asserted (as shown in the *PCWrite* logic of Figure 7.28) and the branch target address (in *ALUOut*) becomes the next PC. *ALUOut* is routed to the PC register by *ResultSrc* being 00. Figure 7.41 shows the BEQ state, and Figure 7.42

Even though the instruction is not yet decoded at the beginning of the Decode state—and it may not even be a `beq` instruction—the branch target address is calculated as if it were a branch. If it turns out that the instruction is not a branch or if the branch is not taken, the resulting calculation is simply not used.

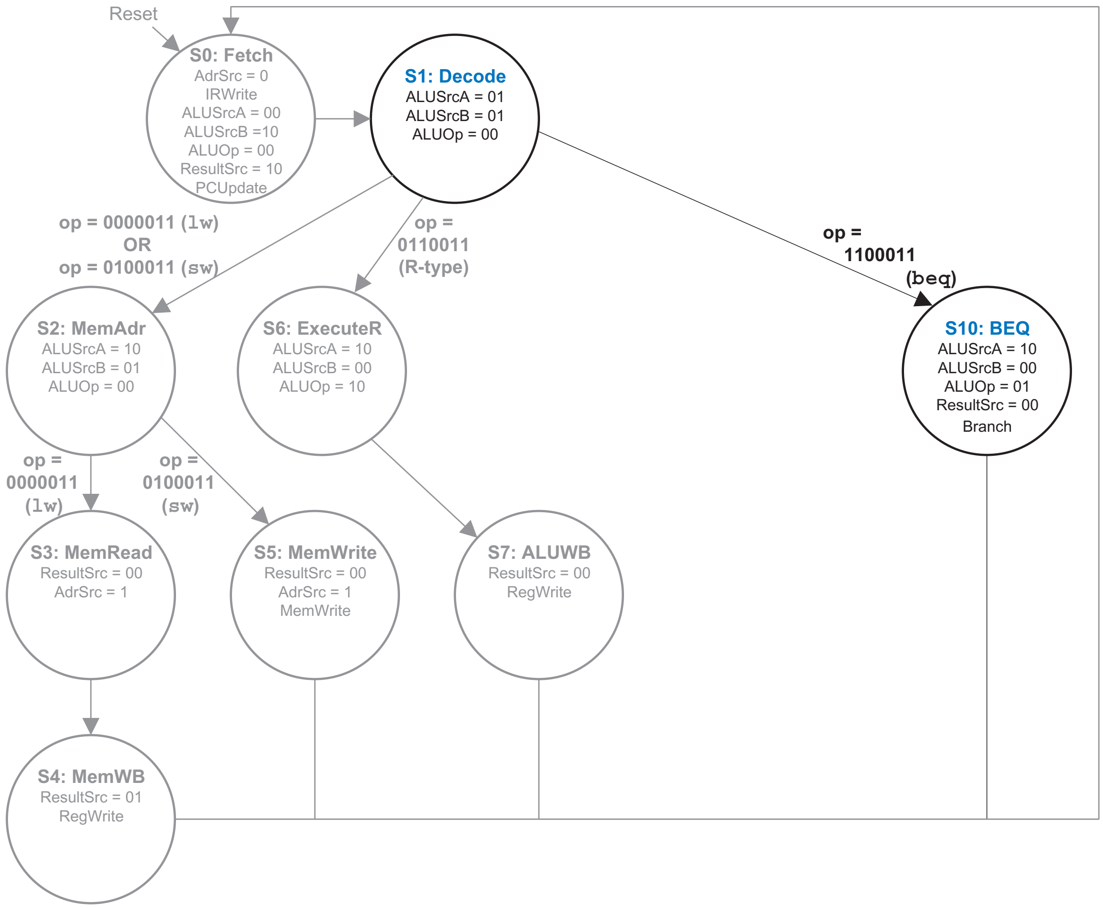


Figure 7.41 Enhanced Decode state, with branch target address calculation, and BEQ state

shows the data flow during the BEQ state. The path for comparing **rs1** and **rs2** is shown in dark blue and the path to (conditionally) set PC to the target address is in gray through the Result register. This concludes the design of the controller for these instructions.

7.4.3 More Instructions

As we did with the single-cycle processor, we next consider examples of how to modify the multicycle processor datapath and controller to handle new instructions: I-type ALU instructions (`addi`, `andi`, `ori`, `slti`) and `jal`.

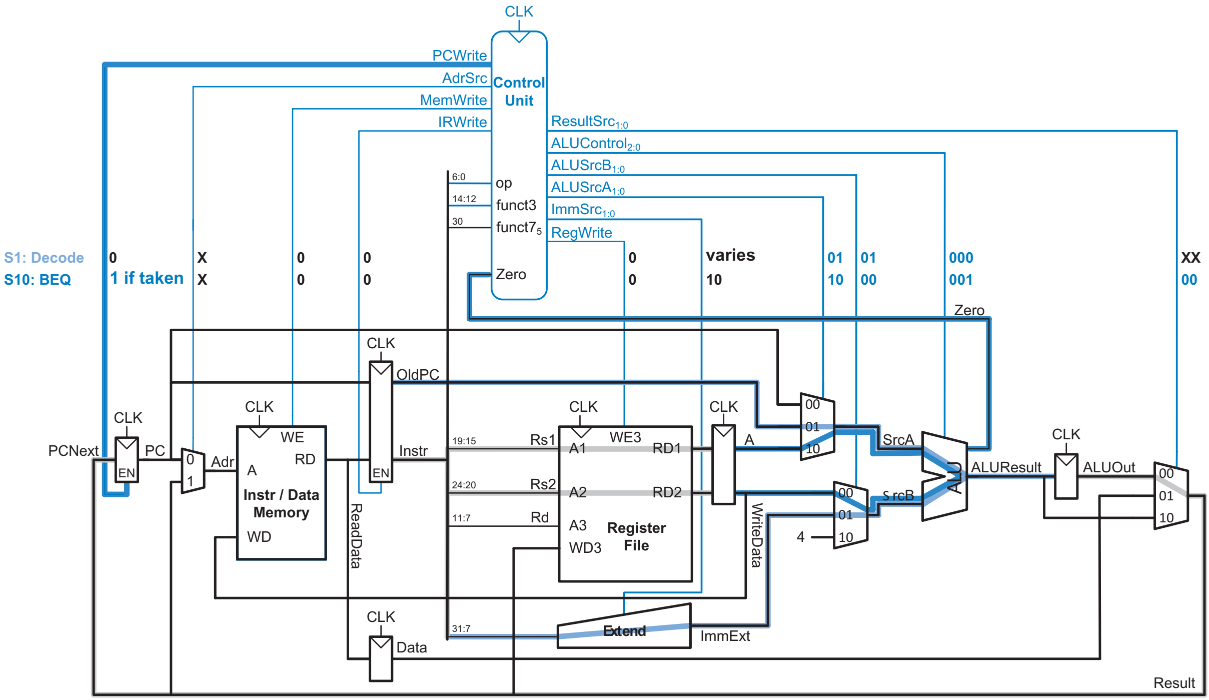


Figure 7.42 Data flow during Decode and BEQ states

Example 7.5 EXPANDING THE MULTICYCLE PROCESSOR TO INCLUDE I-TYPE ALU INSTRUCTIONS

Expand the multicycle processor to include I-type ALU instructions `addi`, `andi`, `ori`, and `slti`.

Solution These I-type ALU instructions are nearly the same as their R-type equivalents (`add`, `and`, `or`, and `slt`) except that the second source comes from *ImmExt* rather than the register file. We introduce the ExecuteI state to perform the desired computation for all I-type ALU instructions. This state is like ExecuteR except that *ALUSrcB* = 01 to choose *ImmExt* as *SrcB*. After the ExecuteI state, I-type ALU instructions proceed to the ALU writeback (ALUWB) state to write the result to the register file. Figure 7.43 shows the enhanced Main FSM, which also includes the JAL state for Example 7.6.

Example 7.6 EXPANDING THE MULTICYCLE PROCESSOR TO INCLUDE `jal`

Expand the multicycle processor to include the jump and link instruction (`jal`).

Solution Like the I-type ALU instructions from Example 7.5, no additional hardware is needed to implement the `jal` instruction. Only the Main FSM needs to be

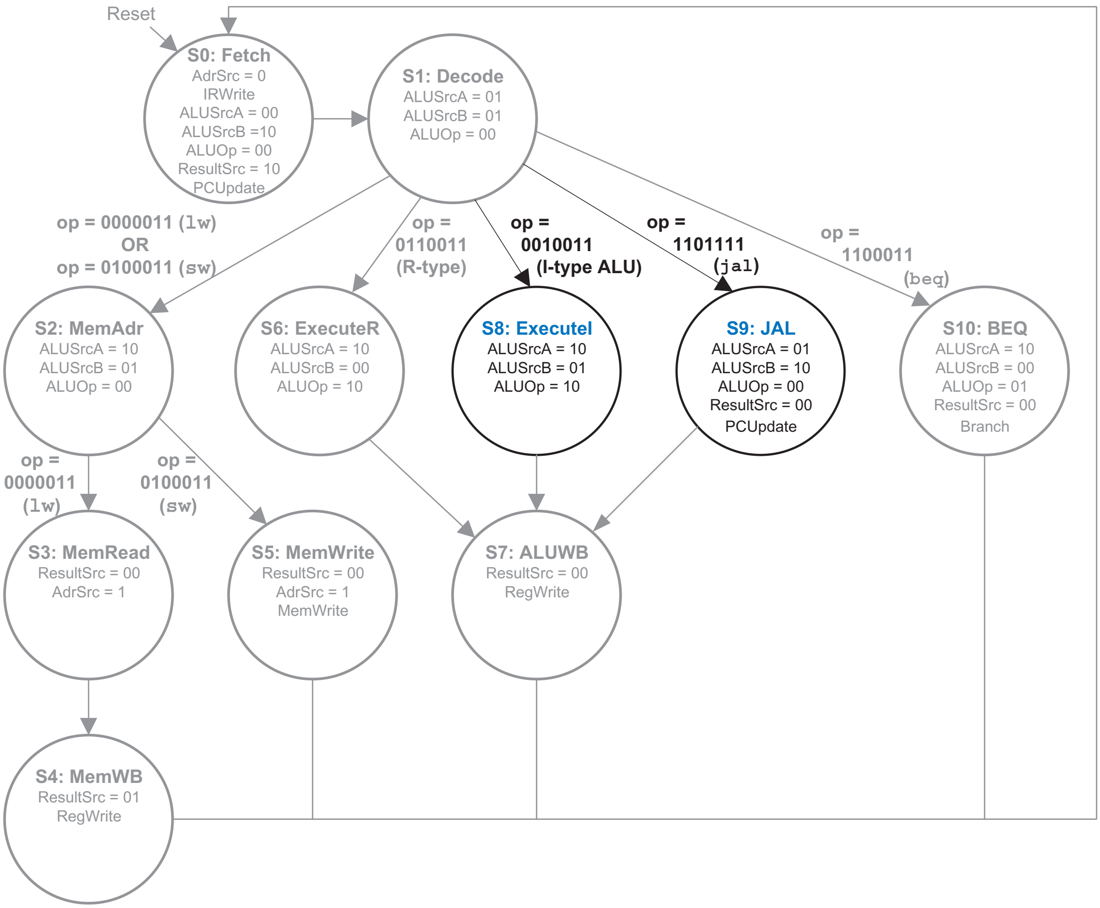


Figure 7.43 Enhanced Main FSM: ExecuteI and JAL states

You may notice that by the time the processor reaches the JAL state, the PC register has already been updated to PC+4. So we could have just used the PC register output to write to **rd**. But that would have required us to extend the Result multiplexer to receive *PC* as an input. The solution above requires less hardware because it uses the existing datapath.

updated. The first two steps are the same as the other instructions. During the Decode state, the jump target address is calculated using the same flow as the branch target address calculation but with *ImmSrc* = 11, as set by the Instruction Decoder. Thus, during the Decode state, the jump offset is sign-extended and added to the current PC (contained in signal *OldPC*) to form the jump target address, which is written to the ALUOut register at the end of that state. `jal` then proceeds to the JAL state, where the processor writes the target address to PC and calculates the return address (PC+4) so that it can write it to **rd** in the next state. The ALU calculates PC+4 (i.e., *OldPC*+4) using *ALUSrcA* = 01 (*SrcA* = *OldPC*), *ALUSrcB* = 10 (*SrcB* = 4), and *ALUOp* = 00 for addition. To write the target address to the PC, *ResultSrc* = 00 to select the target address (held in *ALUOut*) as *Result*, and *PCUpdate* = 1 so that *PCWrite* is asserted. Figure 7.43

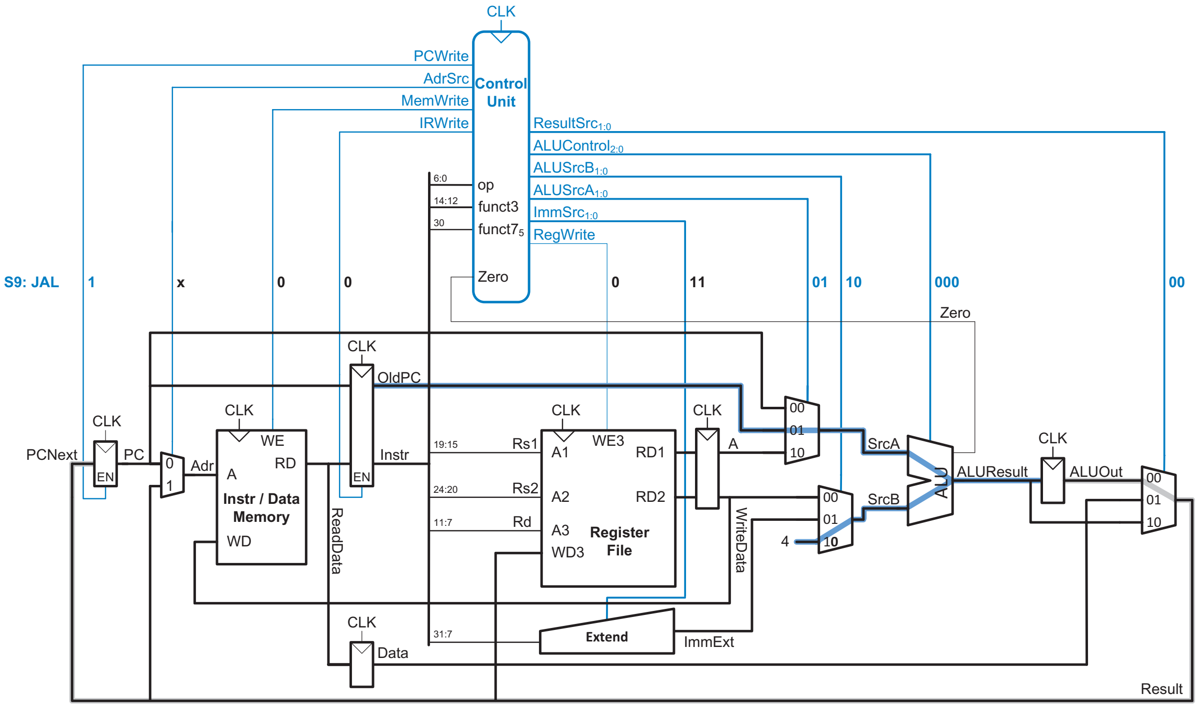


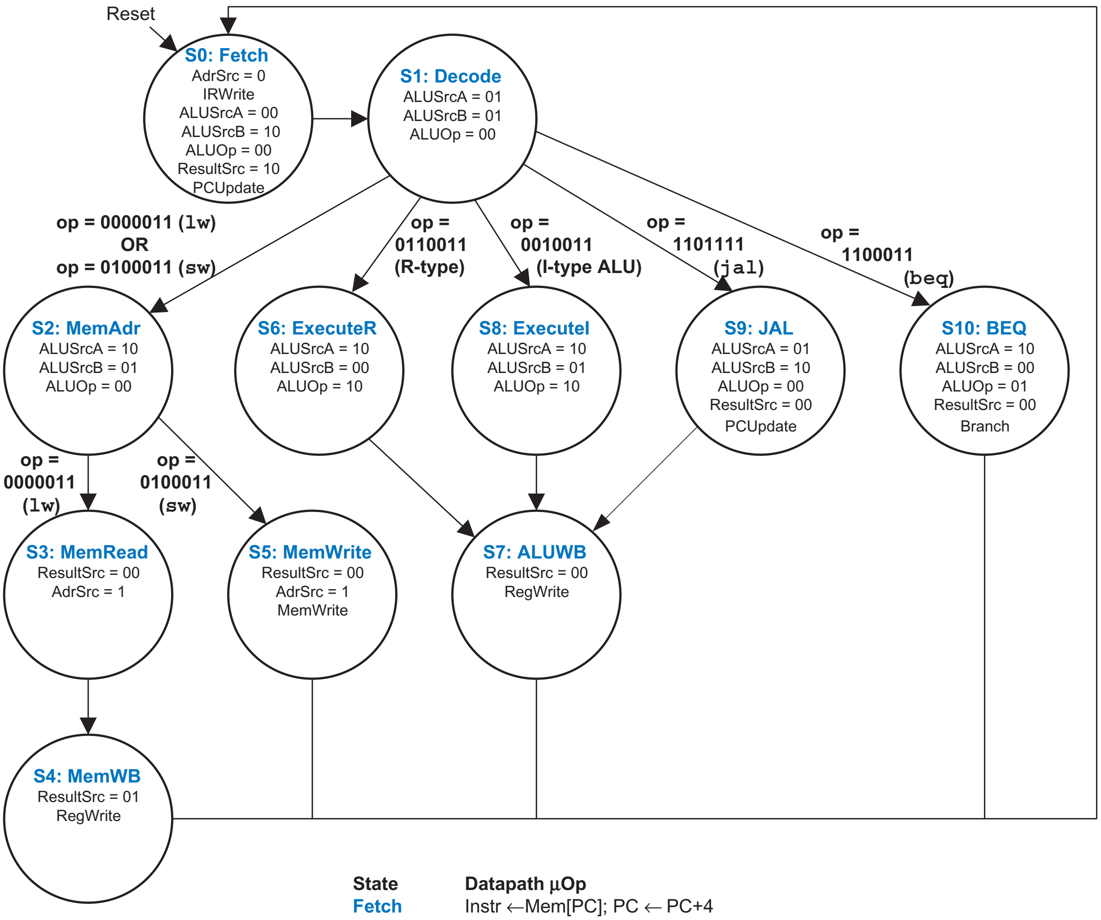
Figure 7.44 Data flow during the JAL state

shows the new JAL state, and Figure 7.44 shows the data flow during the JAL state. The flow for updating the PC to the target address is in gray and the PC+4 calculation is in blue. After the JAL state, `jal` proceeds to the ALUWB state, where the return address (ALUOut = PC+4) is written to rd. This concludes the `jal` instruction, so the Main FSM then goes back to the Fetch state.

Putting these steps together, Figure 7.45 shows the complete Main FSM state transition diagram for the multicycle processor. The function of each state is summarized below the figure. Converting the diagram to hardware is a straightforward but tedious task using the techniques of Chapter 3. Better yet, the FSM can be coded in an HDL and synthesized using the techniques of Chapter 4.

7.4.4 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. While the single-cycle processor performed all instructions in one cycle, the multicycle processor uses



State	Datapath µOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1oprs2
ExecuteI	ALUOut ← rs1opimm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Figure 7.45 Complete multicycle control FSM

varying numbers of cycles for different instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for branches, four for R-type, I-type ALU, jump, and store instructions, and five for loads. The number of clock cycles per instruction (CPI) depends on the relative likelihood that each instruction is used.

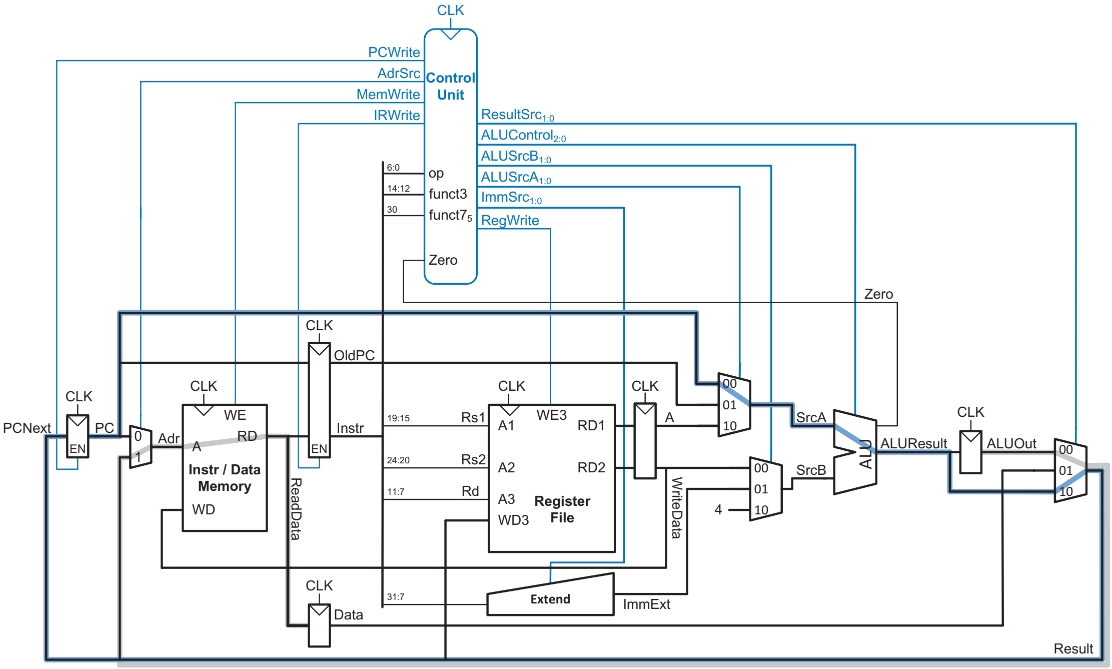


Figure 7.46 **Multicycle processor potential critical paths**

Example 7.7 MULTICYCLE PROCESSOR CPI

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R- or I-type ALU instructions.[2] Determine the average CPI for this benchmark.

Solution The average CPI is the weighted sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. For this benchmark, average CPI = (0.11)(3) + (0.10 + 0.02 + 0.52)(4) + (0.25)(5) = 4.14. This is better than the worst-case CPI of 5, which would be required if all instructions took the same number of cycles.

Recall that we designed the multicycle processor so that each cycle involved one ALU operation, memory access, or register file access. Let us assume that the register file is faster than the memory and that writing memory is faster than reading memory. Examining the datapath reveals two possible critical paths that would limit the cycle time, as shown in Figure 7.46:

1. **The path to calculate PC+4:** From the PC register through the SrcA multiplexer, ALU, and Result multiplexer back to the PC register (highlighted in thick blue lines); or

[2] Instruction frequencies from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011.

2. **The path to read data from memory:** From the ALUOut register through the Result and Adr muxes to read memory into the Data register (highlighted in thick gray lines)

Both of these paths also require a delay through the decoder after the state updates (i.e., after a t_{pcq} delay) to produce the control (multiplexer select and *ALUControl*) signals. Thus, the clock period is given in Equation 7.4.

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup} \tag{7.4}$$

The numerical values of these times will depend on the specific implementation technology.

Example 7.8 MULTICYCLE PROCESSOR PERFORMANCE COMPARISON.

Ben Bitdiddle is wondering whether the multicycle processor would be faster than the single-cycle processor. For both designs, he plans on using the 7-nm CMOS manufacturing process with the delays given in Table 7.7 on page 415. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.4).

Solution According to Equation 7.4, the cycle time of the multicycle processor is $T_{\text{c\_multi}} = t_{pcq} + t_{dec} + 2t_{mux} + t_{mem} + t_{setup} = 40 + 25 + 2(30) + 200 + 50 = 375$ ps. Using the CPI of 4.14 from Example 7.7, the total execution time is $T_{multi} = (100 \times 10^9$ instructions)(4.14 cycles/instruction)(375×10^{-12} s/cycle) = 155 seconds.

According to Example 7.4, the single-cycle processor had a total execution time of 75 seconds, so the multicycle processor is slower.

One of the original motivations for building a multicycle processor was to avoid making all instructions take as long as the slowest one. Unfortunately, this example shows that the multicycle processor is slower than the single-cycle processor, given the assumptions of CPI and circuit element delays. The fundamental problem is that even though the slowest instruction, `lw`, was broken into five steps, the multicycle processor cycle time was not nearly improved fivefold. This is partly because not all of the steps are exactly the same length and partly because the 90-ps sequencing overhead of the register clock-to-Q and setup time must now be paid on every step, not just once for the entire instruction. In general, engineers have learned that it is difficult to exploit the fact that some computations are faster than others unless the differences are large.

Compared with the single-cycle processor, the multicycle processor is likely to be less expensive because it shares a single memory for instructions and data and because it eliminates two adders. It does, however, require five nonarchitectural registers and additional multiplexers.

7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is approximately five times faster. So, ideally, the latency of each instruction is unchanged, but the throughput is five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Recall that *throughput* is the number of tasks (in this case, instructions) that complete per second. *Latency* is the time it takes for a given instruction to complete, from start to finish. (See Section 3.6)

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*. They are similar to the five steps that the multicycle processor used to perform `lw`. In the *Fetch* stage, the processor reads the instruction from instruction memory. In the *Decode* stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the *Execute* stage, the processor performs a computation with the ALU. In the *Memory* stage, the processor reads or writes data memory, if applicable. Finally, in the *Writeback* stage, the processor writes the result to the register file, if applicable.

Figure 7.47 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis and instructions are on the vertical axis. The diagram assumes component delays from Table 7.7 (see page 415) but ignores multiplexers and registers for simplicity. In the single-cycle processor in Figure 7.47(a), the first instruction is read from memory at time 0. Next, the operands are read from the register file. Then, the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file at 680 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of 200 + 100 + 120 + 200 + 60 = 680 ps (see Table 7.7 on page 415) and a throughput of 1 instruction per 680 ps (1.47 billion instructions per second).

Remember that for this abstract comparison of single-cycle and pipelined processor performance, we are ignoring the overhead of decoder, multiplexer, and register delays.

In the pipelined processor in Figure 7.47(b), the length of a pipeline stage is set at 200 ps by the slowest stage, the memory access in the Fetch or Memory stage. Each pipeline stage is indicated by solid or dashed vertical blue lines. At time 0, the first instruction is fetched from memory. At 200 ps, the first instruction enters the Decode stage, and a second instruction is fetched. At 400 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched.

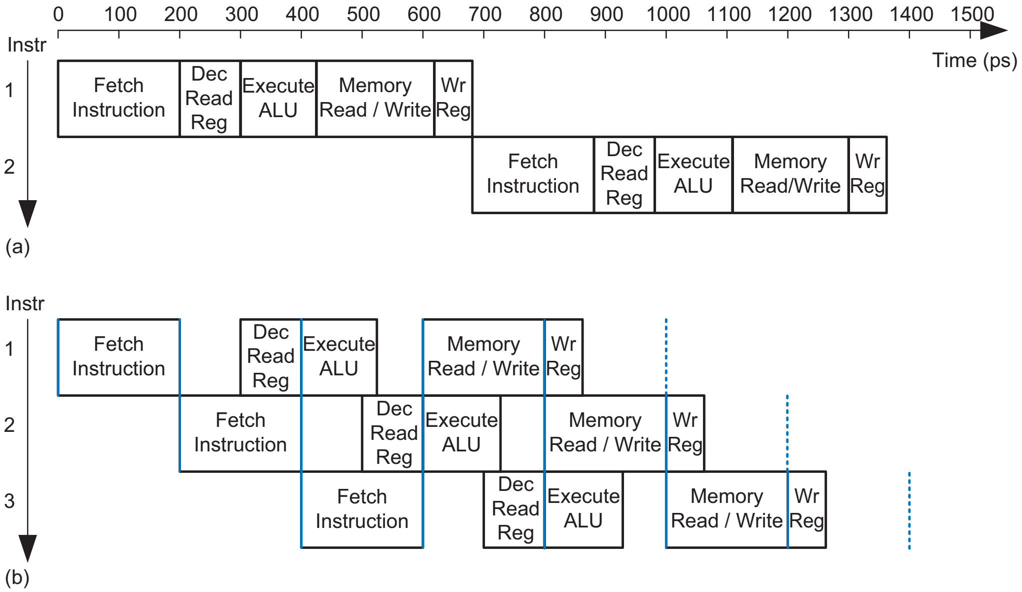


Figure 7.47 **Timing diagrams: (a) single-cycle processor and (b) pipelined processor**

And so forth, until all the instructions complete. The instruction latency is 5 × 200 = 1000 ps. Because the stages are not perfectly balanced with equal amounts of logic, the latency is longer for the pipelined processor than for the single-cycle processor. The throughput is 1 instruction per 200 ps (5 billion instructions per second)—that is, one instruction completes every clock cycle. This throughput is 3.4 times as much as the single-cycle processor—not quite 5 times but, nonetheless, a substantial speedup.

Figure 7.48 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file writeback—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycle in which a particular instruction is in each stage. For example, the sub instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the register file is writing a sum to s3, the data memory is idle, the ALU is computing (s11 & t0), t4 is being read from the register file, and the or instruction is being fetched from instruction memory. Stages are shaded to indicate when they are used. For example, the data memory is used by lw in cycle 4 and by sw in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by every instruction except sw. In the pipelined processor, the register file is

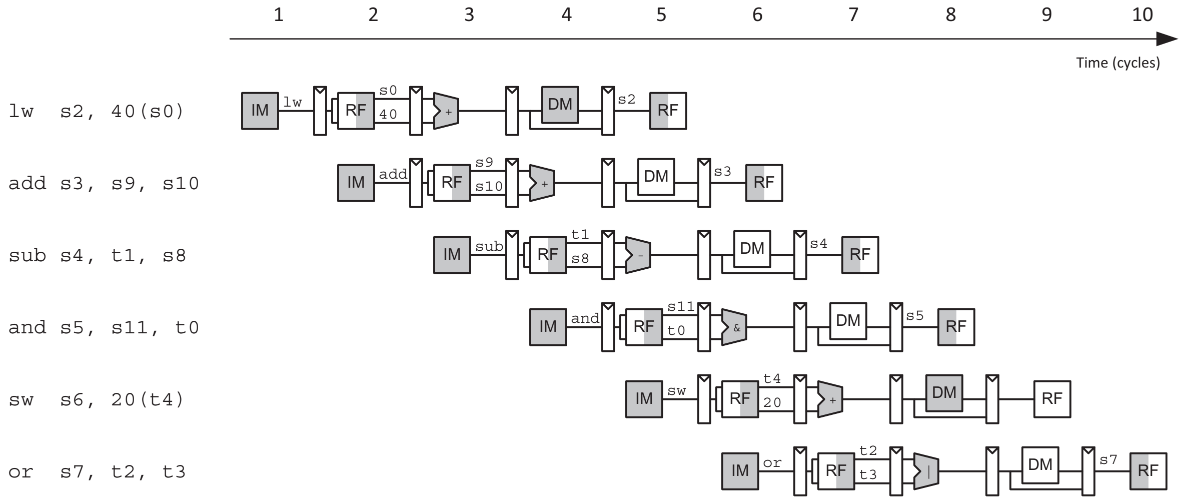


Figure 7.48 **Abstract view of pipeline in operation**

used twice in every cycle: it is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written by one instruction and read by another within a single cycle.

A central challenge in pipelined systems is handling hazards that occur when one instruction's result is needed by a subsequent instruction before the former instruction has completed. For example, if the add in Figure 7.48 used s2 as a source instead of s10, a hazard would occur because the s2 register has not yet been written by the lw instruction when it is read by add in cycle 3. After designing the pipelined datapath and control, this section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards. Finally, this section revisits performance analysis considering sequencing overhead and the impact of hazards.

7.5.1 Pipelined Datapath

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. Figure 7.49(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.49(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. So, although the register file is drawn in the Decode stage, its write address and write data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in

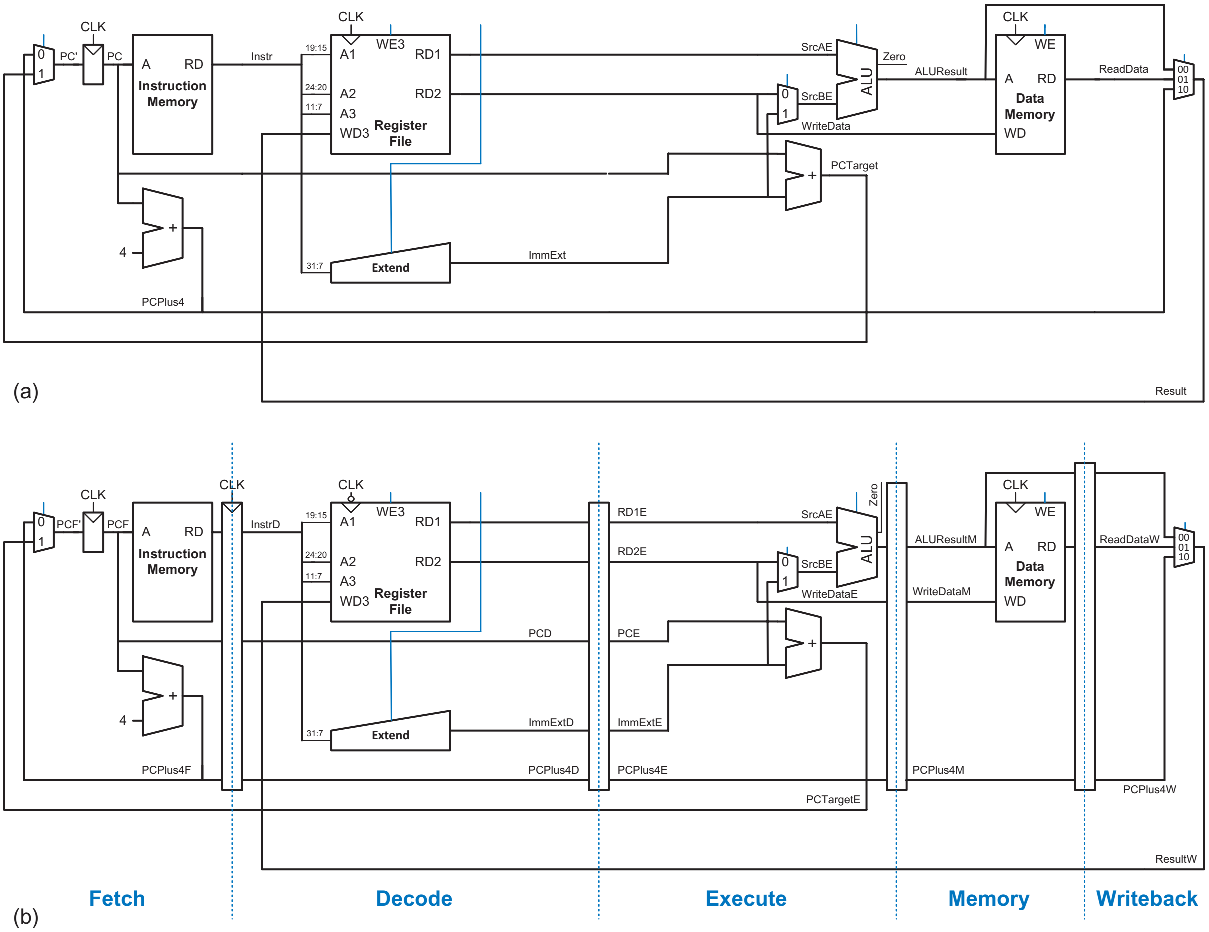


Figure 7.49 Datapaths: (a) single-cycle and (b) pipelined

Section 7.5.3. The register file in the pipelined processor writes on the falling edge of *CLK* so that it can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.

One of the subtle but critical issues in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison. Figure 7.49(b) has an error related to this issue. Can you find it?

The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from *ResultW*, a Writeback stage signal. But the destination register comes from *RdD* ($InstrD_{11:7}$), which is a Decode stage signal. In the pipeline diagram of Figure 7.48, during cycle 5, the result of the `lw` instruction would be incorrectly written to `s5` rather than `s2`.

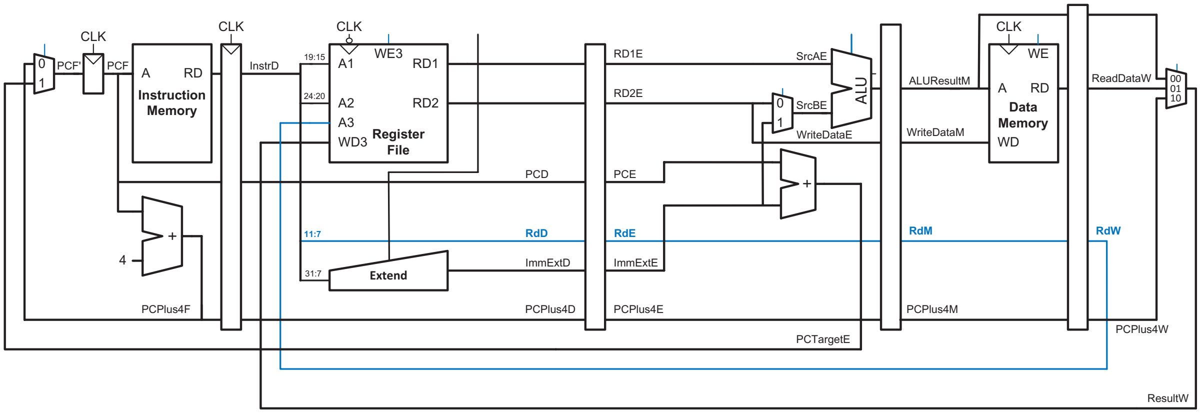


Figure 7.50 Corrected pipelined datapath

Figure 7.50 shows a corrected datapath, with the modification in blue. The *Rd* signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. *RdW* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may note that the logic to produce *PCF'* (the next PC) is also problematic because it could be updated with either a Fetch or an Execute stage signal (*PCPlus4F* or *PCTargetE*). This control hazard will be fixed in Section 7.5.3.

7.5.2 Pipelined Control

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. The control unit examines the **op**, **funct3**, and **funct7**$_5$ fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.3 for the single-cycle processor. These control signals must be pipelined along with the data so that they remain synchronized with the instruction.

The entire pipelined processor with control is shown in Figure 7.51. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *Rd* was pipelined in Figure 7.50. In addition to R-type ALU instructions, `lw`, `sw`, and `beq`, this pipelined processor also supports `jal` and I-type ALU instructions.

7.5.3 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has

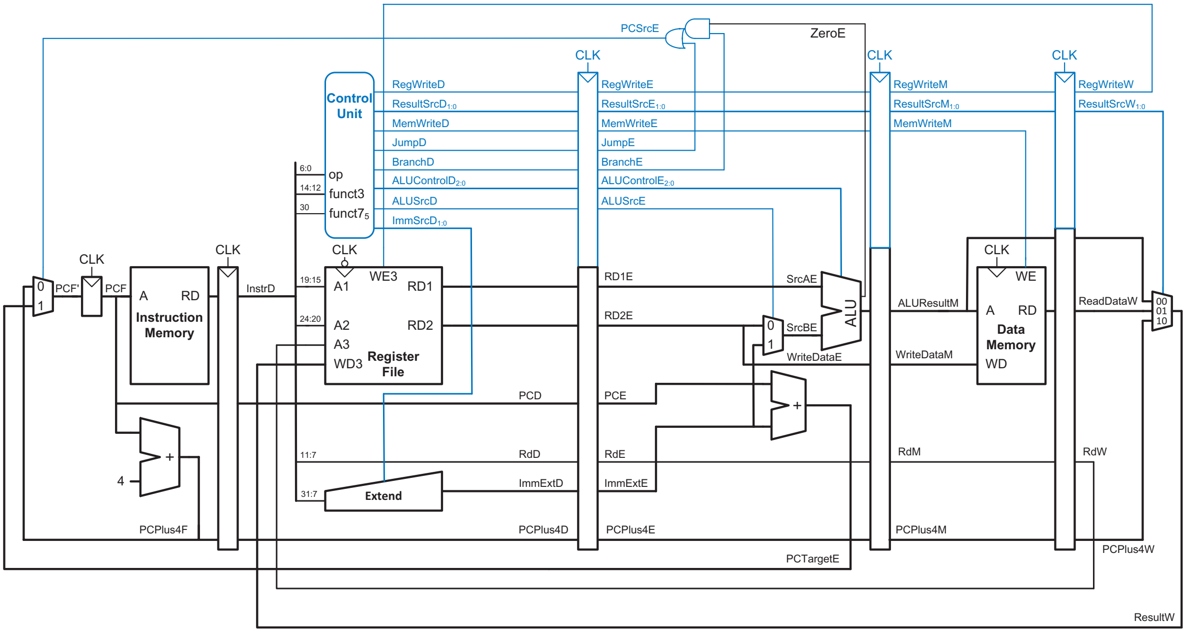


Figure 7.51 Pipelined processor with control

not yet completed, a *hazard* occurs. The register file is written during the first half of the cycle and read during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.52 illustrates hazards that occur when one instruction writes a register (s8) and subsequent instructions read this register. The blue arrows highlight when s8 is written to the register file (in cycle 5) as compared to when it is needed by subsequent instructions. This is called a *read after write* (*RAW*) *hazard*. The add instruction writes a result into s8 in the first half of cycle 5. However, the sub instruction reads s8 on cycle 3, obtaining the wrong value. The or instruction reads s8 on cycle 4, again obtaining the wrong value. The and instruction reads s8 in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of s8. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions reads that register. Without special treatment, the pipeline will compute the wrong result.

A software solution would be to require the programmer or compiler to insert nop instructions between the add and sub instructions so that the dependent instruction does not read the result (s8) until it is available

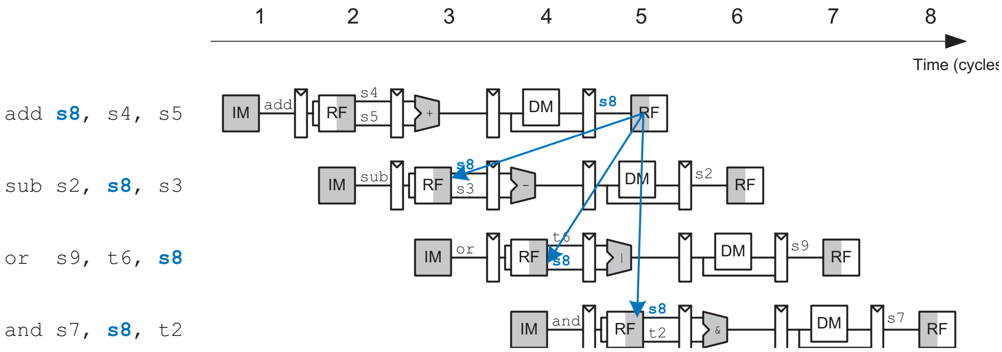


Figure 7.52 **Abstract pipeline diagram illustrating hazards**

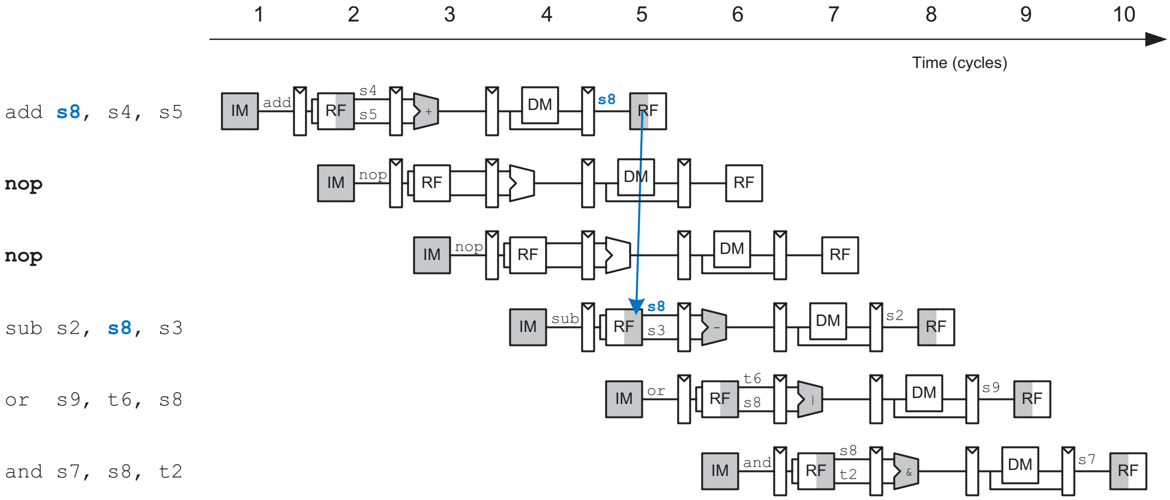


Figure 7.53 **Solving data hazard with nops**

in the register file, as shown in Figure 7.53. Such a *software interlock* complicates programming and degrades performance, so it is not ideal.

On closer inspection, observe from Figure 7.52 that the sum from the add instruction is computed by the ALU in cycle 3 and is not strictly needed by the and instruction until the ALU uses it in cycle 4. In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without waiting for the result to appear in the register file and without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be produced before the subsequent instruction uses the result. In any event, something must be done to solve hazards so that the program executes correctly despite the pipelining.

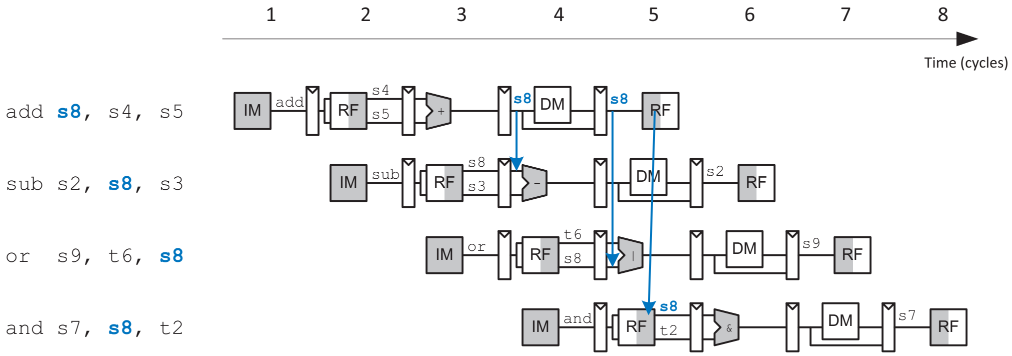


Figure 7.54 Abstract pipeline diagram illustrating forwarding

Hazards are classified as data hazards or control hazards. A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In the remainder of this section, we enhance the pipelined processor with a Hazard Unit that detects hazards and handles them appropriately so that the processor executes the program correctly.

Solving Data Hazards with Forwarding

Some data hazards can be solved by *forwarding* (also called *bypassing*) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select its operands from the register file or the Memory or Writeback stage. Figure 7.54 illustrates this principle. This program computes s8 with the add instruction and then uses s8 in the three subsequent instructions. In cycle 4, s8 is forwarded from the Memory stage of the add instruction to the Execute stage of the dependent sub instruction. In cycle 5, s8 is forwarded from the Writeback stage of the add instruction to the Execute stage of the dependent or instruction. Again, no forwarding is needed for the and instruction because s8 is written to the register file in the first half of cycle 5 and read in the second half.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.55 modifies the pipelined processor to support forwarding. It adds a *Hazard Unit* and two *forwarding multiplexers*. The hazard detection unit receives the two source registers from the instruction in the Execute stage, *Rs1E* and *Rs2E*, and the destination registers from the instructions in the Memory and Writeback

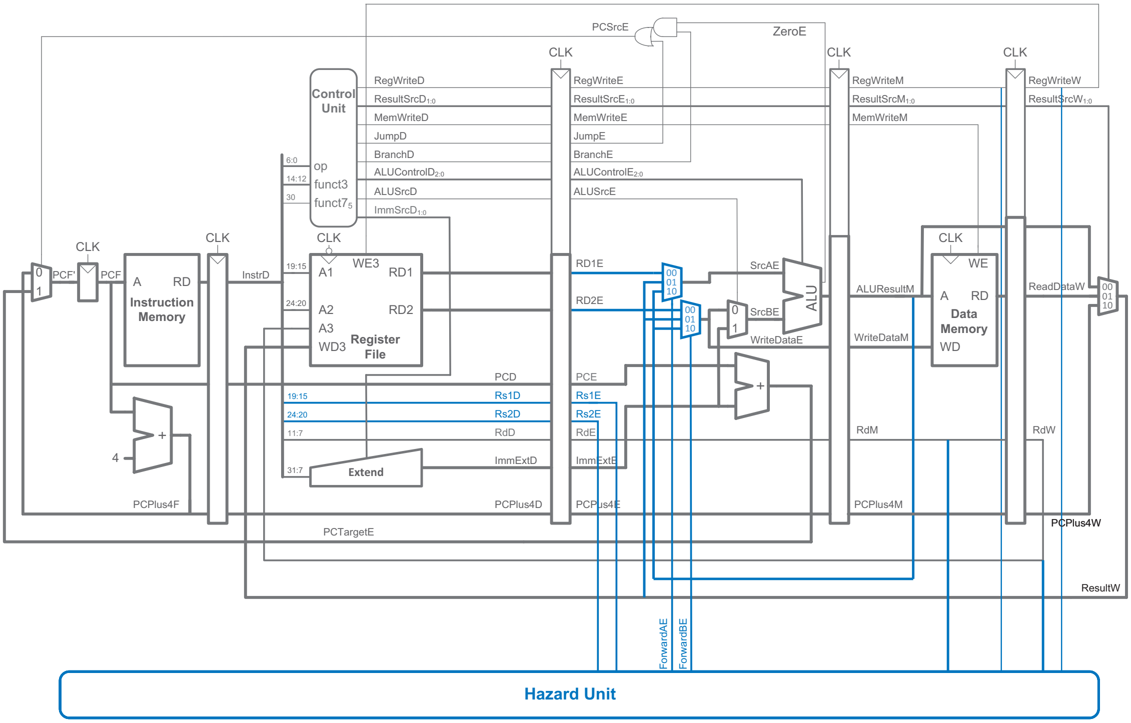


Figure 7.55 Pipelined processor with forwarding to solve some data hazards

stages, *RdM* and *RdW*. It also receives the *RegWrite* signals from the Memory and Writeback stages (*RegWriteM* and *RegWriteW*) to know whether the destination register will actually be written (e.g., the `sw` and `beq` instructions do not write results to the register file and, hence, do not have their results forwarded).

The Hazard Unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage (*ALUResultM* or *ResultW*). The Hazard Unit should forward from a stage if that stage will write a destination register *and* the destination register matches the source register. However, `x0` is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, then the Memory stage should have priority because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcAE* (*ForwardAE*) is given on the next page. The forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

```
if       $((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E\ != 0)$ then  // Forward from Memory stage
         $ForwardAE = 10$
else if  $((Rs1E == RdW)\ \&\ RegWriteW)\ \&\ (Rs1E\ != 0)$ then  // Forward from Writeback stage
         $ForwardAE = 01$
else     $ForwardAE = 00$                                          // No forwarding (use RF output)
```

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the `lw` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `lw` instruction has a *two-cycle latency* because a dependent instruction cannot use its result until two cycles later. Figure 7.56 shows this problem. The `lw` instruction receives data from memory at the end of cycle 4, but the `and` instruction needs that data (the value in `s7`) as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

A solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.57 shows stalling the dependent instruction (`and`) in the Decode stage. `and` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`or`) must remain in the Fetch stage during both cycles as well because the Decode stage is full.

In cycle 5, the result can be forwarded from the Writeback stage of `lw` to the Execute stage of `and`. Also, in cycle 5, source `s7` of the `or` instruction is read directly from the register file, with no need for forwarding.

Note that the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a *bubble*, which behaves like a nop instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stage stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling its pipeline register (i.e., the register to the left of a stage) so that the stage's inputs do not change. When a stage is stalled, all previous stages must also be stalled so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

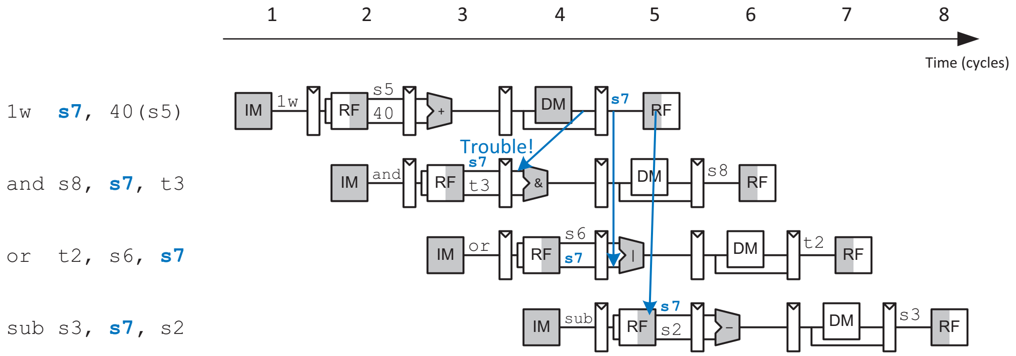


Figure 7.56 Abstract pipeline diagram illustrating trouble forwarding from `lw`

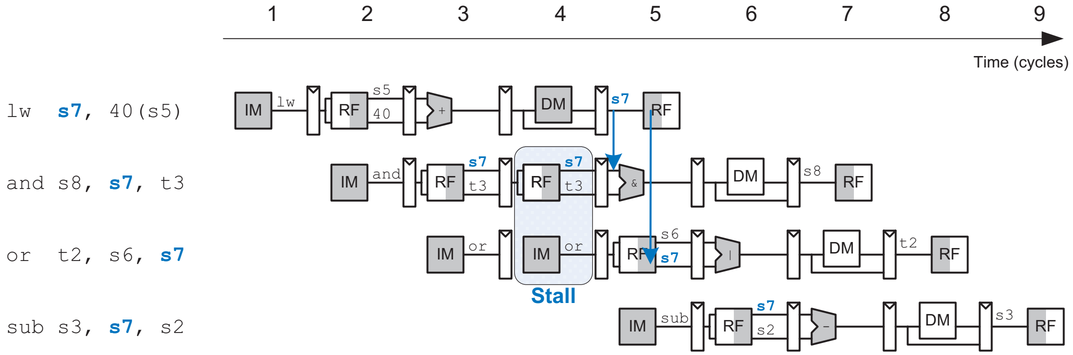


Figure 7.57 Abstract pipeline diagram illustrating stall to solve hazards

Figure 7.58 modifies the pipelined processor to add stalls for `lw` data dependencies. In order for the Hazard Unit to stall the pipeline, the following conditions must be met:

1. A load word is in the Execute stage (indicated by $ResultSrcE_0 = 1$) and
2. The load's destination register (*RdE*) matches *Rs1D* or *Rs2D*, the source operands of the instruction in the Decode stage

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When a load word (`lw`) stall occurs, *StallD* and *StallF* are asserted to force the Decode and Fetch stage pipeline registers to retain their existing values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble. The Hazard Unit *lwStall* (load word stall) signal indicates when the pipeline

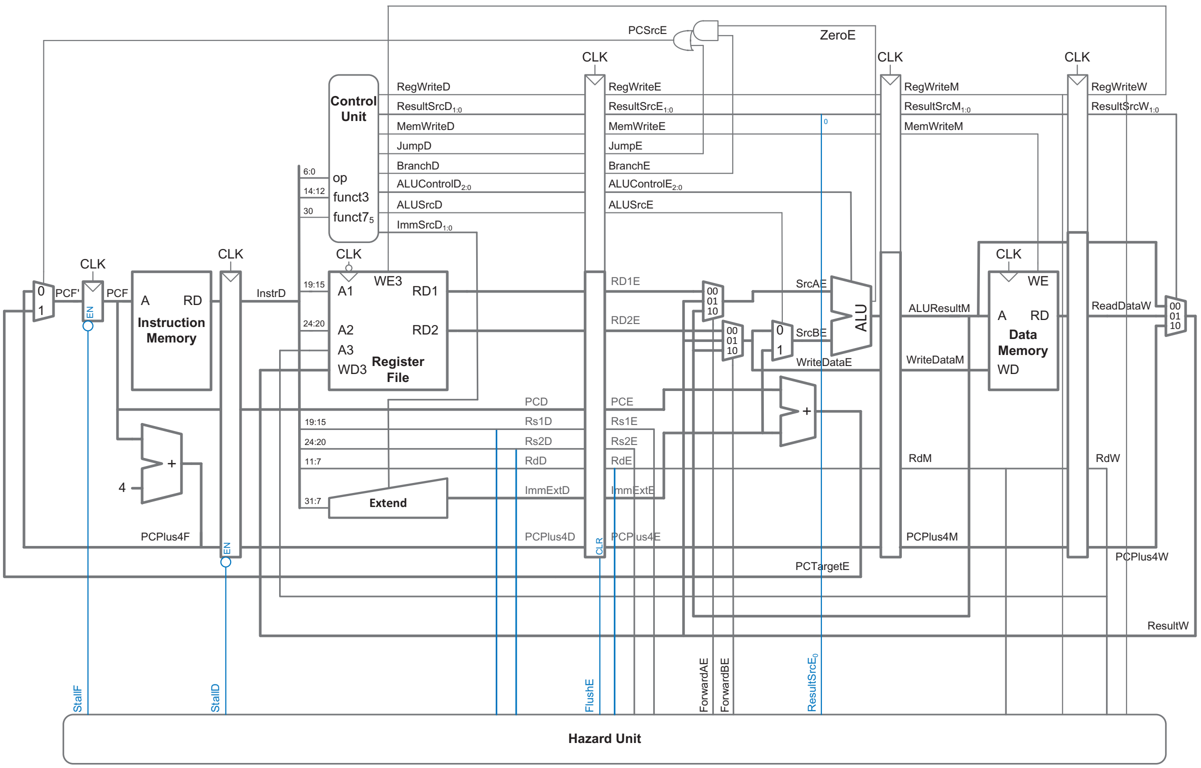


Figure 7.58 **Pipelined processor with stalls to solve `lw` data hazard**

should be stalled due to a load word dependency. Whenever *lwStall* is TRUE, all of the stall and flush signals are asserted. Hence, the logic to compute the stalls and flushes is:

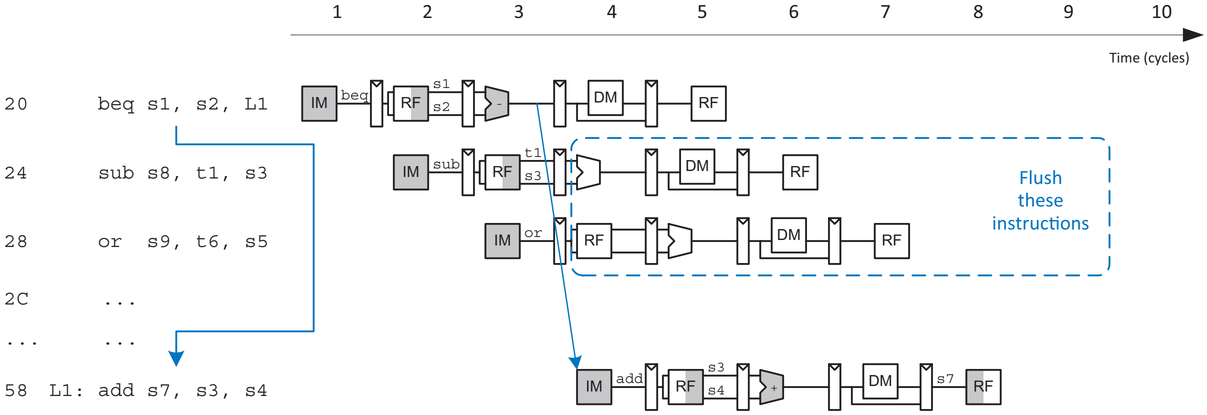
$$lwStall = ResultSrcE_0 \ \& \ ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF = StallD = FlushE = lwStall$$

Solving Control Hazards

The `beq` instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next because the branch decision has not been made by the time the next instruction is fetched.

One mechanism for dealing with this control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcE* is computed). Because the decision is made in the Execute stage, the pipeline would have to be stalled for two cycles at every branch. This would severely degrade the system performance if branches occur often, which is typically the case.

The *lwStall* logic described here could cause the processor to stall unnecessarily when the destination of the load is `x0` or when a false dependency exists—that is, when the instruction in the Decode stage is a J- or I-type instruction that randomly causes a false match between bits in their immediate fields and *RdE*. However, these cases are rare (and poor coding practice, in the case of `x0` being the load destination) and they cause only a small performance loss.



```
20      beq s1, s2, L1
24      sub s8, t1, s3
28      or  s9, t6, s5
2C      ...
...     ...
58  L1: add s7, s3, s4
```

Figure 7.59 **Abstract pipeline diagram illustrating flushing when a branch is taken**

An alternative to stalling the pipeline is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In the pipeline presented so far (Figure 7.58), the processor predicts that branches are not taken and simply continues executing the program in order until *PCSrcE* is asserted to select the next PC from *PCTargetE* instead. If the branch should have been taken, then the two instructions following the branch must be *flushed* (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the *branch misprediction penalty*.

Figure 7.59 shows such a scheme in which a branch from address 0x20 to address 0x58 is taken. The PC is not written until cycle 3, by which point the `sub` and `or` instructions at addresses 0x24 and 0x28 have already been fetched. These instructions must be flushed, and the `add` instruction is fetched from address 0x58 in cycle 4.

Finally, we must work out the stall and flush signals to handle branches and PC writes. When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. Thus, we add a synchronous clear input (CLR) to the Decode pipeline register and add the *FlushD* output to the Hazard Unit. (When CLR = 1, the register contents are cleared, that is, become 0.) When a branch is taken (indicated by *PCSrcE* being 1), *FlushD* and *FlushE* must be asserted to flush the Decode and Execute pipeline registers. Figure 7.60 shows the enhanced pipelined processor for handling control hazards. The flushes are now calculated as:

$$FlushD = PCSrcE$$
$$FlushE = lwStall \mid PCSrcE$$

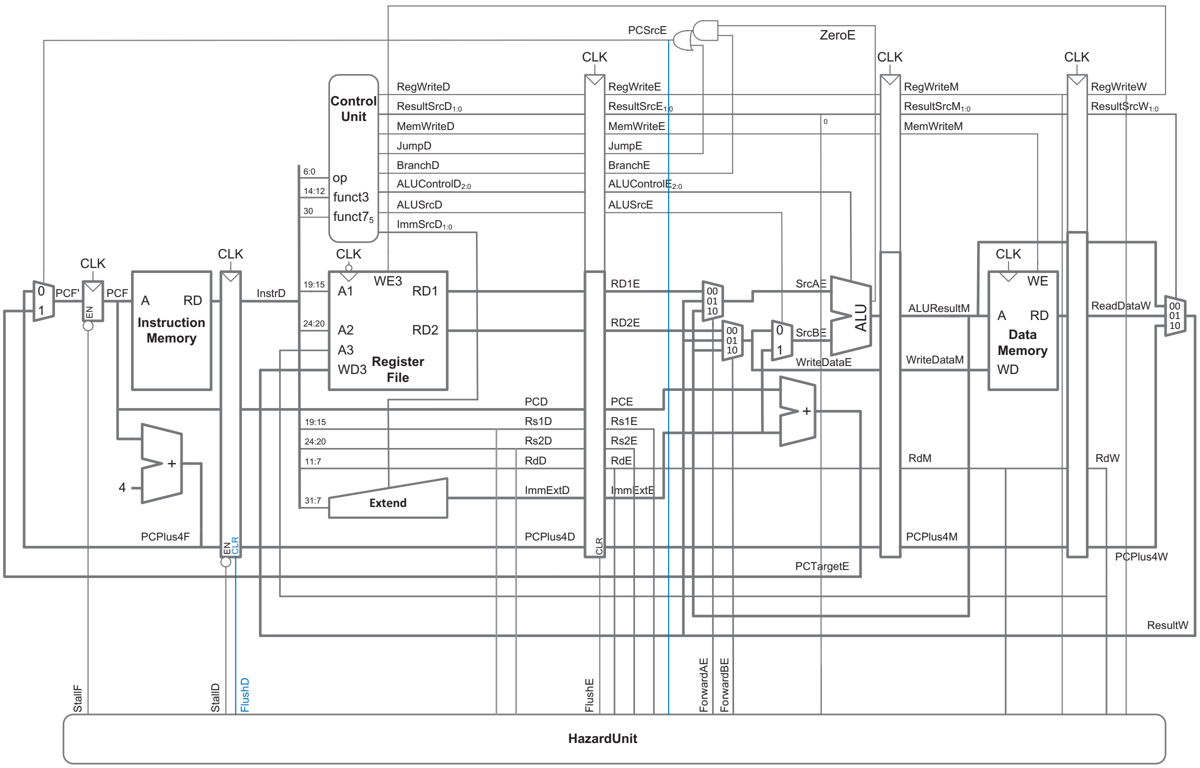


Figure 7.60 Expanded Hazard Unit for handling branch control hazard

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on a result (from another instruction) that has not yet been written into the register file. Data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by stalling the pipeline until the decision is made or by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed by now that one of the challenges of designing a pipelined processor is to understand all possible interactions between instructions and to discover all of the hazards that may exist. Figure 7.61 shows the complete pipelined processor handling all of the hazards. The hazard logic is summarized on the next page.

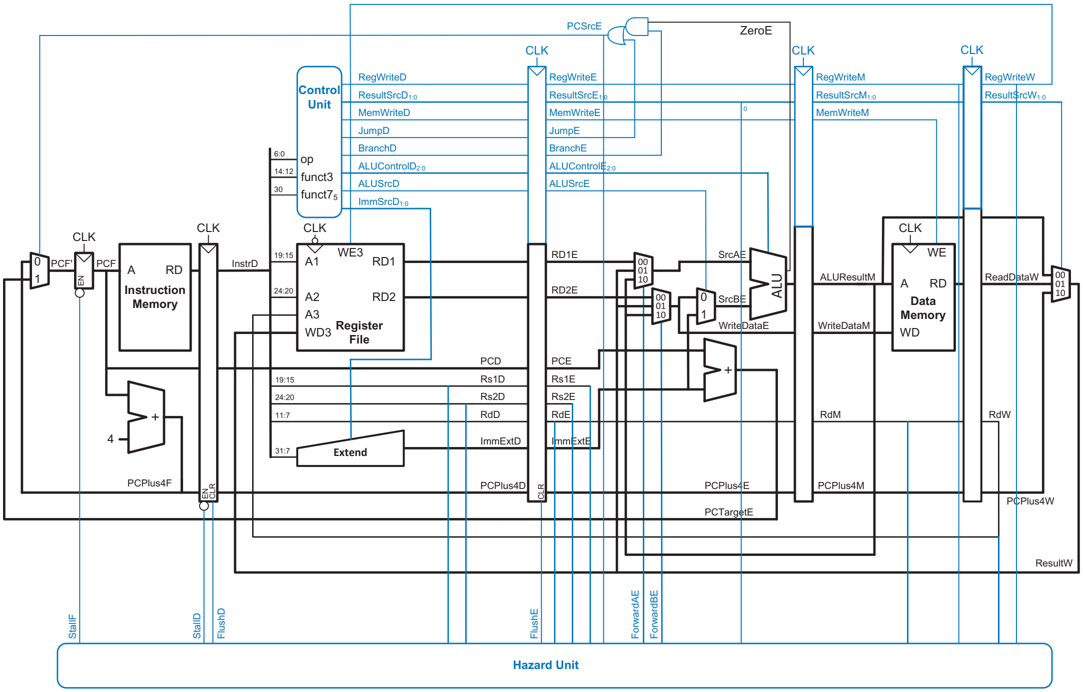


Figure 7.61 Pipelined processor with full hazard handling

Forward to solve data hazards when possible[3]:

$$
\begin{array}{lll}
\text{if} & ((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E \mathrel{!}= 0)\ \text{then} & \\
 & \qquad ForwardAE = 10 & \\
\text{else if} & ((Rs1E == RdW)\ \&\ RegWriteW)\ \&\ (Rs1E \mathrel{!}= 0)\ \text{then} & \\
 & \qquad ForwardAE = 01 & \\
\text{else} & \qquad ForwardAE = 00 &
\end{array}
$$

Stall when a load hazard occurs:

$$lwStall = ResultSrcE_0\ \&\ ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF\ \ = lwStall$$
$$StallD\ \ = lwStall$$

Flush when a branch is taken or a load introduces a bubble:

$$FlushD = PCSrcE$$
$$FlushE\ \ = lwStall \mid PCSrcE$$

[3] Recall that the forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

7.5.4 Performance Analysis

The pipelined processor ideally would have a CPI of 1 because a new instruction is *issued*—that is, fetched—every cycle. However, a stall or a flush wastes 1 to 2 cycles, so the CPI is slightly higher and depends on the specific program being executed.

Example 7.9 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.4 consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R- or I-type ALU instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that 50% of the branches are taken (mispredicted), requiring two instructions to be flushed. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution The average CPI is the weighted sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of (0.6)(1) + (0.4)(2) = 1.4. Branches take one clock cycle when they are predicted properly and three when they are not, so they have a CPI of (0.5)(1) + (0.5)(3) = 2. Jumps take three clock cycles (CPI = 3). All other instructions have a CPI of 1. Hence, for this benchmark, the average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(2) + (0.02)(3) + (0.52)(1) = 1.25.

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.61. Recall that the register file is used twice in a single cycle: it is written in the first half of the Writeback cycle and read in the second half of the Decode cycle; so these stages can use only half of the cycle time for their critical path. Another way of saying it is this: twice the critical path for each of those stages must fit in a cycle. Figure 7.62 shows the critical path for the Execute stage. It occurs when a branch is in the Execute stage that requires forwarding from the Writeback stage: the path goes from the Writeback pipeline register, through the Result, ForwardBE, and SrcB multiplexers, through the ALU and AND-OR logic to the PC multiplexer and, finally, to the PC register.

The critical path analysis for the Execute stage assumes that the Hazard Unit delay for calculating *ForwardAE* and *ForwardBE* is less than or equal to the delay of the Result multiplexer. If the Hazard Unit delay is longer, it must be included in the critical path instead of the Result multiplexer delay.

$$T_{c\_pipelined} = max\begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & Fetch \\ 2(t_{RFread} + t_{setup}) & Decode \\ t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup} & Execute \\ t_{pcq} + t_{mem} + t_{setup} & Memory \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & Writeback \end{bmatrix} \tag{7.5}$$

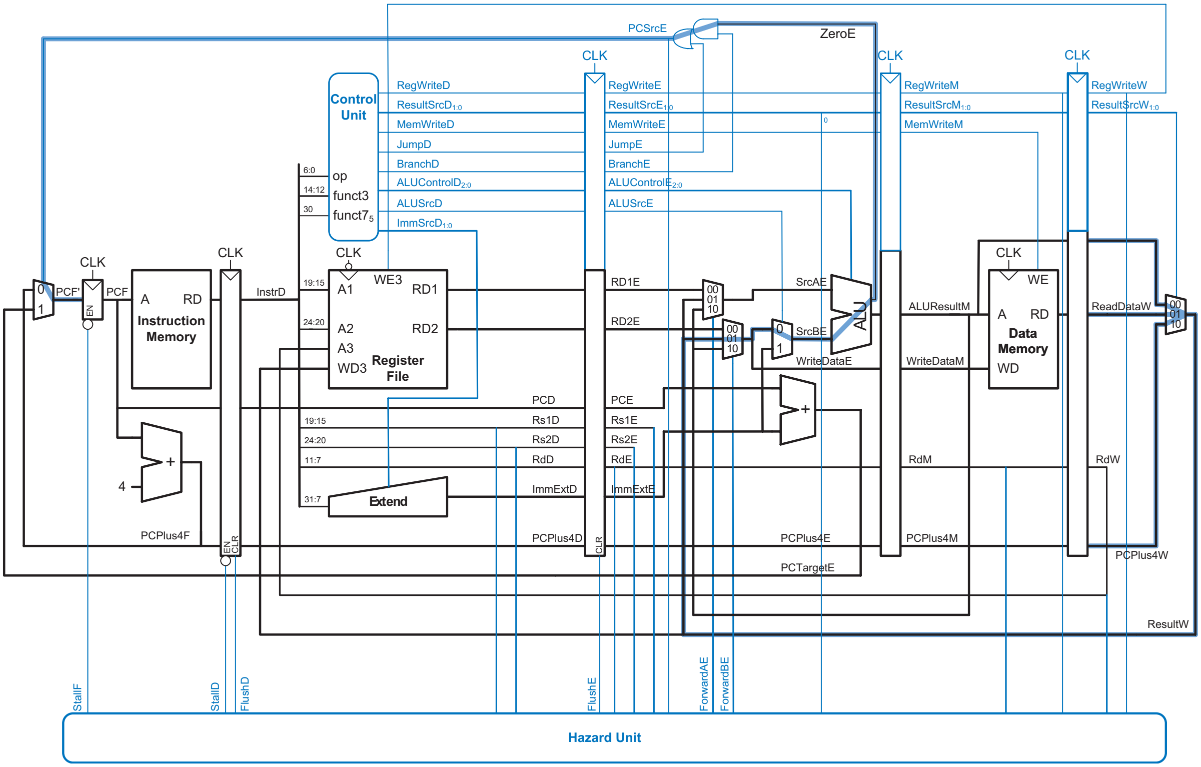


Figure 7.62 Pipelined processor critical path

Example 7.10 PIPELINED PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance with that of the single-cycle and multicycle processors considered in Examples 7.4 and 7.8. The logic delays were given in Table 7.7 (on page 415). Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution According to Equation 7.5, the cycle time of the pipelined processor is $T_{c\_pipelined}$ = max[40 + 200 + 50, 2(100 + 50), 40 + 4(30) + 120 + 20 + 50, 40 + 200 + 50, 2(40 + 30 + 60)] = 350 ps. The Execute stage takes the longest. According to Equation 7.1, the total execution time is $T_{pipelined}$ = $(100 \times 10^9$ instructions) (1.25 cycles/instruction)$(350 \times 10^{-12}$s/cycle) = 44 seconds. This compares with 75 seconds for the single-cycle processor and 155 seconds for the multicycle processor.

Our pipelined processor is unbalanced, with branch resolution in the Execute stage taking much longer than any other stage. The pipeline could be balanced better by pushing the Result multiplexer back into the Memory stage, reducing the cycle time to 320 ps.

The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the fivefold speedup one might hope to get from a five-stage pipeline.

The pipeline hazards introduce a small CPI penalty. More significantly, the sequencing overhead (clk-to-Q and setup times) of the registers applies to every pipeline stage, not just once to the overall datapath. Sequencing overhead limits the benefits one can hope to achieve from pipelining. Imbalanced delay in pipeline stages also decreases the benefits of pipelining. The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds many 32-bit pipeline registers, along with multiplexers, smaller pipeline registers, and control logic to resolve hazards.

7.6 HDL REPRESENTATION*

This section presents HDL code for the single-cycle RISC-V processor that supports the instructions discussed in this chapter. The code illustrates good coding practices for a moderately complex system. HDL code for the multicycle processor and pipelined processor are left to Exercises 7.25 to 7.27 and 7.42 to 7.44.

In this section, the instruction and data memories are separated from the datapath and connected by address and data busses. In practice, most processors pull instructions and data from separate caches. However, to handle smaller memory maps where data may be intermixed with instructions, a more complete processor must also be able to read data (in addition to instructions) from the instruction memory. Chapter 8 will revisit memory systems, including the interaction of caches with main memory.

Figure 7.63 shows a block diagram of the single-cycle RISC-V processor interfaced to external memories. The processor is composed of

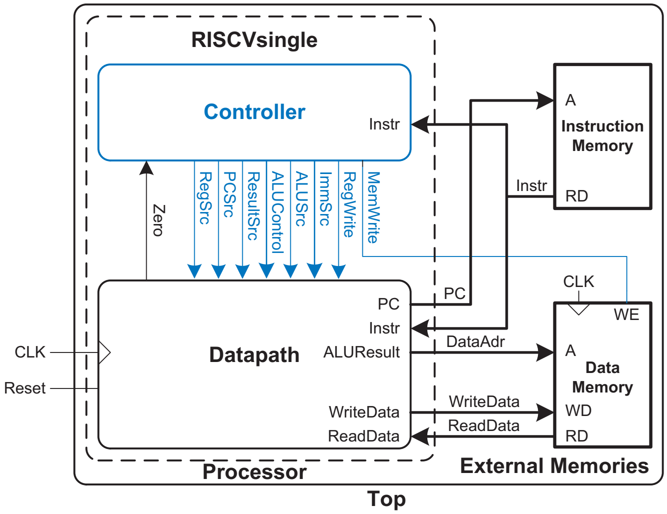


Figure 7.63 **Single-cycle processor interfaced to external memories**

the datapath from Figure 7.15 and the controller from Figure 7.16. The controller, in turn, is composed of the Main Decoder and the ALU Decoder.

The HDL code is partitioned into several sections. Section 7.6.1 provides HDL for the single-cycle processor datapath and controller. Section 7.6.2 presents the generic building blocks, such as registers and multiplexers, which are used by any microarchitecture. Section 7.6.3 introduces the test program, testbench, and external memories. The HDL and test program are available in electronic form on this book's website (see the Preface).

7.6.1 Single-Cycle Processor

The main modules of the single-cycle processor module are given in the following HDL examples.

HDL Example 7.1 SINGLE-CYCLE PROCESSOR

SystemVerilog

```
module riscvsingle(input  logic        clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
                   output logic        MemWrite,
                   output logic [31:0] ALUResult, WriteData,
                   input  logic [31:0] ReadData);

  logic       ALUSrc, RegWrite, Jump, Zero;
  logic [1:0] ResultSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
               ResultSrc, MemWrite, PCSrc,
               ALUSrc, RegWrite, Jump,
               ImmSrc, ALUControl);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrc, RegWrite,
              ImmSrc, ALUControl,
              Zero, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity riscvsingle is
  port(clk, reset:          in  STD_LOGIC;
       PC:                  out STD_LOGIC_VECTOR(31 downto 0);
       Instr:               in  STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:            out STD_LOGIC;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:            in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of riscvsingle is
  component controller
    port(op:             in  STD_LOGIC_VECTOR(6 downto 0);
         funct3:         in  STD_LOGIC_VECTOR(2 downto 0);
         funct7b5, Zero: in  STD_LOGIC;
         ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:       out STD_LOGIC;
         PCSrc, ALUSrc:  out STD_LOGIC;
         RegWrite, Jump: out STD_LOGIC;
         ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:     out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset: in STD_LOGIC;
         ResultSrc:           in  STD_LOGIC_VECTOR(1  downto 0);
         PCSrc, ALUSrc:       in  STD_LOGIC;
         RegWrite:            in  STD_LOGIC;
         ImmSrc:              in  STD_LOGIC_VECTOR(1  downto 0);
         ALUControl:          in  STD_LOGIC_VECTOR(2  downto 0);
         Zero:                out STD_LOGIC;
         PC:                  out STD_LOGIC_VECTOR(31 downto 0);
         Instr:               in  STD_LOGIC_VECTOR(31 downto 0);
         ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
         ReadData:            in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal ALUSrc, RegWrite, Jump, Zero, PCSrc: STD_LOGIC;
  signal ResultSrc, ImmSrc: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUControl: STD_LOGIC_VECTOR(2 downto 0);
begin
  c: controller port map(Instr(6 downto 0), Instr(14 downto 12),
                         Instr(30), Zero, ResultSrc, MemWrite,
                         PCSrc, ALUSrc, RegWrite, Jump,
                         ImmSrc, ALUControl);
  dp: datapath port map(clk, reset, ResultSrc, PCSrc, ALUSrc,
                         RegWrite, ImmSrc, ALUControl, Zero,
                         PC, Instr, ALUResult,WriteData,
                         ReadData);

end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       Zero,
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrc,
                  output logic       RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl);
  logic [1:0] ALUOp;
  logic       Branch;

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

  assign PCSrc = Branch & Zero | Jump;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity controller is
  port(op:             in     STD_LOGIC_VECTOR(6 downto 0);
       funct3:         in     STD_LOGIC_VECTOR(2 downto 0);
       funct7b5, Zero: in     STD_LOGIC;
       ResultSrc:      out    STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out    STD_LOGIC;
       PCSrc, ALUSrc:  out    STD_LOGIC;
       RegWrite:       out    STD_LOGIC;
       Jump:           buffer STD_LOGIC;
       ImmSrc:         out    STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:     out    STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(op:               in  STD_LOGIC_VECTOR(6 downto 0);
         ResultSrc:        out STD_LOGIC_VECTOR(1 downto 0);
         MemWrite:         out STD_LOGIC;
         Branch, ALUSrc:   out STD_LOGIC;
         RegWrite, Jump:   out STD_LOGIC;
         ImmSrc:           out STD_LOGIC_VECTOR(1 downto 0);
         ALUOp:            out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component aludec
    port(opb5:       in  STD_LOGIC;
         funct3:     in  STD_LOGIC_VECTOR(2 downto 0);
         funct7b5:   in  STD_LOGIC;
         ALUOp:      in  STD_LOGIC_VECTOR(1 downto 0);
         ALUControl: out STD_LOGIC_VECTOR(2 downto 0));
  end component;

  signal ALUOp:  STD_LOGIC_VECTOR(1 downto 0);
  signal Branch: STD_LOGIC;
begin
  md: maindec port map(op, ResultSrc, MemWrite, Branch,
                       ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  ad: aludec port map(op(5), funct3, funct7b5, ALUOp, ALUControl);
  PCSrc <= (Branch and Zero) or Jump;
end;
```

HDL Example 7.3 MAIN DECODER

SystemVerilog

```
module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);
  logic [10:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
      7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
      7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
      default:    controls = 11'bx_xx_x_x_xx_x_xx_x; // ???
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity maindec is
  port(op:             in  STD_LOGIC_VECTOR(6 downto 0);
       ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       Branch, ALUSrc: out STD_LOGIC;
       RegWrite, Jump: out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUOp:          out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(10 downto 0);
begin
  process(op) begin
    case op is
      when "0000011" => controls <= "10010010000"; -- lw
      when "0100011" => controls <= "00111000000"; -- sw
      when "0110011" => controls <= "1--00000100"; -- R-type
      when "1100011" => controls <= "01000001010"; -- beq
      when "0010011" => controls <= "10010000100"; -- I-type ALU
      when "1101111" => controls <= "11100100001"; -- jal
      when others    => controls <= "-----------"; -- not valid
    end case;
  end process;

  (RegWrite, ImmSrc(1), ImmSrc(0), ALUSrc, MemWrite,
  ResultSrc(1), ResultSrc(0), Branch, ALUOp(1), ALUOp(0),
  Jump) <= controls;
end;
```

HDL Example 7.4 ALU DECODER

SystemVerilog

```
module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract

  always_comb
    case(ALUOp)
      2'b00:              ALUControl = 3'b000; // addition
      2'b01:              ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity aludec is
  port(opb5:       in  STD_LOGIC;
       funct3:     in  STD_LOGIC_VECTOR(2 downto 0);
       funct7b5:   in  STD_LOGIC;
       ALUOp:      in  STD_LOGIC_VECTOR(1 downto 0);
       ALUControl: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture behave of aludec is
  signal RtypeSub: STD_LOGIC;
begin
  RtypeSub <= funct7b5 and opb5; -- TRUE for R-type subtract
  process(opb5, funct3, funct7b5, ALUOp, RtypeSub) begin
    case ALUOp is
```

```
                        ALUControl = 3'b000; // add, addi
               3'b010:  ALUControl = 3'b101; // slt, slti
               3'b110:  ALUControl = 3'b011; // or, ori
               3'b111:  ALUControl = 3'b010; // and, andi
               default: ALUControl = 3'bxxx; // ???
             endcase
      endcase
endmodule
```

```
      when "00" =>           ALUControl <= "000"; -- addition
      when "01" =>           ALUControl <= "001"; -- subtraction
      when others => case funct3 is    -- R-type or I-type ALU
          when "000" = if RtypeSub = '1' then
                             ALUControl <= "001"; -- sub
                           else
                             ALUControl <= "000"; -- add, addi
                           end if;
          when "010"  =>  ALUControl <= "101"; -- slt, slti
          when "110"  =>  ALUControl <= "011"; -- or, ori
          when "111"  =>  ALUControl <= "010"; -- and, andi
          when others =>  ALUControl <= "---"; -- unknown
        end case;
    end case;
  end process;
end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc, ALUSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                output logic        Zero,
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult, WriteData,
                input  logic [31:0] ReadData);

  logic [31:0] PCNext, PCPlus4, PCTarget;
  logic [31:0] ImmExt;
  logic [31:0] SrcA, SrcB;
  logic [31:0] Result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder       pcadd4(PC, 32'd4, PCPlus4);
  adder       pcaddbranch(PC, ImmExt, PCTarget);
  mux2 #(32)  pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

  // register file logic
  regfile     rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                 Instr[11:7], Result, SrcA, WriteData);
  extend      ext(Instr[31:7], ImmSrc, ImmExt);

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
  mux3 #(32)  resultmux(ALUResult, ReadData, PCPlus4,
                        ResultSrc, Result);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity datapath is
  port(clk, reset:          in     STD_LOGIC;
       ResultSrc:           in     STD_LOGIC_VECTOR(1  downto 0);
       PCSrc, ALUSrc:       in     STD_LOGIC;
       RegWrite:            in     STD_LOGIC;
       ImmSrc:              in     STD_LOGIC_VECTOR(1  downto 0);
       ALUControl:          in     STD_LOGIC_VECTOR(2  downto 0);
       Zero:                out    STD_LOGIC;
       PC:                  buffer STD_LOGIC_VECTOR(31 downto 0);
       Instr:               in     STD_LOGIC_VECTOR(31 downto 0);
       ALUResult, WriteData: buffer STD_LOGIC_VECTOR(31 downto 0);
       ReadData:            in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component adder
    port(a, b: in   STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux3 generic(width: integer);
    port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:          in  STD_LOGIC_VECTOR(1 downto 0);
         y:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component regfile
    port(clk:        in  STD_LOGIC;
         we3:        in  STD_LOGIC;
         a1, a2, a3: in  STD_LOGIC_VECTOR(4  downto 0);
         wd3:       in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:  out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
    port(instr:  in  STD_LOGIC_VECTOR(31 downto 7);
         immsrc: in  STD_LOGIC_VECTOR(1  downto 0);
         immext: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component alu
    port(a, b:       in     STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in     STD_LOGIC_VECTOR(2  downto 0);
         ALUResult:  buffer STD_LOGIC_VECTOR(31 downto 0);
         Zero:       out    STD_LOGIC);
  end component;

  signal PCNext, PCPlus4, PCTarget: STD_LOGIC_VECTOR(31 downto 0);
  signal ImmExt:                    STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB:                STD_LOGIC_VECTOR(31 downto 0);
  signal Result:                    STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd4: adder port map(PC, X"00000004", PCPlus4);
  pcaddbranch: adder port map(PC, ImmExt, PCTarget);
  pcmux: mux2 generic map(32) port map(PCPlus4, PCTarget, PCSrc,
                                       PCNext);
  -- register file logic
  rf: regfile port map(clk, RegWrite, Instr(19 downto 15),
                   Instr(24 downto 20), Instr(11 downto 7),
                   Result, SrcA, WriteData);
  ext: extend port map(Instr(31 downto 7), ImmSrc, ImmExt);
  -- ALU logic
  srcbmux: mux2 generic map(32) port map(WriteData, ImmExt,
                                         ALUSrc, SrcB);
  mainalu: alu port map(SrcA, SrcB, ALUControl, ALUResult, Zero);
  resultmux: mux3 generic map(32) port map(ALUResult, ReadData,
                                           PCPlus4, ResultSrc,
                                           Result);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any digital system, including an adder, flip-flops, and a 2:1 multiplexer. The register file appeared in HDL Example 5.8. The HDL for the ALU is left to Exercises 5.11 through 5.14.

HDL Example 7.6 ADDER

SystemVerilog

```
module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity adder is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL Example 7.7 EXTEND UNIT

SystemVerilog

```
module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      2'b00:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      2'b01:   immext = {{20{instr[31]}}, instr[31:25],
                          instr[11:7]};
               // B-type (branches)
      2'b10:   immext = {{20{instr[31]}}, instr[7],
                          instr[30:25], instr[11:8], 1'b0};
               // J-type (jal)
      2'b11:   immext = {{12{instr[31]}}, instr[19:12],
                          instr[20], instr[30:21], 1'b0};
      default: immext = 32'bx; // undefined
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity extend is
  port(instr:  in  STD_LOGIC_VECTOR(31 downto 7);
       immsrc: in  STD_LOGIC_VECTOR(1  downto 0);
       immext: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(instr, immsrc) begin
    case immsrc is
      -- I-type
      when "00" =>
        immext <= (31 downto 12 => instr(31)) & instr(31 downto 20);
      -- S-types (stores)
      when "01" =>
        immext <= (31 downto 12 => instr(31)) &
                  instr(31 downto 25) & instr(11 downto 7);
      -- B-type (branches)
      when "10" =>
        immext <= (31 downto 12 => instr(31)) & instr(7) & instr(30
                  downto 25) & instr(11 downto 8) & '0';
      -- J-type (jal)
      when "11" =>
        immext <= (31 downto 20 => instr(31)) &
                  instr(19 downto 12) & instr(20) &
                  instr(30 downto 21) & '0';
      when others =>
        immext <= (31 downto 0  => '-');
    end case;
  end process;
end;
```

HDL Example 7.8 RESETTABLE FLIP-FLOP

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH−1:0] d,
               output logic [WIDTH−1:0] q);

  always_ff @(posedge clk, posedge reset)
              if (reset) q <= 0;
              else       q <= d;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopr is
  generic(width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width−1 downto 0);
       q:          out STD_LOGIC_VECTOR(width−1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then          q <= (others => '0');
    elsif rising_edge(clk) then q <= d;
    end if;
  end process;
end;
```

HDL Example 7.9 RESETTABLE FLIP-FLOP WITH ENABLE

SystemVerilog

```
module flopenr #(parameter WIDTH = 8)
               (input  logic             clk, reset, en,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopenr is
  generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset, en) begin
    if reset = '1' then                     q <= (others => '0');
    elsif rising_edge(clk) and en = '1' then q <= d;
    end if;
  end process;
end;
```

HDL Example 7.10 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
            (input  logic [WIDTH-1:0] d0, d1,
             input  logic             s,
             output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0, d1: in  STD_LOGIC_VECTOR(width–1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width–1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s = '1' else d0;
end;
```

HDL Example 7.11 3:1 MULTIPLEXER

SystemVerilog

```
module mux3 #(parameter WIDTH = 8)
            (input  logic [WIDTH–1:0] d0, d1, d2,
             input  logic [1:0]       s,
             output logic [WIDTH–1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux3 is
  generic(width: integer := 8);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width–1 downto 0);
       s:          in  STD_LOGIC_VECTOR(1 downto 0);
       y:          out STD_LOGIC_VECTOR(width–1 downto 0));
end;

architecture behave of mux3 is
begin
  process(d0, d1, d2, s) begin
    if    (s = "00") then y <= d0;
    elsif (s = "01") then y <= d1;
    elsif (s = "10") then y <= d2;
    end if;
  end process;
end;
```

```
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020
#
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11           14        004282B3
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't execute         24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1          28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7         30        402383B3
        sw   x7, 84(x3)         # [96] = 7                  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7             38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18        3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44    40        008001EF
        addi x2, x0, 1          # shouldn't execute         44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25        48        00910133
        sw   x2, 0x20(x3)       # [100] = 25                4C        0221A023
done:   beq  x2, x2, done       # infinite loop             50        00210063
```

Figure 7.64 riscvtest.s

```
00500113
00C00193
FF718393
0023E233
0041F2B3
004282B3
02728863
0041A233
00020463
00000293
0023A233
005203B3
402383B3
0471AA23
06002103
005104B3
008001EF
00100113
00910133
0221A023
00210063
```

Figure 7.65 riscvtest.txt

7.6.3 Testbench

The testbench loads a program into the memories. The program in Figure 7.64 exercises all of the instructions by performing a computation that should produce the correct result only if all of the instructions are functioning correctly. Specifically, the program will write the value 25 to address 100 if it runs correctly, but it is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

The machine code is stored in a text file called riscvtest.txt (Figure 7.65) which is loaded by the testbench during simulation. The file consists of the machine code for the instructions written in hexadecimal, one instruction per line.

The testbench, top-level RISC-V module (that instantiates the RISC-V processor and memories), and external memory HDL code are given in the following examples. The testbench instantiates the top-level module being tested and generates a periodic clock and a reset at the start of the simulation. It checks for memory writes and reports success if the correct value (25) is written to address 100. The memories in this example hold 64 32-bit words each.

HDL Example 7.12 TESTBENCH

SystemVerilog

```
module testbench();

  logic        clk;
  logic        reset;
  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
  end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
  end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 25) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:         in  STD_LOGIC;
         WriteData, DataAdr: out STD_LOGIC_VECTOR(31 downto 0);
         MemWrite:           out STD_LOGIC);
  end component;

  signal WriteData, DataAdr:  STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, MemWrite: STD_LOGIC;
begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- Generate clock with 10 ns period
  process begin
     clk <= '1';
     wait for 5 ns;
     clk <= '0';
     wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 25 gets written to address 100 at end of program
  process(clk) begin
    if(clk'event and clk = '0' and MemWrite = '1') then
      if(to_integer(DataAdr) = 100 and
         to_integer(writedata) = 25) then
        report "NO ERRORS: Simulation succeeded" severity
        failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite,
                       DataAdr, WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is
  port(clk, reset:        in     STD_LOGIC;
       WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:          buffer STD_LOGIC);
end;

architecture test of top is
  component riscvsingle
    port(clk, reset:       in  STD_LOGIC;
      PC:                  out STD_LOGIC_VECTOR(31 downto 0);
      Instr:               in  STD_LOGIC_VECTOR(31 downto 0);
      MemWrite:            out STD_LOGIC;
      ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:            in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
  port(clk, we: in  STD_LOGIC;
    a, wd:      in  STD_LOGIC_VECTOR(31 downto 0);
    rd:         out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  signal PC, Instr, ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  rvsingle: riscvsingle port map(clk, reset, PC, Instr,
                                 MemWrite, DataAdr,
                                 WriteData, ReadData);
  imem1: imem port map(PC, Instr);
  dmem1: dmem port map(clk, MemWrite, DataAdr, WriteData,
                       ReadData);
end;
```

HDL Example 7.14 INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
use ieee.std_logic_textio.all;

entity imem is
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
type ramtype is array(63 downto 0) of
                    STD_LOGIC_VECTOR(31 downto 0);
  -- initialize memory from file
  impure function init_ram_hex return ramtype is
  file text_file : text open read_mode is "riscvtest.txt";
    variable text_line : line;
    variable ram_content : ramtype;
    variable i : integer := 0;
  begin
    for i in 0 to 63 loop -- set all contents low
      ram_content(i) := (others => '0');
    end loop;
    while not endfile(text_file) loop -- set contents from file
      readline(text_file, text_line);
      hread(text_line, ram_content(i));
      i := i + 1;
    end loop;

    return ram_content;
  end function;

  signal mem : ramtype := init_ram_hex;
  begin
  -- read memory
  process(a) begin
    rd <= mem(to_integer(a(31 downto 2)));
  end process;
end;
```

HDL Example 7.15 DATA MEMORY

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                    STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if rising_edge(clk) then
        if (we = '1') then mem(to_integer(a(7 downto 2))) := wd;
        end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

7.7 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance, we would like to speed up the clock and/or reduce the CPI. This section surveys some existing speedup techniques. The implementation details become quite complex, so we focus on the concepts. Hennessy and Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Advances in integrated circuit manufacturing have steadily reduced transistor sizes. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption has become an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.7.1 Deep Pipelines

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 8 to 20 stages are now commonly used. For example, the SweRV EH1 core, the open-source commercial RISC-V processor developed by Western Digital, has nine pipeline stages.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). Due to this sequencing overhead, adding more pipeline stages gives diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

In the late 1990's and early 2000's, microprocessors were marketed largely based on clock frequency ($f = 1/T_c$). This pushed microprocessors to use very deep pipelines (20–31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits to overall performance were questionable. Power is proportional to clock frequency and increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are shorter.

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages. The single-cycle processor has a propagation delay of 750 ps through the combinational logic. The sequencing overhead of a register is 90 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.25. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution The cycle time for an N-stage pipeline is $T_c = [(750/N) + 90]$ ps. The CPI is $1.25 + 0.1(N - 5)$, where $N \geq 5$. The time per instruction (i.e., instruction time) is the product of the cycle time T_c and the CPI. Figure 7.66 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 281 ps at $N = 8$ stages. This minimum is only slightly better than the 295 ps per instruction achieved with a five-stage pipeline, and the curve is almost flat between 7 to 10 stages.

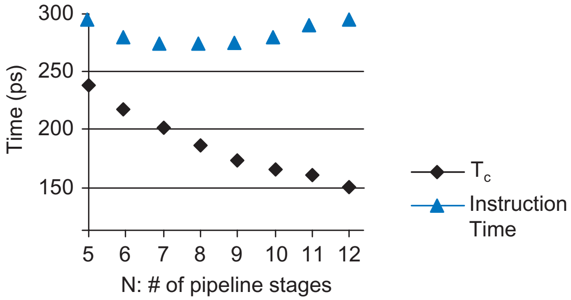


Figure 7.66 Cycle time and instruction time vs. the number of pipeline stages

7.7.2 Micro-Operations

Recall our design principles of "regularity supports simplicity" and "make the common case fast." Pure reduced instruction set computer (RISC) architectures such as RISC-V contain only simple instructions, typically those that can be executed in a single cycle on a simple, fast datapath with a three-ported register file, single ALU, and single data memory access like the ones we have developed in this chapter. Complex instruction set computer (CISC) architectures generally include instructions requiring more registers, more additions, or more than one memory access per instruction. For example, the x86 instruction `ADD [ESP],[EDX+80+EDI*2]` involves reading the three registers (`ESP`, `EDX`, and `EDI`), adding the base (`EDX`), displacement (80), and scaled index (EDI*2), reading two memory locations, summing their values, and writing the result back to memory. A microprocessor that could perform all of these functions at once would be unnecessarily slow when executing more common, simpler instructions.

Microarchitects make the decision of whether to provide hardware to implement a complex operation directly or to break it into micro-op sequences. They make similar decisions about other options described later in this section. These choices lead to different points in the performance-power-cost design space.

Computer architects of CISC processors make the common case fast by defining a set of simple *micro-operations* (also known as *micro-ops* or μops) that can be executed on simple datapaths. Each CISC instruction is decoded into one or more micro-ops. For example, if we defined μops resembling RISC-V instructions, and used temporary registers t1 and t2 to hold intermediate results, then the x86 instruction above could become six μops:

```
slli t2, EDI, 1  # t2 = EDI*2
add  t1, EDX, t2 # t1 = EDX + EDI*2
lw   t1, 80(t1)  # t1 = MEM[EDX + EDI*2 + 80]
lw   t2, 0(ESP)  # t2 = MEM[ESP]
add  t1, t2, t1  # t1 = MEM[ESP] + MEM[EDX + EDI*2 + 80]
sw   t1, 0(ESP)  # MEM[ESP] = MEM[ESP] + MEM[EDX + EDI*2 + 80]
```

7.7.3 Branch Prediction

An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur at the beginning of a loop to check a condition and branch past the loop when the condition is no longer met (e.g., in *for* and *while* loops). Loops tend to execute many times, so these forward branches are usually not taken. Other branches occur when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a *do/while* loop). Again, because loops tend to execute many times, these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches are taken and forward branches are not. This is called *static branch prediction*, because it does not depend on the history of the program.

However, branches, especially forward branches, are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

To see the operation of dynamic branch predictors, consider the following loop from Code Example 6.20. The loop repeats 10 times, and the branch out of the loop (`bge s0, t0, done`) is taken only on the last iteration.

```
    addi s1, zero, 0  # s1 = sum = 0
    addi s0, zero, 0  # s0 = i = 0
    addi t0, zero, 10 # t0 = 10
for:
    bge  s0, t0, done # i >= 10?
    add  s1, s1, s0   # sum = sum + i
    addi s0, s0, 1    # i = i + 1
    j    for          # repeat loop
done:
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `beq` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A *two-bit dynamic branch predictor* solves this problem by having four states: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken, as shown in Figure 7.67. When the loop is repeating, it enters the Strongly Not Taken state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the Weakly Not Taken state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the Strongly Not Taken state. In summary, a two-bit branch predictor mispredicts only the last branch of a loop. It is called a two-bit branch predictor because it requires two bits to encode the four states.

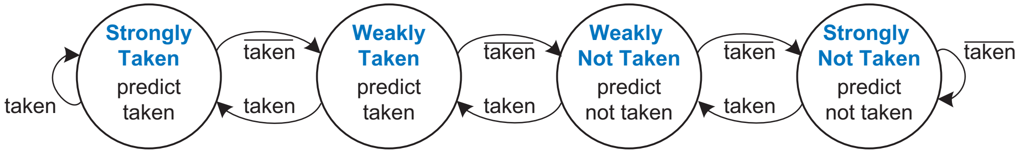


Figure 7.67 Two-bit branch predictor state transition diagram

Western Digital's RISC-V SweRV EH1 is a two-way superscalar core.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

A *scalar* processor acts on one piece of data at a time. A *vector* processor acts on several pieces of data with a single instruction. A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Scalar processors are classified as single-instruction single-data (SISD) machines. Vector processors and graphics processors (GPUs: graphics processing units) are single-instruction multiple-data (SIMD) machines. Multiprocessors, such as multicore processors, are classified as multiple-instruction multiple-data (MIMD) machines. Typically, MIMD machines have a single program running that uses all or a subset of the cores. This style of programming is called single-program multiple-data (SPMD), but multiprocessors can be programmed in other ways as well.

7.7.4 Superscalar Processors

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.68 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

Figure 7.69 is a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or IPC. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.70 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are indicated in blue. The add instruction is dependent on s8, which is

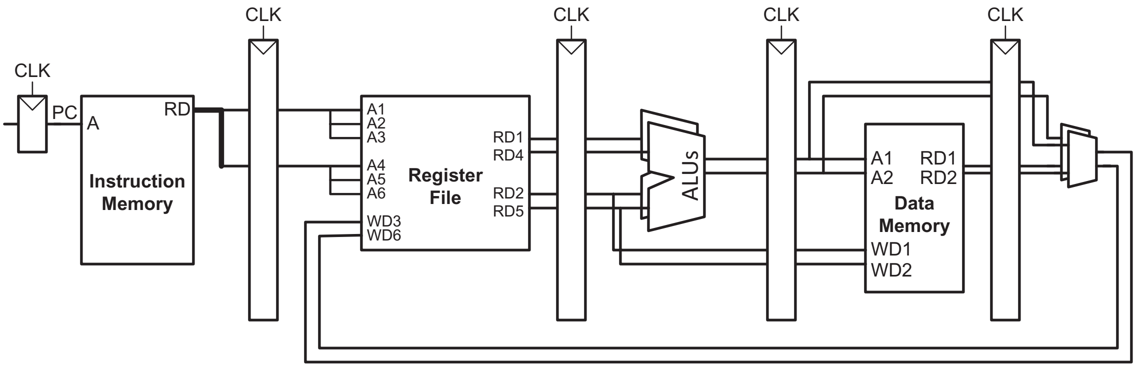


Figure 7.68 Superscalar datapath

produced by the lw instruction, so it cannot be issued at the same time as lw. The add instruction stalls for yet another cycle so that lw can forward s8 to add in cycle 5. The other dependencies (between sub and and based on s8, and between or and sw based on s11) are handled by forwarding results produced in one cycle to be consumed in the next. This program requires five cycles to issue six instructions, for an IPC of 6/5 = 1.2.

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Using multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

Our RISC-V pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980's and 1990's because they efficiently handled the long vectors of data common in scientific computations, and they are heavily used now in GPUs. Processing vectors, or SIMD in general, is an example of *data-level parallelism*, where multiple data can be operated on in parallel. Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors. However, modern processors also include hardware, called SIMD units, to handle short vectors of data that are common in multimedia and graphics applications. RISC-V includes the vector (V) extension to support vector operations.

7.7.5 Out-of-Order Processor

To cope with the problem of dependencies, an out-of-order processor looks ahead across many instructions to issue independent instructions as rapidly as possible. The instructions can issue in a different order than that written by the programmer as long as dependencies are honored so that the program produces the intended result.

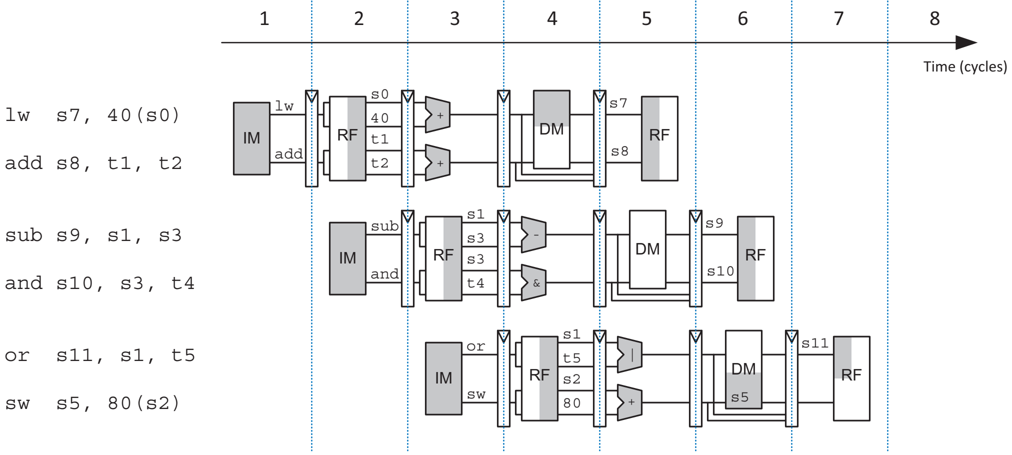


Figure 7.69 Abstract view of a superscalar pipeline in operation

Consider running the same program from Figure 7.70 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.71 shows the data dependencies and the

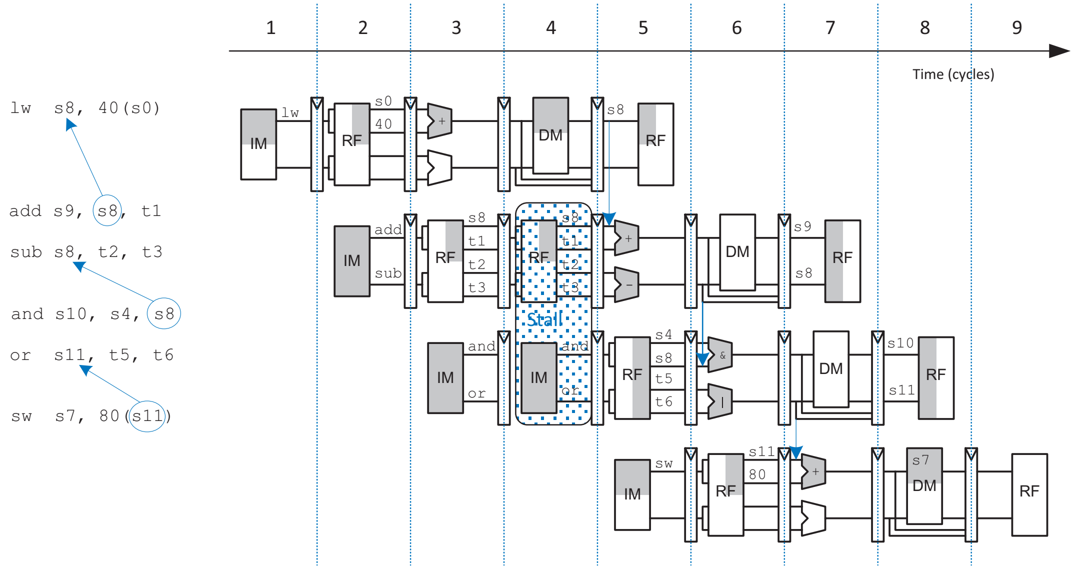


Figure 7.70 Program with data dependencies

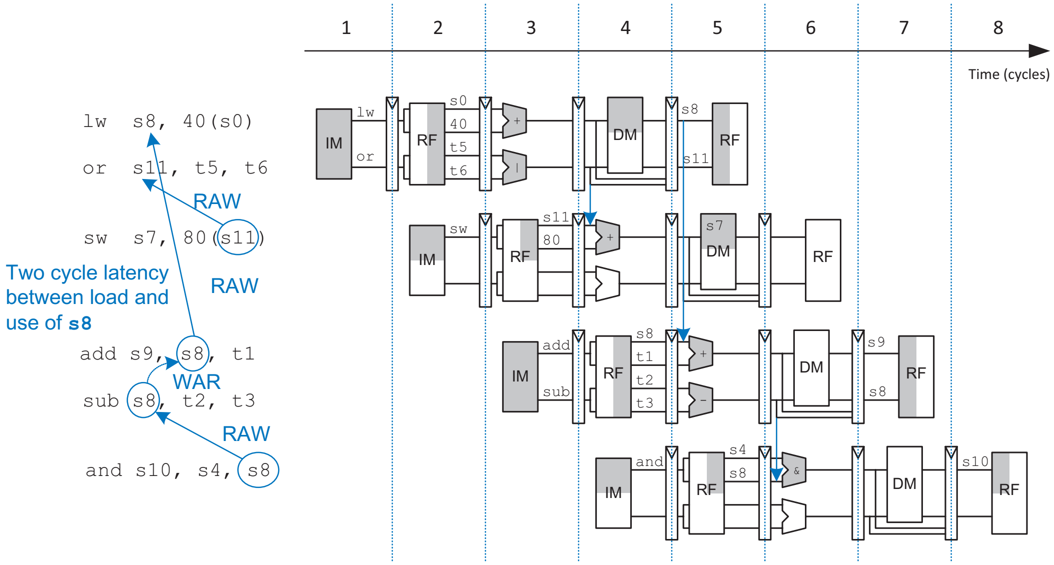


Figure 7.71 Out-of-order execution of a program with dependencies

operation of the processor. The classifications of dependencies as RAW and WAR will be discussed soon. The constraints on issuing instructions are:

Cycle 1

- The lw instruction issues.
- The add, sub, and and instructions are dependent on lw by way of s8, so they cannot issue yet. However, the or instruction is independent, so it also issues.

Cycle 2

- Remember that a two-cycle latency exists between issuing lw and a dependent instruction, so add cannot issue yet because of the s8 dependence. sub writes s8, so it cannot issue before add, lest add receive the wrong value of s8. and is dependent on sub.
- Only the sw instruction issues.

Cycle 3

- On cycle 3, s8 is available (or, rather, will be when add needs it), so the add issues. sub issues simultaneously, because it will not write s8 until after add consumes (i.e., reads) it.

Cycle 4

- The and instruction issues. s8 is forwarded from sub to and.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 6/4 = 1.5. The dependence of add on lw by way of s8 is a *read after write* (*RAW*) hazard. add must not read s8 until after lw has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of sw on or by way of s11 and of and on sub by way of s8 are RAW dependencies.

The dependence between sub and add by way of s8 is called a *write after read* (*WAR*) hazard or an *antidependence*. sub must not write s8 before add reads s8, so that add receives the correct value according to the original order of the program. WAR hazards could not occur in the simple pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, sub) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the sub instruction had written s3 instead of s8, then the dependency would disappear and sub could be issued before add. The RISC-V architecture has only 31 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all of the other registers are in use.

A third type of hazard, not shown in the program, is called a *write after write* (*WAW*) hazard or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being written to the register. For example, in the following code, `lw` and `add` both write `s7`. The final value in `s7` should come from `add` according to the order of the program. If an out-of-order processor attempted to execute `add` first and then `lw`, a WAW hazard would occur.

```
lw  s7, 0(t3)
add s7, s1, t2
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer using the same destination register for two unrelated instructions. If the `add` instruction were issued first, then the program could eliminate the WAW hazard by discarding the result of the `lw` instead of writing it to `s7`. This is called *squashing* the `lw`.[4]

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction-level parallelism* (ILP) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than two or three, even with six-way superscalar datapaths with out-of-order execution.

7.7.6 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR and WAW hazards. Register renaming adds some nonarchitectural renaming registers to the processor. For example, a processor might add 20 renaming registers, called `r0` to `r19`. The programmer cannot

[4] You might wonder why the `lw` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `lw` potentially may produce a load address misaligned exception or load access fault, so it must be issued to check for the exception, even though the result can be discarded.

use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

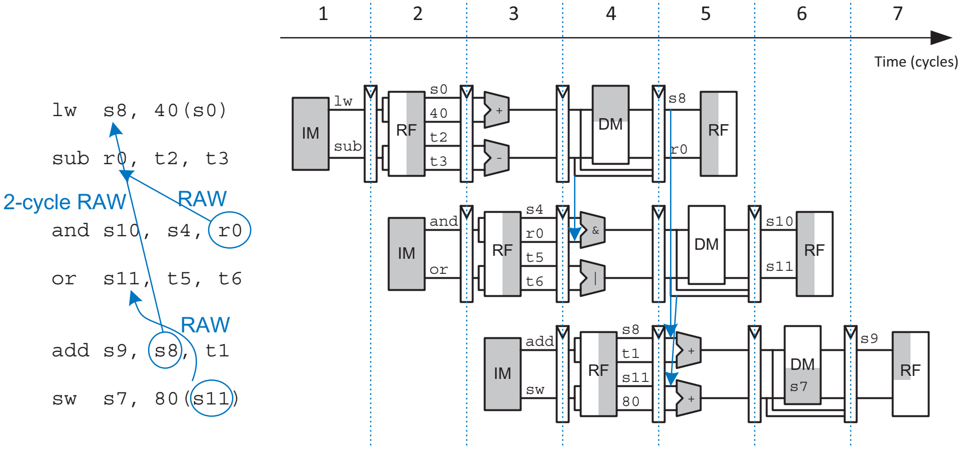
For example, in the previous section, a WAR hazard occurred between the `sub s8, t2, t3` and `add s9, s8, t1` instructions based on reusing `s8`. The out-of-order processor could rename `s8` to `r0` for the `sub` instruction. Then, `sub` could be executed sooner, because `r0` has no dependency on the `add` instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, `s8` must also be renamed to `r0` in the `and` instruction, because it refers to the result of `sub`. Figure 7.72 shows the same program from Figure 7.71 executing on an out-of-order processor with register renaming. `s8` is renamed to `r0` in `sub` and `and` to eliminate the WAR hazard. The constraints on issuing instructions are:

Cycle 1

- The `lw` instruction issues.
- The `add` instruction is dependent on `lw` by way of `s8`, so it cannot issue yet. However, the `sub` instruction is independent now that its destination has been renamed to `r0`, so `sub` also issues.

Cycle 2

- Remember that a two-cycle latency must exist between issuing `lw` and a dependent instruction, so `add` cannot issue yet because of the `s8` dependence.



```
lw   s8, 40(s0)

sub  r0, t2, t3

and  s10, s4, r0

or   s11, t5, t6

add  s9, s8, t1

sw   s7, 80(s11)
```

Figure 7.72 Out-of-order execution of a program using register renaming

- The `and` instruction is dependent on `sub`, so it can issue. `r0` is forwarded from `sub` to `and`.
- The `or` instruction is independent, so it also issues.

Cycle 3

- On cycle 3, `s8` is available, so the `add` issues.
- `s11` is also available, so `sw` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.7.7 Multithreading

Because the instruction-level parallelism (ILP) of real programs tends to be fairly low, adding more execution units to a superscalar or out-of-order processor gives diminishing returns. Another problem, discussed in Chapter 8, is that memory is much slower than the processor. Most loads and stores access a smaller and faster memory, called a *cache*. However, when the instructions or data are not available in the cache, the processor may stall for 100 or more cycles while retrieving the information from the main memory. Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

To explain multithreading, we need to define a few new terms. A program running on a computer is called a *process*. Computers can run multiple processes simultaneously; for example, you can play music on a PC while surfing the web and running a virus checker. Each process consists of one or more *threads* that also run simultaneously. For example, a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user works, and a third thread printing the document. In this way, the user does not have to wait, for example, for a document to finish printing before being able to type again. The degree to which a process can be split into multiple threads that can run simultaneously defines its level of *thread-level parallelism* (TLP).

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the operating system (OS). When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*. As long as the processor switches through all threads fast enough, the user perceives all of the threads as running at the same time. RISC-V dedicates one of its 32 registers, the thread pointer register, `tp` (i.e., `x4`), to point to (hold the address of) a thread's local memory.

A *hardware multithreaded* processor contains more than one copy of its architectural state so that more than one thread can be active at a time. For example, if we extended a processor to have four program counters and 128 registers, four threads could be available at one time. If one thread stalls while waiting for data from main memory, then the processor could context switch to another thread without any delay, because the program counter and registers are already available. Moreover, if one thread lacks sufficient parallelism to keep all execution units busy in a superscalar design, then another thread could issue instructions to the idle units. Switching between threads can either be fine-grained or coarse-grained. *Fine-grained* multithreading switches between threads on each instruction and must be supported by hardware multithreading. *Coarse-grained* multithreading switches out a thread only on expensive stalls, such as long memory accesses due to cache misses.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread. Multithreading is also relatively inexpensive to implement because it replicates only the PC and register file, not the execution units and memories.

7.7.8 Multiprocessors

With contributions from Matthew Watkins

Modern processors have enormous numbers of transistors available. Using them to increase the pipeline depth or to add more execution units to a superscalar processor gives little performance benefit and wastes power. Around the year 2005, computer architects made a major shift to building multiple copies of the processor on the same chip; these copies are called *cores*.

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. Three common classes of multiprocessors include *symmetric* (or *homogeneous*) multiprocessors, *heterogeneous* multiprocessors, and *clusters*.

Symmetric Multiprocessors

Symmetric multiprocessors include two or more identical processors sharing a single main memory. The multiple processors may be separate chips or multiple cores on the same chip.

Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running more threads simultaneously is easy; the threads are simply divided up among the processors. Unfortunately, typical PC users need to run only a small number of threads at any given time. Running a particular thread faster is much more challenging. The programmer must divide the existing thread into multiple threads to execute on each processor. This becomes tricky when the processors need to communicate with each other. One of the major challenges for computer designers and programmers is to effectively use large numbers of processor cores.

Symmetric multiprocessors have a number of advantages. They are relatively simple to design because the processor can be designed once and then replicated multiple times to increase performance. Programming for and executing code on a symmetric multiprocessor is also relatively straightforward because any program can run on any processor in the system and achieve approximately the same performance.

Heterogeneous Multiprocessors

Unfortunately, continuing to add more and more symmetric cores is not guaranteed to provide continued performance improvement. Most consumer applications use only a few threads at any given time, and consumers typically have only a couple of applications actually computing simultaneously. Although this is enough to keep dual-core and quad-core systems busy, unless programs start incorporating significantly more parallelism, continuing to add more cores beyond this point will provide diminishing benefits. As an added issue, because general-purpose processors are designed to provide good average performance, they are generally not the most power-efficient option for performing a given operation. This energy inefficiency is especially important in highly power-constrained systems, such as mobile phones.

Heterogeneous multiprocessors aim to address these issues by incorporating different types of cores and/or specialized hardware in a single system. Each application uses those resources that provide the best performance, or power-performance ratio, for that application. Because transistors are fairly plentiful these days, the fact that not every application will make use of every piece of hardware is of lesser concern.

Heterogeneous systems can take a number of forms. A heterogeneous system can incorporate cores with the same architecture but different microarchitectures, each with different power, performance, and area trade-offs. The RISC-V architecture was specifically designed with the aim of supporting a range of processor implementations, from low-cost embedded processors to high-performance multiprocessors. Another heterogeneous strategy is accelerators, in which a system contains special-purpose hardware optimized for performance or energy efficiency on specific types of tasks. For example, a mobile system-on-chip (SoC) presently may contain dedicated accelerators for graphics processing, video, wireless communication, real-time tasks, and cryptography. These accelerators can be 10 to 100 times more efficient (in performance, cost, and area) than general-purpose processors for the same tasks. Digital signal processors are another class of accelerators. These processors have a specialized instruction set optimized for math-intensive tasks.

Heterogeneous systems are not without their drawbacks. They add complexity in terms of both designing the different heterogeneous elements and the additional programming effort to decide when and how to make use of the varying resources. Symmetric and heterogeneous systems both have their places in modern systems. Symmetric multiprocessors are good for situations like large data centers that have lots of thread-level parallelism available. Heterogeneous systems are good for systems that have more varying or special-purpose workloads, such as mobile devices.

Clusters

In contrast to the other multiprocessors, processors in *clustered* multiprocessor systems each have their own local memory system instead of sharing memory. One type of cluster is a group of personal computers connected on a network and running software to jointly solve a large problem. The computers communicate using message passing instead of via shared memory. A large-scale computer cluster that has become increasingly important is the *data center*, also called *warehouse-scale computers* (WSCs), in which racks of computers and disks are networked and share power and cooling. Such systems typically include 50,000 to 100,000 computers, or *servers*, and cost $150 million.[5] Major Internet companies—including Google, Amazon, and Facebook—have driven the rapid development of data centers to support millions of users around the world. One major advantage of such clusters is that single computers can be swapped out as needed due to failures or upgrades.

In recent years, traditional servers owned by various companies are being replaced by *cloud computing*, where a smaller company rents a part of a WSC from such companies as Google and Amazon. Similarly, instead of an application running completely on a handheld device, such as a smartphone or tablet, generally called a *personal mobile device* (*PMD*), part of the application may run on the cloud to speed up computation and make data storage more efficient. This is called *software as a service* (*SaaS*). A common example of SaaS is a web search, where the database is stored on a WSC. Companies that rent cloud or web services demand both privacy (protection from other software running on the cloud) and performance. Both are realized using a virtual machine, which emulates an entire computer, including its operating system, running on a physical machine that may itself be running a

[5] D. Patterson and J. Hennessy, *Computer Organization and Design, The Hardware-Software Interface: RISC-V Edition*, Morgan Kaufmann, © 2018.

different operating system. Several virtual machines may run on one physical machine at once, with resources such as memory and I/O either partitioned or shared in time. This allows providers such as Amazon Web Services (AWS) to efficiently use physical resources, provide protection between virtual machines, and migrate virtual machines off of nonworking or low-performing computers. The *hypervisor*, also called the *virtual machine monitor* (VMM), is the software that runs the virtual machine and that maps virtual resources to physical resources. The hypervisor performs the functions typically performed by the operating system, such as managing I/O, CPU resources, and memory. The hypervisor runs between the *host* (the underlying physical hardware platform) and the operating system it is emulating. Instruction set architectures that allow the hypervisor to run directly on the hardware (as opposed to in software) are called *virtualizable*. This allows for more efficient, higher-performance virtual machines. Examples of virtualizable architectures include x86 (as of 2005), RISC-V, and IBM 370. ARMv7 and MIPS architectures are not virtualizable, but ARM did introduce virtualization extensions in 2013 with the introduction of ARMv8.

Cloud computing is also a critical part of Internet of Things (IoT) applications, where devices such as speakers, phones, and sensors connect through a network such as Bluetooth or Wi-Fi. Example IoT applications include connecting headphones to a smartphone using Bluetooth or connecting Alexa or Siri using a Wi-Fi connection. The low-cost devices (headphones, Google Home for Google Assistant, or Echo Dot for Alexa) connect through a network to higher-power servers that can stream music or, in the case of Siri and Alexa, perform speech recognition, query databases, and perform computations.

7.8 REAL-WORLD PERSPECTIVE: EVOLUTION OF RISC-V MICROARCHITECTURE*

Although RISC-V is an open-source architecture, not microarchitecture, many open-source hardware implementations are emerging, including Western Digital's SweRV cores, the SweRVolf SoC, and the PULP (Parallel Ultra Low Power) Platform. The ever-increasing RISC-V hardware implementations and supporting tools are referred to as the RISC-V ecosystem. See the Preface for information on how to access and use some of these open-source tools and hardware.

This section traces the development of RISC-V microarchitectures since RISC-V's inception in 2010. Because the base instruction set was fully described only recently, in 2017, many RISC-V chips are in development but only few are on the market as of 2021. But that is expected to change quickly as supporting tools and development cycles mature.

Most existing processor implementations are in low-level or embedded processors, but high-performance chips are on the horizon. RISC-V International (riscv.org) provides an ever-growing list of cores and SoC platforms. RISC-V commercial cores are found in SiFive's HiFive development board, Western Digital hard drives, and NVIDIA GPUs, amongst others.

As of 2021, two notable commercial RISC-V processors are SiFive's Freedom E310 core and Western Digital's open-source SweRV core, which comes in three versions. The Freedom E310 is a low-cost embedded processor used in SiFive's HiFive and Sparkfun's RED-V development boards. It runs RV32IMAC (RV32I with multiply/divide [M], atomic memory accesses [A], and compressed instructions [C] extensions) and has 8 KB of program memory, 8 KB of mask ROM for boot code, 16 KB of data SRAM, and a 16-KB two-way set-associative instruction cache. It also includes JTAG, SPI, I2C, and UART interfaces as well as a QSPI flash memory interface. The processor runs at 320 MHz and is a single-issue, in-order core with a 5-stage pipeline that has the same stages described in this chapter. Figure 7.73 shows a block diagram of the FE310-G002 processor found on the HiFive 1 Rev B board.

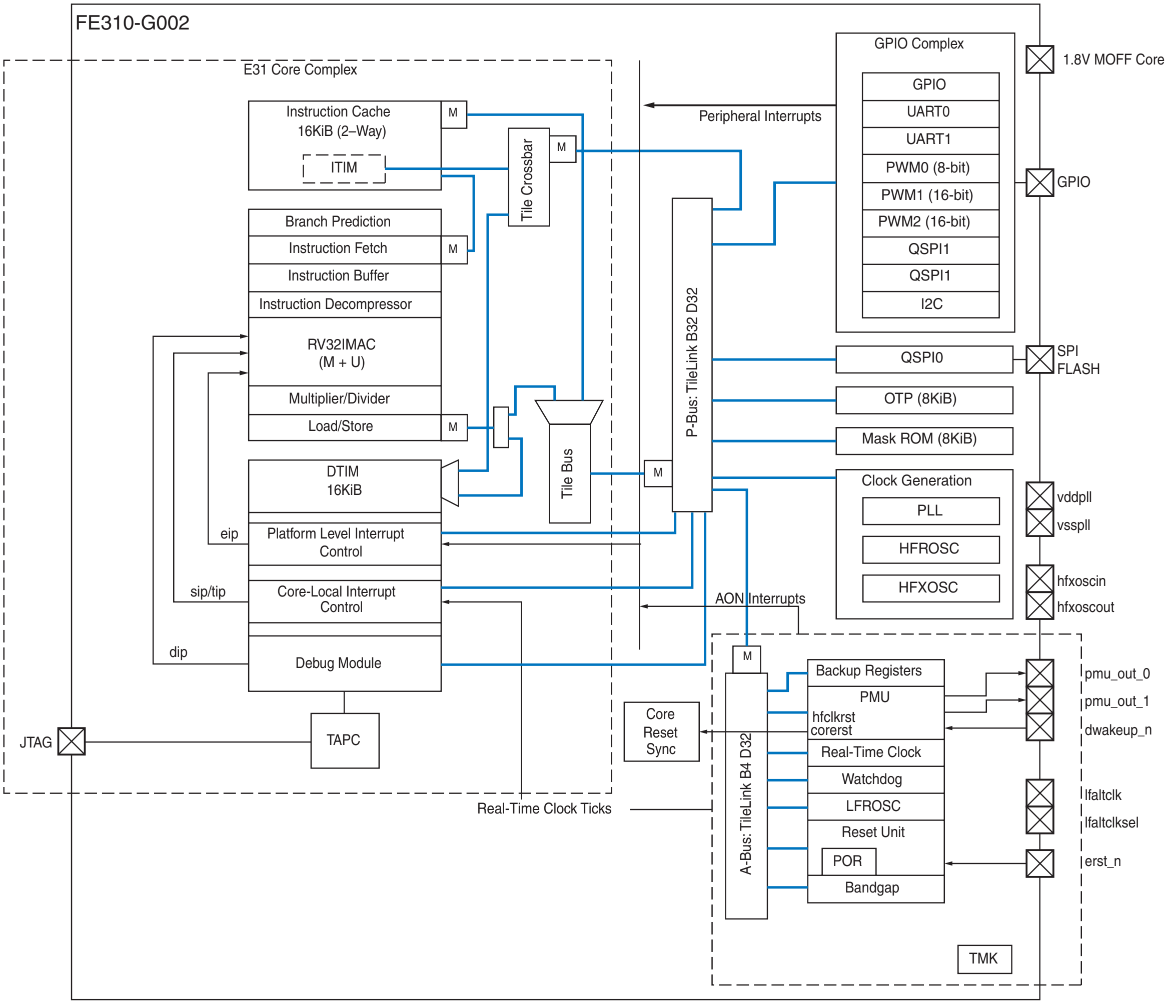


Figure 7.73 Freedom E310-G002 Block Diagram (Courtesy of SiFive Inc., *SiFive FE310-G002 Preliminary Datasheet v1p0*, ©2019)

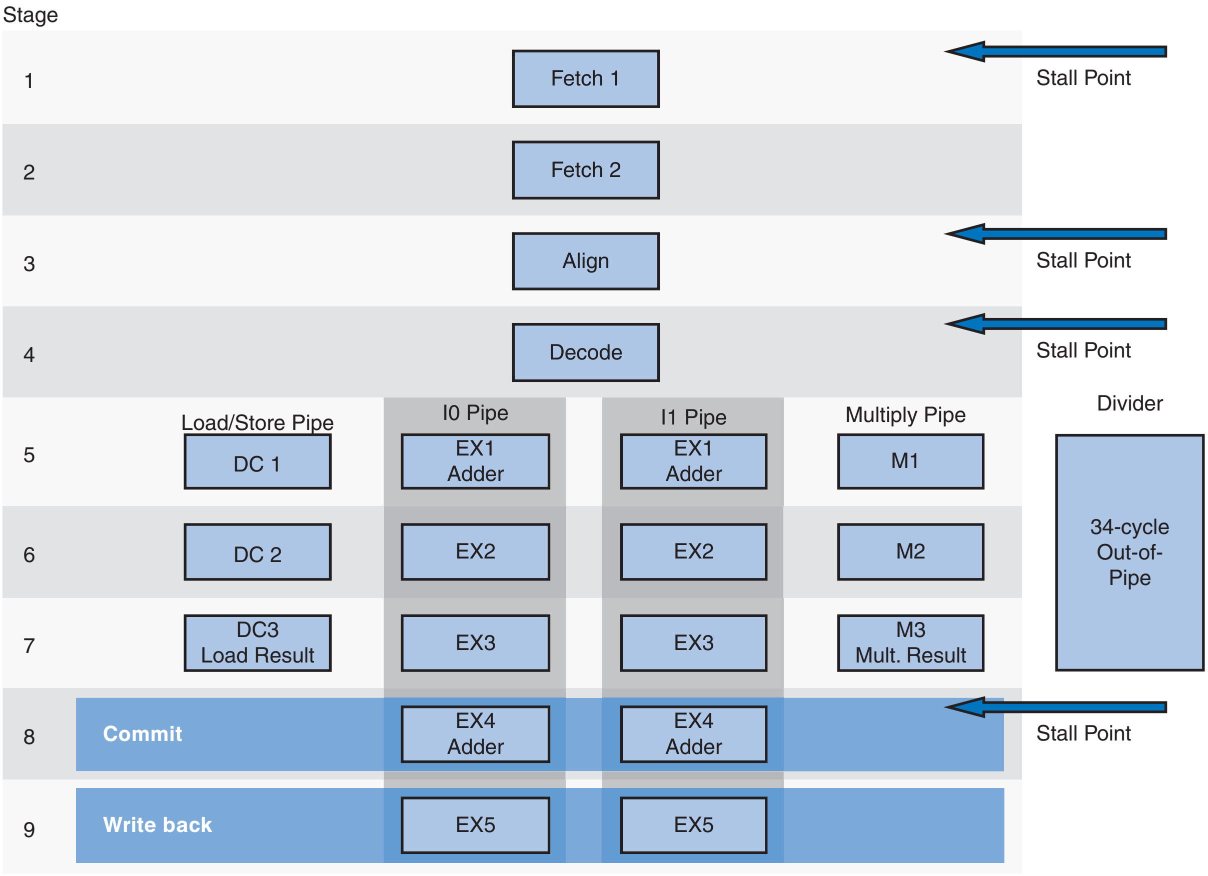


Figure 7.74 **SweRV EH1 9-stage pipeline**

(Courtesy of Western Digital Corporation, *RISC-V SweRV*^{TM} *EH1 Programmer's Reference Manual*, ©2020)

The source code (SystemVerilog) for Western Digital's open-source cores is available for download at https://github.com/chipsalliance/Cores-SweRV

The Western Digital SweRV core comes in three open-source versions: EH1, EH2, and EL2. The EH1 is a 32-bit, two-way superscalar core with a nine-stage pipeline and some support for out-of-order execution. These cores implement the RV32IMC instruction set, which includes the 32-bit base instruction set and compressed (C) and multiply/divide (M) extensions. It has a target frequency of 1 GHz using a 28-nm chip manufacturing process. The HDL can also be synthesized onto an FPGA. The EH2 core adds dual threading to the EH1. The EL2 core is a lower-performance processor targeted to embedded systems. Figure 7.74 shows the nine EH1 pipeline stages, which start with two fetch, one align, and one decode stage. The decode stage decodes up to two instructions. After this, the pipeline

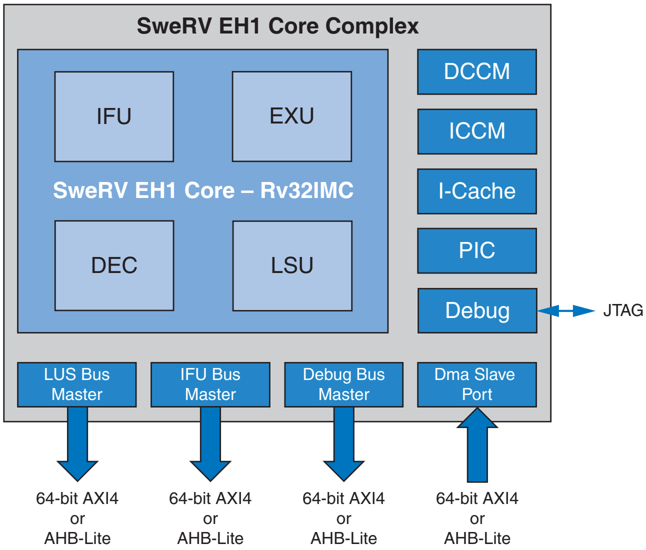


Figure 7.75 SweRV EH1 Core Complex (Courtesy of Western Digital Corporation, *RISC-V SweRV*^{TM} *EH1 Programmer's Reference Manual*, ©2020)

Robert Golla is a Senior Fellow at Western Digital and was responsible for the architecture of Western Digital's EH1, EL2, and EH2 RISC-V open-source embedded processor cores. He also architected Oracle's T4, M7, M8, and M9 out-of-order cores, Sun's N2 multithreaded core, and Motorola's embedded e500 core and contributed to Cyrix's next generation x86 M3, NXP's low-power 603 and 603e, and IBM's POWER1 and POWER2 microprocessors. He has over 50 patents related to microprocessor design.

splits into five parallel paths: a load/store path, two paths for integer instructions (such as `add`, `sub`, and `xor`), one path for multiply, and one path for divide—that is out of the pipeline because of its 34-cycle delay. The last two pipeline stages are the commit and writeback stages. The commit stage is required because of out-of-order execution and store buffers. The final stage, the writeback stage, writes results back to the register file, if needed.

Figure 7.75 shows a block diagram of the SweRV EH1 Core Complex. It includes the processor (labeled SweRV EH1 Core in the figure), instruction cache (I-Cache), data and instruction memories (DCCM and ICCM—data and instruction closely coupled memories), programmable interrupt controller (PIC), JTAG debugger interface, and memory/debug interfaces that can be configured as AXI4 or AHB-Lite busses. The processor consists of instruction fetch (IFU), decode (DEC), execute (EXU), and load/store (LSU) units. The IFU encompasses both pipeline fetch stages; DEC includes the align and decode stages; and EXU encompasses all other stages except the load/store pipeline, which is in the load/store unit (LSU). The system includes a 4-way set-associative instruction cache that can be configured as 16 to 256 KiB. The DCCM and ICCM are called *closely coupled* memories because they are low-latency on-chip memories, and they can be configured as 4 to 512 KiB.

The RISC-V FPGA (RVfpga) course from Imagination Technologies shows how to target the SweRV EH1 core to an FPGA and how to compile and run programs on the RISC-V core. This free course also provides labs and exercises that show how to expand the core to add peripherals, how to understand and modify the pipeline, superscalar execution, branch predictor, and memory hierarchy, and how to use features such as timers, interrupts, and performance counters. These labs and materials are freely available from the Imagination Technologies University Program at https://university.imgtec.com/rvfpga/.

7.9 SUMMARY

This chapter has described three ways to build processors, each with different performance, area, and cost trade-offs. We find this topic almost magical: how can such a seemingly complicated device as a microprocessor actually be simple enough to fit in a half-page schematic? Moreover, the inner workings, so mysterious to the uninitiated, are actually reasonably straightforward.

The microarchitectures have drawn together almost every topic covered in the text so far. Piecing together the microarchitecture puzzle illustrates the principles introduced in previous chapters, including the design of combinational and sequential circuits (covered in Chapters 1 through 3), the application of many of the building blocks (described in Chapter 5), and the implementation of the RISC-V architecture (introduced in Chapter 6). The microarchitectures can be described in a few pages of HDL using the techniques from Chapter 4.

Building the microarchitectures has also heavily relied on our techniques for managing complexity. The microarchitectural abstraction forms the link between the logic and architecture abstractions, forming the crux of this book on digital design and computer architecture. We also use the abstractions of block diagrams and HDL to succinctly describe the arrangement of components. The microarchitectures exploit regularity and modularity, reusing a library of common building blocks such as ALUs, memories, multiplexers, and registers. Hierarchy is used in numerous ways. The microarchitectures are partitioned into the datapath and control units, which themselves are partitioned into smaller units. Each of these units is built from logic blocks, which can be built from gates, which, in turn, can be built from transistors—all of which use the techniques developed in the first five chapters.

This chapter has compared single-cycle, multicycle, and pipelined microarchitectures for the RISC-V processor. All three microarchitectures implement the same subset of the RISC-V instruction set and have the same architectural state. The single-cycle processor is the most straightforward and has a CPI of 1.

The multicycle processor uses a variable number of shorter steps to execute instructions. It thus can reuse the ALU, rather than requiring several adders, and includes a unified memory. However, it does require several nonarchitectural registers to store results between steps. The multicycle design, in principle, could be faster because not all instructions must be equally long. In practice, it is generally slower, because it is limited by the slowest steps and by the sequencing overhead in each step.

The pipelined processor divides the single-cycle processor into five relatively fast pipeline stages. It adds pipeline registers between the stages to separate the five instructions that are simultaneously executing. It nominally has a CPI of 1, but hazards force stalls or flushes that increase the CPI slightly. Hazard resolution also costs some extra hardware and design complexity. The clock period ideally could be five times shorter than that of the single-cycle processor. In practice, it is not that short because it is limited by the slowest stage and by the sequencing overhead in each stage. Nevertheless, pipelining provides substantial performance benefits. All modern high-performance microprocessors use pipelining today.

Although the microarchitectures in this chapter implement only a subset of the RISC-V architecture, we have seen that supporting more instructions involves straightforward enhancements of the datapath and controller.

A major limitation of this chapter is that we have assumed an ideal memory system that is fast and large enough to store the entire program and data. In reality, large fast memories are prohibitively expensive. The next chapter shows how to get most of the benefits of a large fast memory by using a small fast memory that holds the most commonly used information and one or more larger but slower memories holding the rest of the information.

Exercises

Exercise 7.1 Suppose that one of the following control signals in the single-cycle RISC-V processor has a stuck-at-0 fault, meaning that the signal is always 0 regardless of its intended value. What instructions would malfunction? Why? Use the extended version of the single-cycle processor shown in Figures 7.15 and 7.16.

(a) *RegWrite*

(b) $ALUOp_1$

(c) $ALUOp_0$

(d) *MemWrite*

(e) $ImmSrc_1$

(f) $ImmSrc_0$

(g) $ResultSrc_1$

(h) $ResultSrc_0$

(i) *PCSrc*

(j) *ALUSrc*

Exercise 7.2 Repeat Exercise 7.1, assuming that the signal has a stuck-at-1 fault.

Exercise 7.3 Modify the single-cycle RISC-V processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.15 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Tables 7.3 and 7.6 to show the changes to the ALU Decoder and Main Decoder. Also, mark up any changes to the ALU, ALU Decoder, and Main Decoder diagrams (see Figure 7.16) as needed. Describe any other changes that are required.

(a) `xor`

(b) `sll`

(c) `srl`

(d) `bne`

Exercise 7.4 Repeat Exercise 7.3 for the following RISC-V instructions.

(a) `lui`

(b) `sra`

(c) `lbu`

(d) blt

(e) bltu

(f) bge

(g) bgeu

(h) jalr

(i) auipc

(j) sb

(k) slli

(l) srai

Exercise 7.5 Extend the RISC-V instruction set to include lwpostinc, which performs postindexing. lwpostinc rd, imm(rs) is equivalent to the following two instructions:

```
lw   rd, 0(rs)
addi rs, rs, imm
```

Repeat Exercise 7.3 for lwpostinc with post-indexing.

Exercise 7.6 Extend the RISC-V instruction set to include lwpreinc, which performs preindexing. lwpreinc rd, imm(rs) is equivalent to the following two instructions:

```
lw   rd, imm(rs)
addi rs, rs, imm
```

Repeat Exercise 7.3 for lwpreinc.

Exercise 7.7 Your friend is a crack circuit designer. She has offered to redesign one of the units in the single-cycle RISC-V processor to have half the delay. Using the delays from Table 7.7 on page 415, which unit should she work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be? Explain why.

Exercise 7.8 Consider the delays given in Table 7.7 on page 415. Ben Bitdiddle builds a prefix adder that reduces the ALU delay by 20 ps. If the other element delays stay the same, find the new cycle time of the single-cycle RISC-V processor and determine how long it takes to execute a benchmark with 100 billion instructions.

Exercise 7.9 Modify the HDL code for the single-cycle RISC-V processor given in Section 7.6 to handle one of the new instructions from Exercise 7.3. Enhance the testbench and test program (riscvtest.s and riscvtest.txt) given in Section 7.6.3 to test the new instruction. Add comments to indicate any changes.

Exercise 7.10 Repeat Exercise 7.9 for the new instructions from Exercise 7.4.

Exercise 7.11 Suppose one of the following control signals in the multicycle RISC-V processor has a stuck-at-0 fault, meaning that the signal is always 0 regardless of its intended value. What instructions would malfunction? Why? Refer to the multicycle datapath and control shown in Figures 7.27 and 7.45.

(a) $ResultSrc_1$

(b) $ResultSrc_0$

(c) $ALUSrcB_1$

(d) $ALUSrcB_0$

(e) $ALUSrcA_1$

(f) $ALUSrcA_0$

(g) $ImmSrc_1$

(h) $ImmSrc_0$

(i) *RegWrite*

(j) *PCUpdate*

(k) *Branch*

(l) *AdrSrc*

(m) *MemWrite*

(n) *IRWrite*

Exercise 7.12 Repeat Exercise 7.11, assuming that each signal has a stuck-at-1 fault.

Exercise 7.13 Modify the multicycle RISC-V processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Name any new control signals. Mark up copies of Figure 7.27 for changes to the datapath and Figure 7.45 for changes to the controller FSM. Describe any other changes that are required.

(a) `xor`

(b) `sll`

(c) `srl`

(d) `bne`

Exercise 7.14 Repeat Exercise 7.13 for the following RISC-V instructions.

(a) `lui`

(b) `sra`

(c) `lbu`

(d) `blt`

(e) `bltu`

(f) `bge`

(g) `bgeu`

(h) `jalr`

(i) `auipc`

(j) `sb`

(k) `slli`

(l) `srai`

Exercise 7.15 Repeat Exercise 7.5 for the multicycle RISC-V processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file? If so, show how.

Exercise 7.16 Repeat Exercise 7.6 for the multicycle RISC-V processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file? If so, show how.

Exercise 7.17 Repeat Exercise 7.7 for the multicycle RISC-V processor.

Exercise 7.18 Repeat Exercise 7.8 for the multicycle RISC-V processor. Use the instruction mix from Example 7.7.

Exercise 7.19 Your friend, the crack circuit designer, has offered to redesign one of the units in the multicycle RISC-V processor to be much faster. Using the delays from Table 7.7 on page 415, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor? Explain and show your work.

Exercise 7.20 Goliath Corp claims to have a patent on a three-ported register file. Rather than fighting Goliath in court, Ben Bitdiddle designs a new register file that has only a single read/write port (like the combined instruction and data memory). Redesign the RISC-V multicycle datapath and controller to use his new register file.

Exercise 7.21 Suppose the multicycle RISC-V processor has the component delays given in Table 7.7 on page 415. Alyssa P. Hacker designs a new register file that has 40% less power but twice as much delay. Should she switch to the slower but lower power register file for her multicycle processor design? Explain why.

Exercise 7.22 What is the CPI of the redesigned multicycle RISC-V processor from Exercise 7.20? Use the instruction mix from Example 7.7.

Exercise 7.23 How many cycles are required to run the following program on the multicycle RISC-V processor? What is the CPI of this program?

```
   addi s0, zero, 5     # result = 5
L1:
   bge  zero, s0, Done # if result <= 0, exit loop
   addi s0, s0, -1      # result = result - 1
   j    L1
Done:
```

Exercise 7.24 Repeat Exercise 7.23 for the following program.

```
   addi s0, zero, 0   # i = 0
   addi s1, zero, 0   # sum = 0
   addi t3, zero, 10  # t3 = 10
Loop:
   beq  s0, t3, L2    # if i == 10, goto L2
   add  s1, s1, s0    # sum = sum + i
   addi s0, s0, 1     # i = i + 1
   j    Loop
L2:
```

Exercise 7.25 Write HDL code for the multicycle RISC-V processor, and name the module `riscvmulti`. It should support the instructions described in this chapter: `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `beq`, and `jal`. The processor should be compatible with the top-level module below. The `mem` module is used to hold both instructions and data. Remember that you can use the building blocks from the single-cycle processor HDL in Section 7.6. Test your processor using the testbench and test program (riscvtest.s and riscvtest.txt) from Section 7.6.3. Add comments to indicate any changes.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);
  logic [31:0] ReadData;

  // instantiate processor and memories
  riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                     WriteData, ReadData);
  mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

Exercise 7.26 Extend your HDL code for the multicycle RISC-V processor from Exercise 7.25 to handle one of the new instructions from Exercise 7.14. Enhance the testbench and test program (riscvtest.s and riscvtest.txt) from Section 7.6.3 to test the new instruction. Add comments to indicate any changes.

Exercise 7.27 Repeat Exercise 7.26 for one of the new instructions from Exercise 7.13.

Exercise 7.28 The pipelined RISC-V processor is running the following code snippet. Which registers are being written and which are being read on the fifth cycle? Recall that the pipelined RISC-V processor has a Hazard Unit. You may assume a memory system that returns the result within one cycle.

```
addi s1, zero, 11 # s1 = 11
lw   s2, 25(s0)   # s2 = memory[s0+25]
add  s3, s3, s4   # s3 = s3 + s4
or   s4, s1, s2   # s4 = s1 | s2
lw   s5, 16(s2)   # s5 = memory[s2+16]
```

Exercise 7.29 Repeat Exercise 7.28 for the following RISC-V code snippet.

```
xor  s1, s2, s3  # s1 = s2 ^ s3
addi s0, s3, −4  # s0 = s3 − 4
lw   s3, 16(s7)  # s3 = memory[s7+16]
sw   s4, 20(s1)  # memory[s1+20] = s4
or   t2, s0, s1  # t2 = s0 | s1
```

Exercise 7.30 Repeat Exercise 7.28 for the following RISC-V code snippet.

```
addi s1, zero, 11 # s1 = 11
lw   s2, 25(s1)   # s2 = memory[36]
lw   s5, 16(s2)   # s5 = memory[s2+16]
add  s3, s2, s5   # s3 = s2 + s5
or   s4, s3, t4   # s4 = s3 | t4
and  s2, s3, s4   # s2 = s3 & s4
```

Exercise 7.31 Repeat Exercise 7.28 for the following RISC-V code snippet.

```
addi s1, zero, 52 # s1 = 52
addi s0, s1, −4   # s0 = s1 − 4 = 48
lw   s3, 16(s0)   # s3 = memory[64]
sw   s3, 20(s0)   # memory[68] = s3
xor  s2, s0, s3   # s2 = s0 ^ s3
or   s2, s2, s3   # s2 = s2 | s3
```

Exercise 7.32 Using a diagram similar to Figure 7.57, show the forwarding and stalls needed to execute the instructions from Exercise 7.30 on the pipelined RISC-V processor.

Exercise 7.33 Repeat Exercise 7.32 for the instructions from Exercise 7.31.

Exercise 7.34 How many cycles are required for the pipelined RISC-V processor to issue all of the instructions for the program in Exercise 7.30? What is the CPI of the processor on this program?

Exercise 7.35 Repeat Exercise 7.34 for the instructions of the program in Exercise 7.31.

Exercise 7.36 Explain how to extend the pipelined RISC-V processor to handle the load upper immediate instruction, `lui`. Name any new control signals. Mark up copies of Figure 7.61 for changes to the datapath and Tables 7.3 and 7.6 for changes to the ALU Decoder and Main Decoder. Describe any other changes that are required.

Exercise 7.37 Repeat Exercise 7.36 for the `xor` instruction.

Exercise 7.38 The pipelined processor's performance might be better if branches take place during the Decode stage rather than the Execute stage. Show how to modify the pipelined processor from Figure 7.61 to move the branch logic to the Decode stage. How do the stall, flush, and forwarding signals change? Redo Examples 7.9 and 7.10 to find the new CPI, cycle time, and overall time to execute the program.

Exercise 7.39 Your friend, the crack circuit designer, has offered to redesign one of the units in the pipelined RISC-V processor to be much faster. Using the delays from Table 7.7 on page 415, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor? Explain your answers and show your work.

Exercise 7.40 Consider the delays from Table 7.7 on page 415. Now, suppose that the ALU were 20% faster. Would the cycle time of the pipelined RISC-V processor change? What if the ALU were 20% slower? Explain your answers and show your work.

Exercise 7.41 Suppose the RISC-V pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also, assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of processing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

Exercise 7.42 Write HDL code for the pipelined RISC-V processor, and call the module `riscv`. The processor should be compatible with the top-level module below. It should support the instructions described in this chapter: `lw`, `sw`, `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `beq`, and `jal`. Remember that you can use the building blocks from the single-cycle processor HDL in Section 7.6. Modify the testbench from Section 7.6.3 to test your processor using the test program (riscvtest.s and riscvtest.txt) from that section.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
              WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule
```

Exercise 7.43 Extend the HDL for the pipelined RISC-V processor from Exercise 7.42 to handle the `xor` instruction from Exercise 7.37. Modify the testbench and test program (riscvtest.s and riscvtest.txt) from Section 7.6.3 to test your enhanced processor.

Exercise 7.44 Extend the HDL for the pipelined RISC-V processor from Exercise 7.42 to handle the `lui` instruction from Exercise 7.36. Modify the testbench and test program (riscvtest.s and riscvtest.txt) from Section 7.6.3 to test your enhanced processor.

Exercise 7.45 Design the Hazard Unit shown in Figure 7.61 for the pipelined RISC-V processor. Use an HDL to implement your design. Sketch the hardware that a synthesis tool might generate from your HDL.

Exercise 7.46 Show how to modify the RISC-V multicycle processor to take an exception if an undefined instruction is encountered. Exceptions are described in Section 6.6.2. The cause code for an undefined instruction is 2 (see Table 6.6 on page 357). Modify both the datapath (Figure 7.27) and control unit, including the Main FSM (Figure 7.45), as needed. You may assume that `mtvec` has already been written with the exception handler address.

Exercise 7.47 Repeat Exercise 7.46 for the misaligned load exception, whose cause code is 4 (see Table 6.6).

Exercise 7.48 Show how to modify the RISC-V multicycle processor to implement the privileged instruction `csrrw` (CSR read/write). Figure 7.76 shows

Assembly	Field Values					Machine Code					
	$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	
`cssrw x9, mscratch, x8`	0x340	8	1	9	115	0011 0100 0000	01000	001	01001	111 0011	(0x340414F3)
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 7.76 **Privileged instruction** `cssrw` **(CSR read/write)**

the assembly and machine code for `csrrw x9, mscratch, x8` that simultaneously copies `mscratch` into `x9` and `x8` into `mscratch`. The CSR number is encoded in the 12-bit immediate field of the I-type instruction. `mscratch` is CSR number 0x340. Any 12-bit CSR number should be able to be read/written. See Table B.8 in Appendix B for more information on privileged instruction formats. Modify both the datapath (Figure 7.27) and control unit, including the Main FSM (Figure 7.45), as needed to accommodate the `cssrw` instruction.

Exercise 7.49 Repeat Exercise 7.48 for the privileged instruction `csrrs` (CSR read and set). Figure 7.77 shows the assembly and machine code for `csrrs x7, mcause, x3` that simultaneously copies `mcause` into `x7` and puts (`mcause` | `x3`) into `mcause`. `mcause` is CSR number 0x342. Any 12-bit CSR number should be able to be read/set.

Assembly	Field Values					Machine Code					
	$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	
`csrrs x7, mcause, x3`	0x342	3	2	7	115	0011 0100 0010	00011	010	00111	111 0011	(0x3421A3F3)
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Figure 7.77 **Privileged instruction** `csrrs` **(CSR read/set)**

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 7.1 Explain the advantages of pipelined microprocessors.

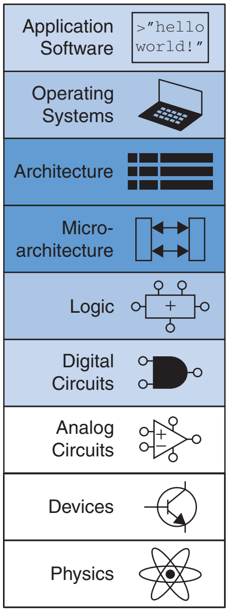
Question 7.2 If additional pipeline stages allow a processor to go faster, why don't processors have 100 pipeline stages?

Question 7.3 Describe what a hazard is in a microprocessor and explain ways in which it can be resolved. What are the pros and cons of each way?

Question 7.4 Describe the concept of a superscalar processor and its pros and cons.

Memory Systems

8



8.1 INTRODUCTION

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. However, this would be true only for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. DRAM memories are currently 10 to 100 times slower than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. The first half of this chapter investigates memory systems and considers trade-offs of speed, capacity, and cost.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle RISC-V processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

The major issues in memory system design can be broadly explained using a metaphor of books in a library. A library contains many books on the shelves. If you were writing a term paper on the meaning of dreams, you might go to the library[1] and pull Freud's *The Interpretation of Dreams* off the shelf and bring it to your cubicle. After skimming it, you might put it back and pull out Jung's *The Psychology of the Unconscious*. You might then go back for another quote from *Interpretation of Dreams*, followed by yet another trip to the stacks for Freud's *The Ego and the Id*. Pretty soon,

[1] We realize that library usage is plummeting among college students because of the Internet. But we also believe that libraries contain vast troves of hard-won human knowledge that are not electronically available. We hope that Web searching does not completely displace the art of library research.

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00008-8

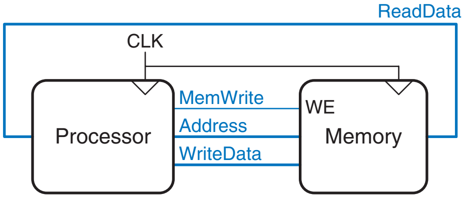


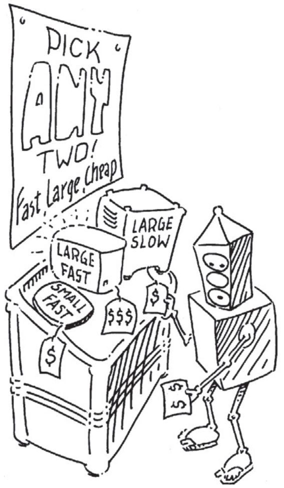
Figure 8.1 **The memory interface**

you would get tired of walking from your cubicle to the stacks. If you are clever, you would save time by keeping the books in your cubicle rather than schlepping them back and forth. Furthermore, when you pull a book by Freud, you could also pull several of his other books from the same shelf.

This metaphor emphasizes the principle, introduced in Section 6.2.1, of making the common case fast. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

The library itself makes the common case fast by using these principles of locality. The library has neither the shelf space nor the budget to accommodate all of the books in the world. Instead, it keeps some of the lesser-used books in deep storage in the basement. Also, it may have an interlibrary loan agreement with nearby libraries so that it can offer more books than it physically carries.

In summary, you obtain the benefits of both a large collection and quick access to the most commonly used books through a hierarchy of storage. The most commonly used books are in your cubicle. A larger collection is on the shelves. And an even larger collection is available, with advanced notice, from the basement and other libraries. Similarly, memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data.



Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (DRAM) and static RAM (SRAM). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory has only two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by combining a fast, small, cheap memory and a slow, large, cheap memory. The fast memory stores the most commonly used data and instructions; so, on average, the memory system appears fast. The large memory stores the remainder of the data and instructions; so, the overall capacity is large. The combination of two cheap memories is much less expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

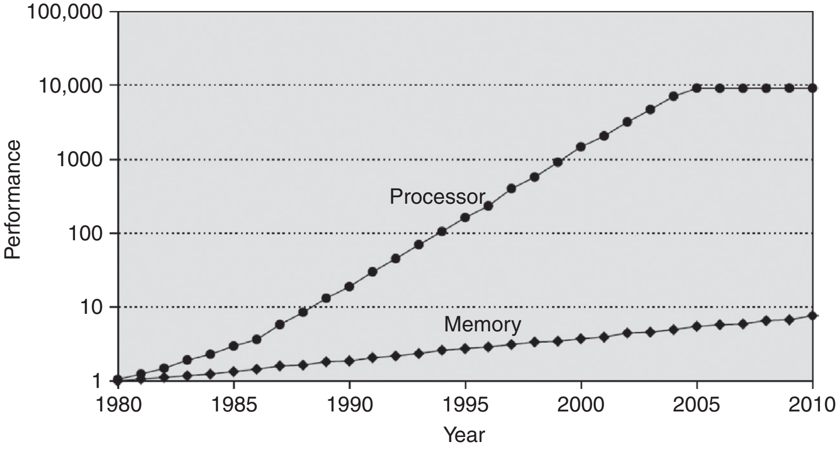


Figure 8.2 Diverging processor and memory performance. Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011

Remember that speed is characterized by both latency and throughput. Memory latency is the time to access the first byte of information. Throughput is the number of bytes per second that can be delivered. Many memories have good throughput but long latency.

Computer memory is generally built from DRAM chips. In 2021, a typical PC had a *main memory* consisting of 8 to 32 GiB of DRAM, and DRAM cost about $3 per gibibyte (GiB). DRAM prices have declined at 15% to 25% per year for the last three decades and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 25% to 50% per year, as shown in Figure 8.2. The plot shows memory (DRAM) and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. However, performance has diverged since then, with memories badly lagging.[2]

DRAM could keep up with processors in the 1970's and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond). DRAM throughput is good, on the order of 30 GB/s.

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed because SRAM is inherently faster than DRAM and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip.

[2] Although recent single-processor performance has remained approximately constant, as shown in Figure 8.2 for the years 2005 to 2010, the increase in multicore systems (not depicted on the graph) only worsens the gap between processor and memory performance.

In 2021, on-chip SRAM costs were on the order of $100/GiB, but the cache is relatively small (kibibytes to several mebibytes), so the overall cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data." SRAM latency ranges from a few tenths of a nanosecond for a 16 KiB cache to several nanoseconds for a 4 MiB cache. Throughput can reach hundreds of GB/s.

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that does not fit in main memory. In 2021, a hard disk drive (HDD), built using magnetic storage, cost less than $0.03/GB and had an access time of about 5 to 10 ms. Throughput is on the order of 100 MB/s for large files down to 1 MB/s for random accesses to small (4 KiB) files. Hard disk costs have decreased at 60% per year but access times scarcely improved. Solid state drives (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs. SSDs have been used by niche markets for over two decades, and they were introduced into the mainstream market in 2007. SSDs overcome some of the mechanical failures of HDDs, but they cost 3 to 4 times as much at $0.10/GB. Since SSDs hit the market, the price difference between them and HDDs has shrunk, and the popularity of SSDs over HDDs has increased accordingly. SSDs have access times of less than 0.1 ms. Throughput can be 500 to 3,000 MB/s for large files down to 50 to 250 MB/s for 4 KiB files.

The hard drive provides an illusion of more capacity than actually exists in the main memory. It is thus called *virtual memory*. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called *physical memory*, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the

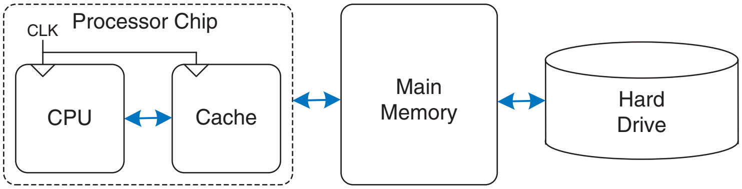
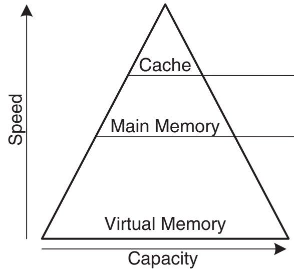


Figure 8.3 A typical memory hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2–3	100+
DRAM	$3	10–50	30
SSD	$0.10	20,000	0.05–3
HDD	$0.03	5,000,000	0.001–0.1

Figure 8.4 Memory hierarchy components, with typical characteristics in 2021

data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs, access times, and bandwidth in 2021 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems.

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time*. Miss and hit rates are calculated as:

$$\begin{aligned} \textit{Miss Rate} &= \frac{\textit{Number of misses}}{\textit{Number of total memory accesses}} = 1 - \textit{Hit Rate} \\ \textit{Hit Rate} &= \frac{\textit{Number of hits}}{\textit{Number of total memory accesses}} = 1 - \textit{Miss Rate} \end{aligned} \tag{8.1}$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2,000 data access instructions (loads or stores) and 1,250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

Average memory access time (*AMAT*) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \tag{8.2}$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates in Table 8.1?

Solution The average memory access time is 1 + 0.1(100) = 11 cycles.

Table 8.1 **Access times and miss rates**

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

Gene Amdahl, 1922–2015
Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation and later went on to found three companies, including one called Amdahl Corporation in 1970.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's instructions are loads and stores, a tenfold memory system improvement means only a 1.82-fold improvement in program performance. This general principle is called *Amdahl's*

Law, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, *C*. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is data found? and (3) What data is replaced to make room for new data when the cache is full?

Cache: a hiding place especially for concealing and preserving provisions or implements.
—Merriam Webster Online Dictionary. 2021. www.merriam-webster.com

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (*C*), number of sets (*S*), block size (*b*), number of blocks (*B*), and degree of associativity (*N*).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

8.3.1 What Data is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations.

Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block* or *cache line*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the same variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How is Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider a RISC-V memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache

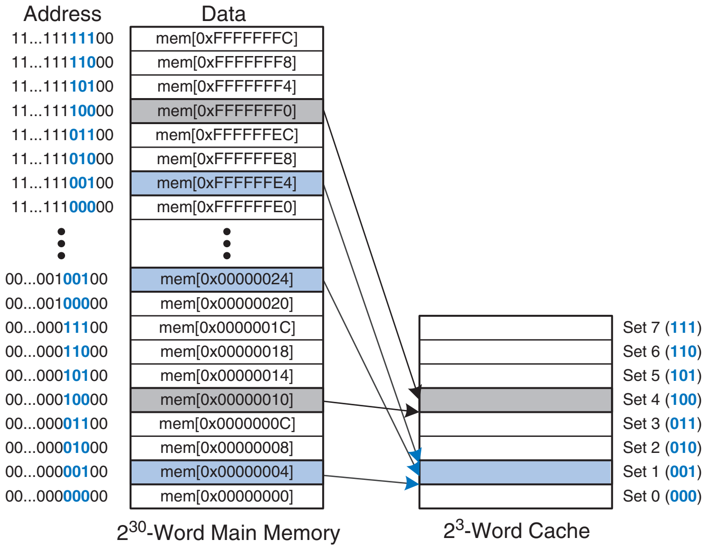


Figure 8.5 Mapping of main memory to a direct mapped cache

has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, …, 0xFFFFFFE4 all map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, …, 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, …, 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the *byte offset* because they indicate the byte within the word. The next three bits are called the *set bits* because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the

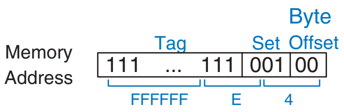


Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all l's.

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks

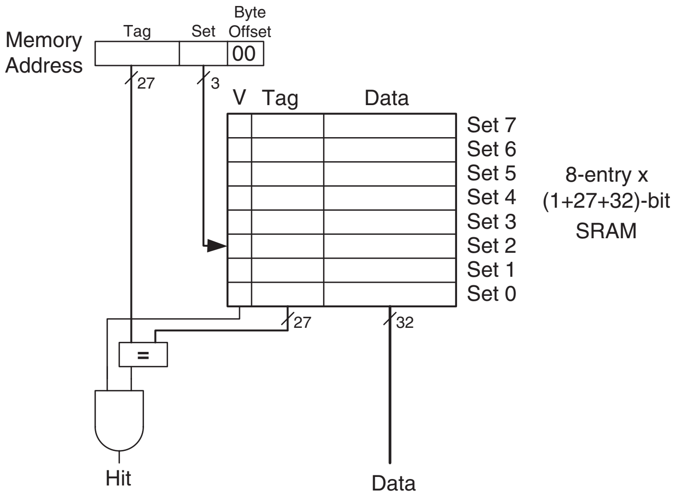


Figure 8.7 Direct mapped cache with 8 sets

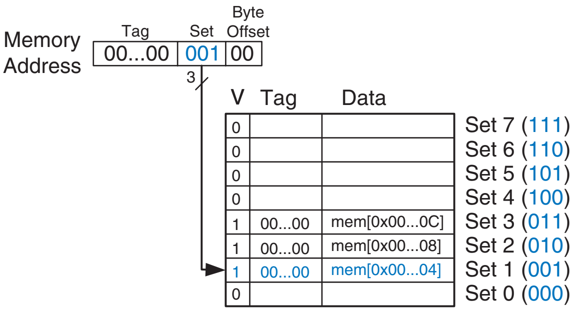
the tag and valid bits. If the tag matches the most significant 27 bits of the requested address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in RISC-V assembly code. Assume that the cache is initially empty. What is the miss rate?

```
       addi s0, zero, 5
       addi s1, zero, 0
LOOP:  beq  s0, zero, DONE
       lw   s2, 4(s1)
       lw   s3, 12(s1)
       lw   s4, 8(s1)
       addi s0, s0, −1
       j    LOOP
DONE:
```

Solution The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.



V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

Figure 8.8 Direct mapped cache contents

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict. Example 8.7 illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 0x4(s1)
        lw   s4, 0x24(s1)
        addi s0, s0, −1
        j    LOOP
DONE:
```

Solution Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then, data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict and the miss rate is 100%.

Multiway Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the *N* blocks

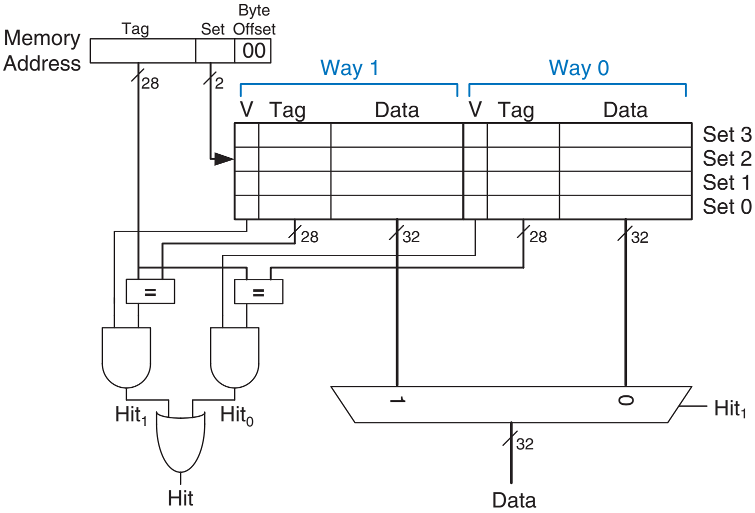


Figure 8.9 **Two-way set associative cache**

in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the *degree of associativity* of the cache.

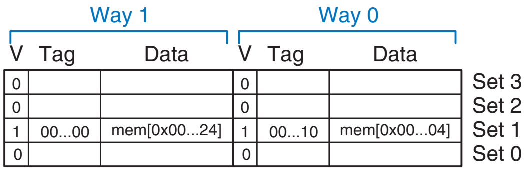
Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.



Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...24]	1	00...10	mem[0x00...04]	Set 1
0			0			Set 0

Figure 8.10 **Two-way set associative cache contents**

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made because the data could be in any block. Similarly, an 8:1

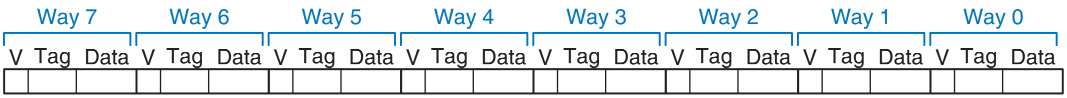

Way 7
Way 6
Way 5
Way 4
Way 3
Way 2
Way 1
Way 0
V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data

Figure 8.11 Eight-block fully associative cache

multiplexer chooses the proper data if a hit occurs. Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a C = 8-word direct mapped cache with a b = 4-word block size. The cache now has only $B = C/b$ = 2 blocks. A direct mapped cache has one block in each set, so this cache

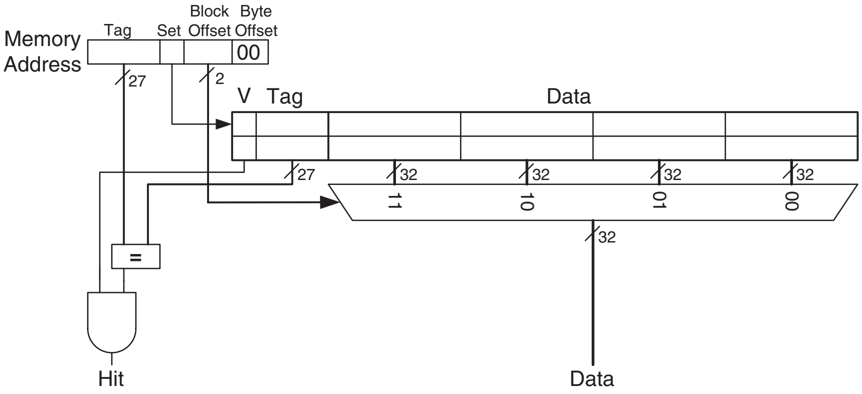


Figure 8.12 Direct mapped cache with two sets and a four-word block size

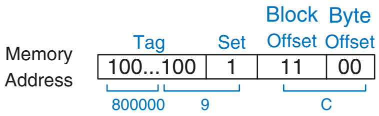


Figure 8.13 **Cache fields for address 0x8000009C when mapping to the cache of Fig. 8.12**

is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block and the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

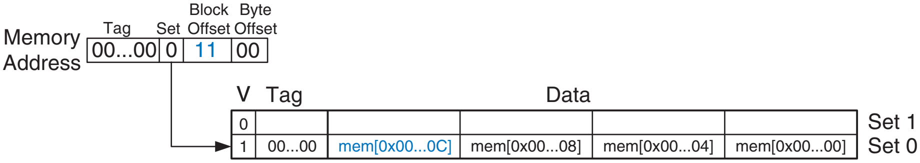


Figure 8.14 **Cache contents with a block size *b* of four words**

Putting it All Together

Caches are organized as two-dimensional arrays. The rows are called *sets*, and the columns are called *ways*. Each entry in the array

Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of two. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity N usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size b takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and, therefore, could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (*LRU*) replacement policy.

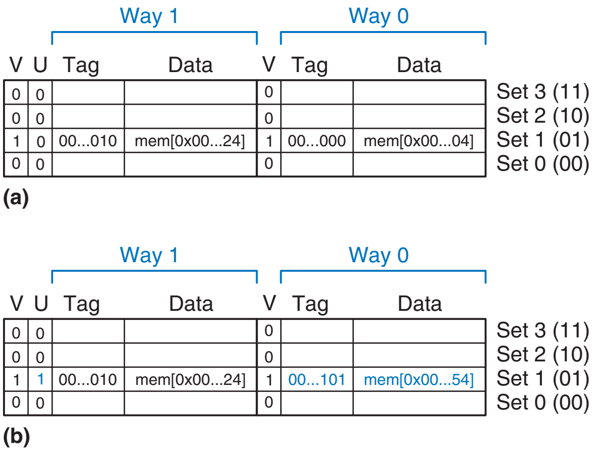
In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used. Upon replacement, the new block replaces a random block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
addi t0, zero, 0
lw   s1, 0x4(t0)
lw   s2, 0x24(t0)
lw   s3, 0x54(t0)
```

Solution The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). $U = 0$ indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b). The use bit U is set to 1 to indicate that data in way 1 was the least recently used.



V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

(a)

V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)

(b)

Figure 8.15 Two-way associative cache with LRU replacement

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

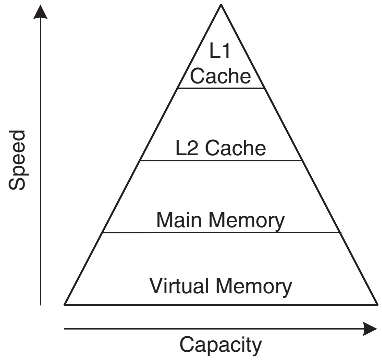


Figure 8.16 **Memory hierarchy with two levels of cache**

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and, therefore, have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use at least two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger—and, therefore, slower—than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows: 1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles.

The L2 miss rate is high because it receives only the "hard" memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more types of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

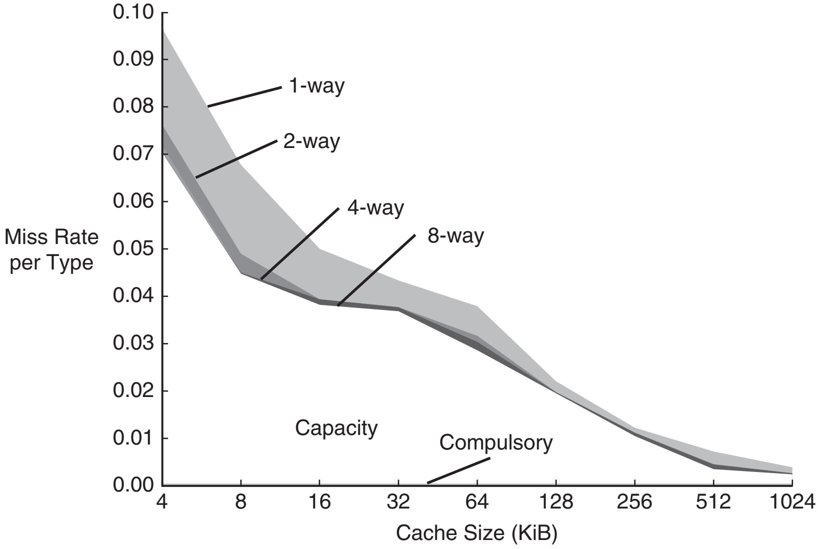


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

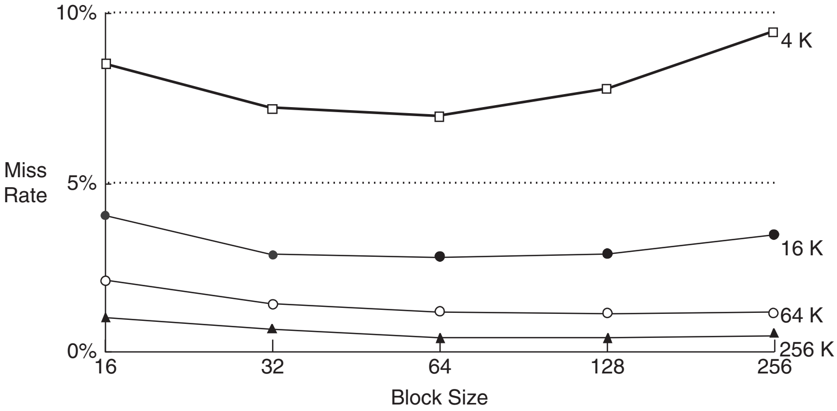


Figure 8.18 **Miss rate versus block size and cache size on SPEC92 benchmark** Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4 KiB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
addi t5, zero, 0
sw   t1, 0(t5)
sw   t2, 12(t5)
sw   t3, 8(t5)
sw   t4, 4(t5)
```

Solution All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid-state storage as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard drive made of magnetic storage, also called a *hard disk*, with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor. Hard disk drives are increasingly being replaced by solid-state drives because reading is orders of magnitude faster (see Figure 8.4) and they are not as susceptible to mechanical failures.

The objective of adding a hard drive to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 16 GiB of DRAM, for example, could effectively provide 128 GiB of memory using the hard drive. This larger 128 GiB memory is called *virtual memory*, and the smaller 16 GiB main memory is called *physical*

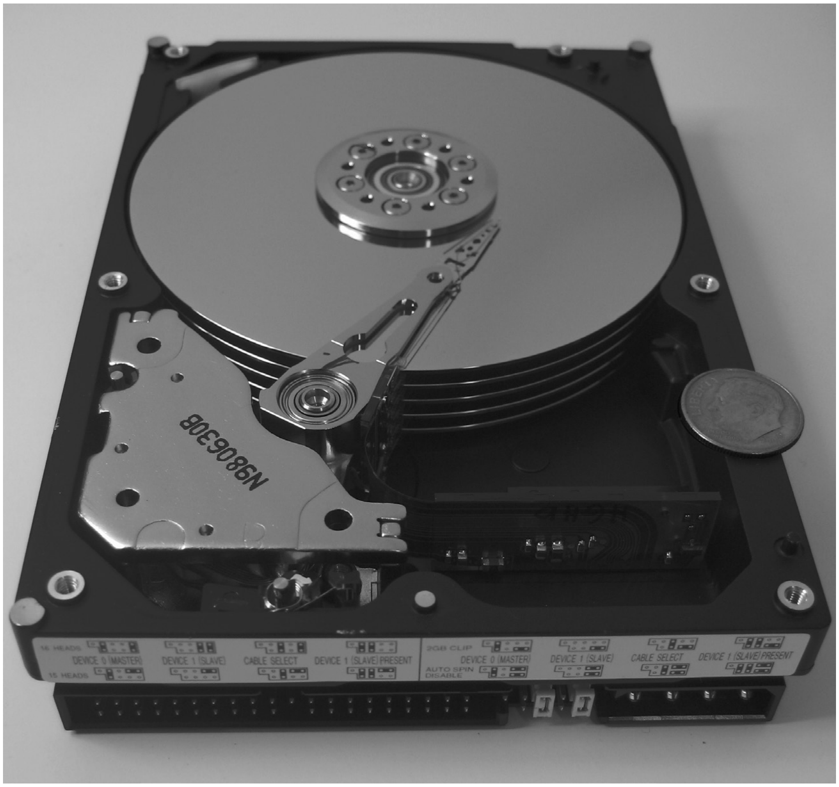


Figure 8.19 **Hard disk**

memory. We will use the term *physical memory* to refer to main memory throughout this section.

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GiB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.3 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KiB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the hard drive. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the hard drive. The process of determining the physical

Table 8.3 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

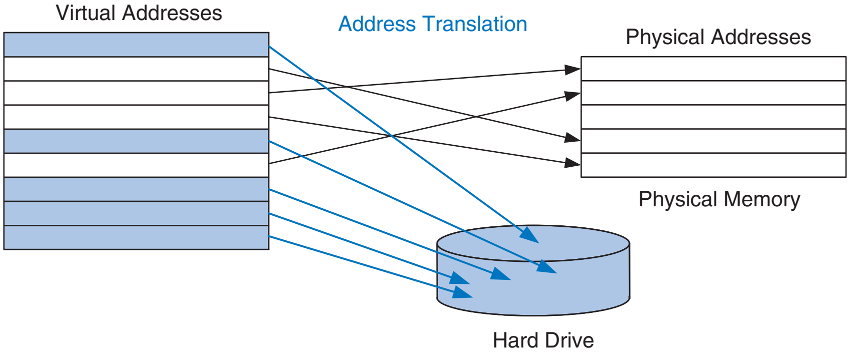


Figure 8.20 Virtual and physical pages

address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault* occurs and the operating system (OS) loads the page from the hard drive into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GiB of virtual memory and 128 MiB of physical memory divided into 4 KiB pages. MIPS accommodates 32-bit addresses. With a 2 GiB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128 MiB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

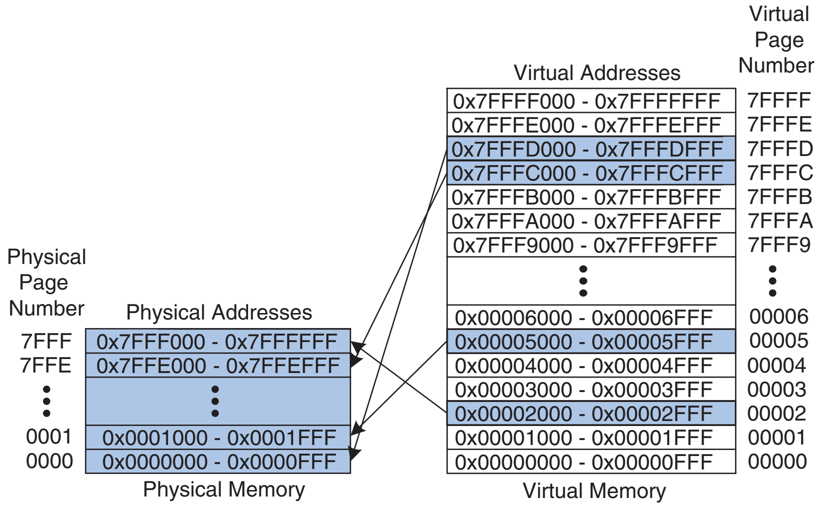

Virtual Page Number
Virtual Addresses
0x7FFFF000 - 0x7FFFFFFF 7FFFF
0x7FFFE000 - 0x7FFFEFFF 7FFFE
0x7FFFD000 - 0x7FFFDFFF 7FFFD
0x7FFFC000 - 0x7FFFCFFF 7FFFC
0x7FFFB000 - 0x7FFFBFFF 7FFFB
0x7FFFA000 - 0x7FFFAFFF 7FFFA
0x7FFF9000 - 0x7FFF9FFF 7FFF9
0x00006000 - 0x00006FFF 00006
0x00005000 - 0x00005FFF 00005
0x00004000 - 0x00004FFF 00004
0x00003000 - 0x00003FFF 00003
0x00002000 - 0x00002FFF 00002
0x00001000 - 0x00001FFF 00001
0x00000000 - 0x00000FFF 00000
Virtual Memory
Physical Page Number
Physical Addresses
7FFF 0x7FFF000 - 0x7FFFFFF
7FFE 0x7FFE000 - 0x7FFEFFF
0001 0x0001000 - 0x0001FFF
0000 0x0000000 - 0x0000FFF
Physical Memory

Figure 8.21 **Physical and virtual pages**

Because the page size is 4 KiB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

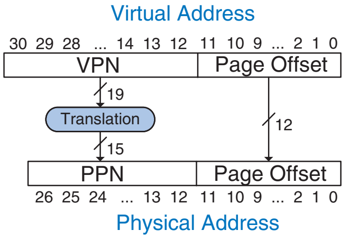


Figure 8.22 **Translation from virtual address to physical address**

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on the hard drive.

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Figure 8.23 The page table for Figure 8.21

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

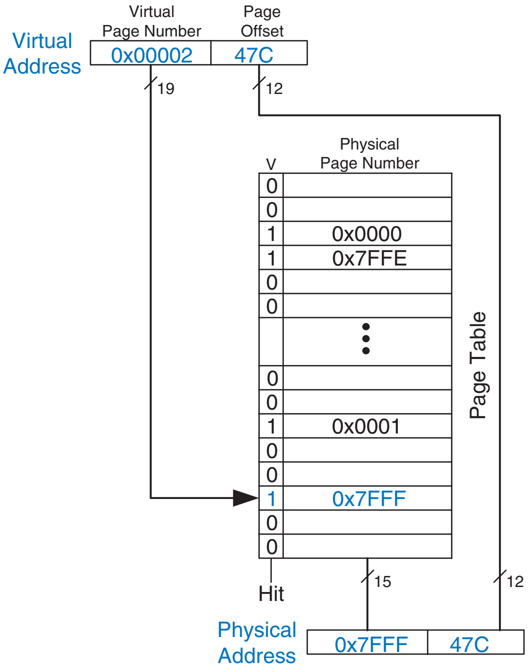


Figure 8.24 Address translation using the page table

The page table can be stored anywhere in physical memory at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose that the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFFD.

Solution Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

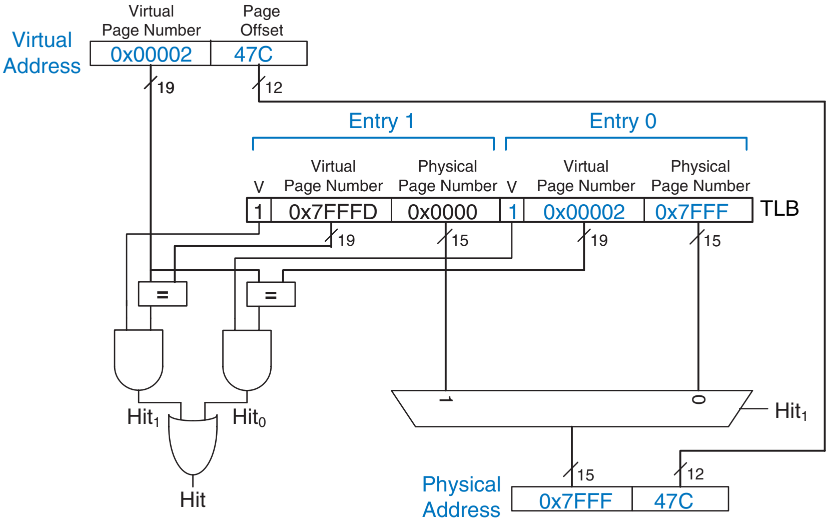


Figure 8.25 Address translation using a two-entry TLB

8.4.4 Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The OS adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be impractical. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called *paging*, and the hard drive in a virtual memory system is sometimes called *swap space*. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from the hard drive. When a physical page is paged out, it needs to be written back to the hard drive only if its dirty bit is 1; otherwise, the hard drive already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all of the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to page out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GiB virtual memory with 4 KiB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MiB.

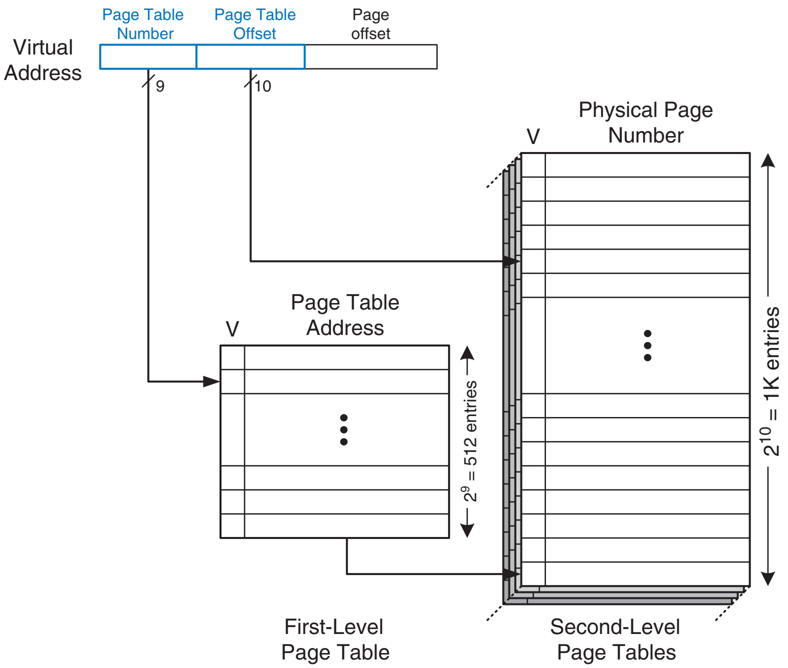


Figure 8.26 **Hierarchical page tables**

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be paged out to the hard drive so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that it must be fetched from the hard drive when *V* is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of 2^{12} = 4 KiB.

In Figure 8.26, the 19-bit virtual page number is broken into 9 and 10 bits to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has 2^9 = 512 entries. Each of these 512 second-level page tables has 2^{10} = 1 Ki entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the

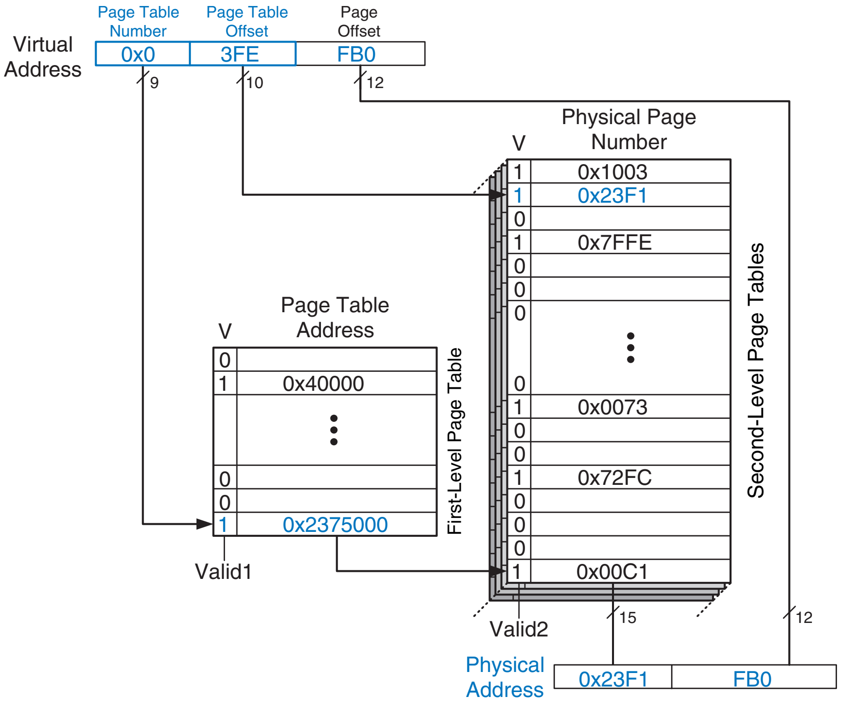


Figure 8.27 Address translation using a two-level page table

hierarchical page table uses only (512 × 4 bytes) + 2 × (1 Ki × 4 bytes) = 10 KiB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MiB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

8.5 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies—such as DRAM, SRAM, and hard drives—offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory. Most processors also use the memory interface to communicate with input/output (I/O) devices. This is called *memory-mapped I/O*. Programs use load and store operations to access I/O devices, which are discussed in Chapter 9, an online supplemental chapter available on this book's companion website (see the Preface).

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program a RISC-V processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But, at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. Hackers and governments wage undeclared cyberwarfare, attacking industrial infrastructure and financial networks. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16 KiB direct mapped cache always has a lower miss rate than an 8 KiB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: *b*, block size given in numbers of words; *S*, number of sets; *N*, number of ways; and *A*, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, *C*?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are *S* and *N* for a fully associative cache of capacity *C* words with block size *b*?

(d) What is *S* for a direct mapped cache of size *C* words and block size *b*?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `lw` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring start-up effects (i.e., compulsory misses).

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 1$ word

(c) two-way set associative cache, $b = 1$ word

(d) direct mapped cache, $b = 2$ words

Exercise 8.10 Repeat Exercise 8.9 for the following repeating sequence of `lw` addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 2$ words

(c) two-way set associative cache, $b = 2$ words

(d) direct mapped cache, $b = 4$ words

Exercise 8.11 Suppose you are running a program with the following data access pattern (given in hexadecimal). The pattern is executed only once.

0 8 10 18 20 28

(a) If you use a direct mapped cache with a cache size of 1 KiB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

(i) Increasing the degree of associativity to 2.

(ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.12 You are building an instruction cache for a RISC-V processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.13 Consider a cache with the following parameters:
N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits,
C (cache size) = 32 Ki words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose that each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.28 and a small number of two-input logic gates. The cache design must include tag storage, data

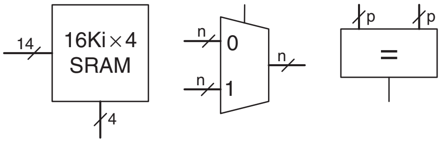


Figure 8.28 **Building blocks**

storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n or p bits wide, respectively), but the SRAM blocks must be 16Ki × 4 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

Exercise 8.14 You've joined a hot new Internet start-up to build wristwatches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.29. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

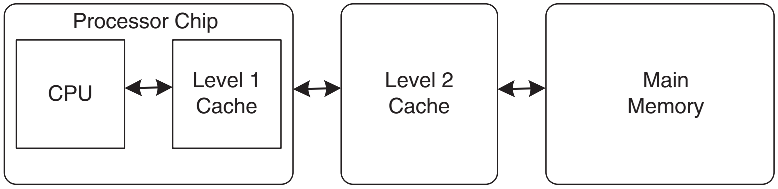


Figure 8.29 **Computer system**

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.4. The DRAM has an access time of t_m and a size of 512 MiB.

Table 8.4 **Memory characteristics**

Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256 Ki

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.15 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other less common replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.16 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the RISC-V multicycle processor from Figure 7.44 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the nonideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions[3]. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

[3] Data from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011. Used with permission.

Exercise 8.17 Repeat Exercise 8.16 with the following parameters.

(a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Taking the nonideal memory system into account, what is the average CPI for this benchmark?

(d) Now, suppose that the instruction cache is also nonideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.18 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte (tebibyte)*, 2^{50} bytes = 1 *petabyte (pebibyte)*, and 2^{60} bytes = 1 *exabyte (exbibyte)*.

Exercise 8.19 A supercomputer designer chooses to spend $1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

Exercise 8.20 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space but are limited to only 8 MiB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KiB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this "direct mapping" a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (e). Suppose that you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.21 Consider a virtual memory system that can address a total of 2^{50} bytes. You have unlimited hard drive space but are limited to 2 GiB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KiB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) How many page table entries will the page table contain?

(f) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(g) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.22 You decide to speed up the virtual memory system of Exercise 8.20 by using a translation lookaside buffer (TLB). Suppose that your memory system has the characteristics shown in Table 8.5. The TLB and cache miss rates indicate how

Table 8.5 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
Cache	1	2%
Main memory	100	0.0003%
Hard drive	1,000,000	0%

often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.23 You decide to speed up the virtual memory system of Exercise 8.21 by using a translation lookaside buffer (TLB) with 128 entries.

(a) How big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(b) Sketch the TLB. Clearly label all fields and dimensions.

(c) What size SRAM would you need to build the TLB described in part (b)? Give your answer in terms of depth × width.

Exercise 8.24 Suppose that the RISC-V multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.25 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KiB) pages. Each page table entry contains a physical page number, a valid bit (V), and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KiB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.26 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.27 Suppose that you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard drive affect performance?

(c) How does the size of your PC's physical memory affect performance?

Interview Questions

The following exercises present questions that have been asked on interviews.

Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

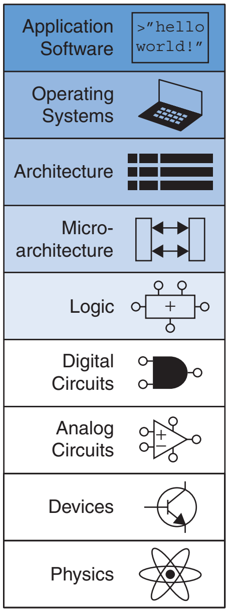
Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.

e9 Embedded I/O Systems

9.1 INTRODUCTION

Input/Output (I/O) systems are used to connect a computer with external devices called *peripherals*. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems. It shows how to use SparkFun's RED-V RedBoard, which has a RISC-V microcontroller, to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers and pulse-width modulation (PWM). Section 9.4 gives examples of interfacing with other common devices, such as character LCDs, VGA monitors, Bluetooth radios, and motors.



9.2 MEMORY-MAPPED I/O

Recall from Section 6.5.1 that a portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that physical addresses in the range 0x20000000 to 0x20FFFFFF are used for I/O. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped* I/O.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure e9.1 shows the hardware needed to support two memory-mapped I/O devices. An address decoder determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00019-2

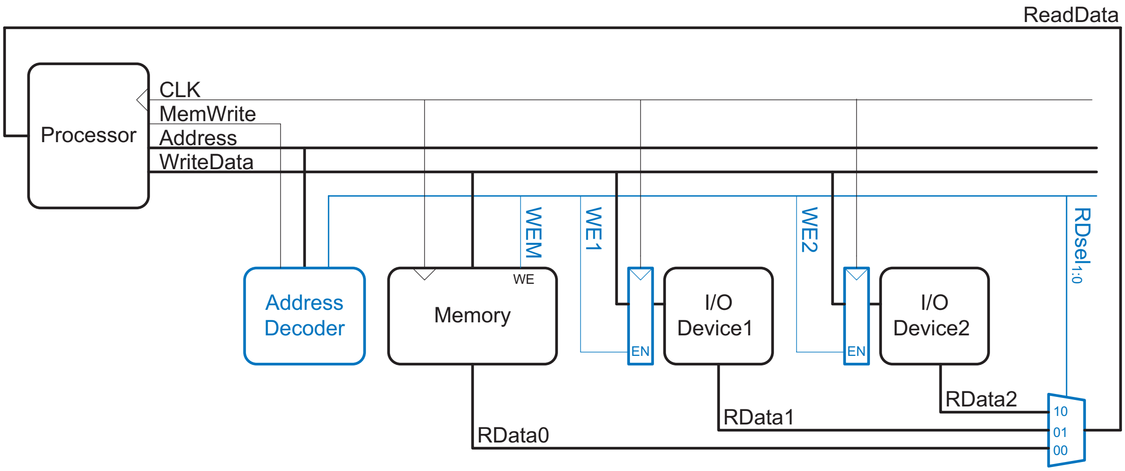


Figure e9.1 **Support hardware for memory-mapped I/O**

the hardware. The ReadData multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.

Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose that I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the RISC-V assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution The following RISC-V assembly code writes the value 7 to I/O Device 1. The `.equ` assembler directive replaces the named symbol with the given value. So, the `li s1,ioadr` instruction becomes `li s1,0x20001000`.

```
.equ ioadr   0x20001000

     li s0,7
     li s1,ioadr
     sw s0,0(s1)
```

The address decoder detects address 0x20001000 and *MemWrite* = 1, so it asserts *WE1*, the write enable for Device 1's register. At the next clock edge, the value on the *WriteData* bus, 7, is written into the register, whose output connects to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following RISC-V assembly code.

```
lw s0, 0(s1)
```

Embedded processors are so named because they are typically embedded within a larger system (such as a toy or an automobile) and have a limited user interface. In contrast, processors found in PCs have interfaces such as keyboards and screens that make them accessible to program or run applications. But all types of processors are essentially the same—they all execute instructions. Only the interfaces and peripheral devices used by embedded and traditional processors differ.

The address decoder detects the address 0x20001000, so it sets $RDsel_{1:0}$ to 01. The multiplexer thus selects *RData1*, the read data from Device 1, and connects it to the *ReadData* bus, the value of which is then loaded into `s0` in the processor.

The addresses associated with I/O devices are often called *I/O registers* because they may correspond with physical registers in the I/O device like those shown in Figure e9.1.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware, including the addresses and behavior of the memory-mapped I/O registers. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

According to IC Insights, approximately 24 billion microcontrollers were sold in 2020, and the market is forecast to grow at 10% per year through 2029. The average price of a standalone microcontroller is about 60 cents, and an 8-bit microcontroller can be integrated on a system-on-chip (SoC) for less than a tenth of a penny. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 100 or more microcontrollers in an average new car in 2021.

Automobiles are the largest and fastest-growing market for microcontrollers, followed by consumer electronics, industrial systems, medical devices, and military applications. 16-bit microcontrollers account for the most revenue in 2020, but 32-bit microcontrollers are increasing in market share because of their greater capabilities.

Leading microcontroller manufacturers are Infineon, Microchip, NXP, Renesas, STMicroelectronics, and Texas Instruments. Leading architectures include the 8051, AVR, PIC, and ARM. ARM holds a near-monopoly as the application processor for 90% of mobile devices. However, RISC-V is gaining great interest as a new and open-source architecture.

9.3 EMBEDDED I/O SYSTEMS

Embedded systems use a processor to control interactions with the physical environment. They are typically built around microcontroller units (MCUs) which combine a microprocessor with a set of easy-to-use peripherals such as general-purpose digital and analog I/O pins, serial ports, timers, etc. Microcontrollers are generally inexpensive and are designed to minimize system cost and size by integrating most of the necessary components onto a single chip. Most are smaller and lighter than a dime, consume milliwatts of power, and range in cost from a few dimes up to several dollars. Microcontrollers are classified by the size of data that they operate on. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

9.3.1 RED-V Board

For the sake of concreteness, this section will illustrate embedded system I/O in the context of a real system. Specifically, we will focus on the FE310-G002 system-on-chip (SoC) from SiFive, which contains a 320 MHz 32-bit RISC-V processor that implements the RV32IMAC architecture—that is, the base 32-bit integer instruction set (RV32I) plus the multiply/divide (M), atomic memory accesses (A), and compressed 16-bit instructions (C) extensions. This MCU is available on the HiFive development board from SiFive as well as on a set of third-party development boards such as the RED-V series from SparkFun (available in both Arduino and Thing Plus footprints). The I/O interfaces described in each subsection will be followed by specific examples that run on the FE310. All of the examples have been tested on SparkFun's RED-V

RedBoard and could be readily run on the HiFive development board or adapted to the RED-V Thing Plus Board.

The RED-V RedBoard is called simply *the RED-V* throughout this chapter.

Figure e9.2(a) shows SparkFun's RED-V RedBoard, which is available for less than $40 and is 2.7" × 2.1". The figure also shows each pin's signal names, which we describe throughout this section. The development board can be powered from a 5 V USB power supply or from a 7 to 15 V DC source via the barrel jack. The FE310-G002 on board is powered by 3.3 V and 1.8 V on-board regulators. The FE310-G002 has a 16-KiB L1 Instruction Cache and a 16-KiB Data SRAM Scratchpad. The SparkFun development board also has 32 MiB of off-chip flash storage accessible via a serial peripheral interface (SPI) that can be used to store programs and data.

This book's companion materials (see the Preface) include laboratory exercises that use the RED-V board.

Figure e9.2(b) shows the RED-V Thing Plus, which has capabilities similar to the RED-V RedBoard but in a smaller form factor (2.3" × 0.9") that fits on a breadboard for easy interfacing. The I/O pins are numbered differently than on the RedBoard and are difficult to read on the silk screen, but they are labeled in Figure e9.2(b).

The RED-V RedBoard form factor is designed around an Arduino R3 footprint in an effort to preserve as much compatibility as possible with the many Arduino shields available in this footprint. All 19 configurable I/O signals are accessible via header pins and operate at 3.3 V. The header also provides 3.3 V, 5 V, and ground to conveniently power small devices attached to the RedBoard, but the maximum total current is 50 mA from the 3.3 V supply and ~300 mA from the 5 V supply.

Caution: Connecting 5 V to one of the 3.3 V I/Os may damage the I/O and possibly the entire FE310. If you probe the I/O pins with a voltmeter, beware that you do not accidentally make contact between VUSB or VBAT and a nearby pin!

Maintaining compatibility with the Arduino R3 footprint results in multiple names for each pin: the silkscreen (text printed on the board) lists the standard Arduino pin numbers, but the RED-V pinout documented in Figure e9.2 shows both the Arduino pin numbers and the corresponding FE310 GPIO (general-purpose I/O) pin numbers. For

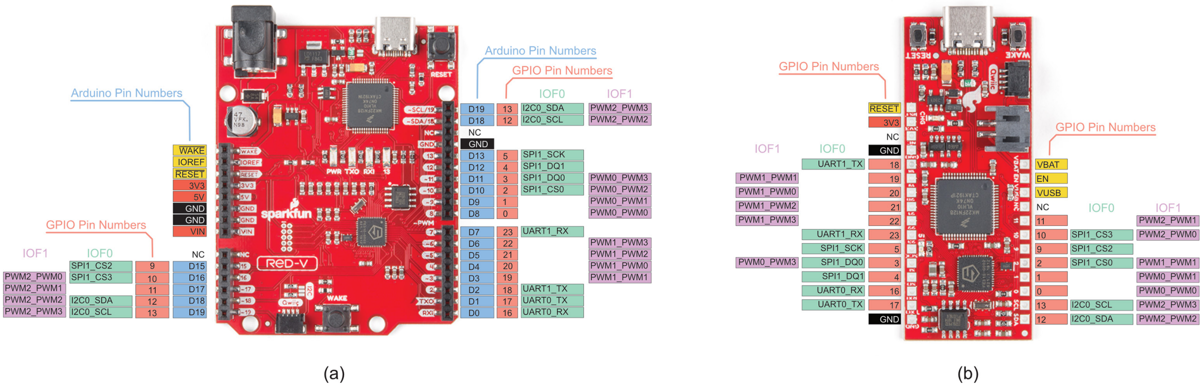


Figure e9.2 (a) RED-V RedBoard; (b) RED-V Thing Plus Board
(Photos courtesy of SparkFun used under CC BY 2.0)

example, as shown in Figure e9.2, GPIO5 (FE310 pin 5) corresponds to D13 (Arduino pin 13). The RED-V Thing Plus board lacks the Arduino-compatible pin naming but, thus, also avoids having multiple pin numbers for a single pin. Also notice in Figure e9.2 that some GPIO pins have multiple purposes. For example, GPIO18 (D2) can also act as the transmit line for UART 1 (UART1_TX), as will be described later.

On both boards, GPIO5 is connected to a blue LED. This pin is labeled 13 (i.e., D13) on the RedBoard and 5 on the Thing Plus board.

This section begins by describing the FE310-G002 SoC and describing a general device driver for memory-mapped I/O. The remainder of this section illustrates how embedded systems perform general-purpose digital, analog, and serial I/O.

SiFive was founded in 2015 by three researchers from the University of California, Berkeley: Krste Asanović, Yunsup Lee, and Andrew Waterman. SiFive's vision is to make custom silicon development faster and more affordable than ever before. Focused around the open RISC-V instruction set architecture (ISA), they have developed a platform that enables system-level design of custom chips. More information, including the FE310-G002 datasheet, can be found at sifive.com.

9.3.2 FE310-G002 System-on-Chip

The FE310-G002 SoC is a powerful yet inexpensive microcontroller chip designed by SiFive. It includes a RISC-V microprocessor with a 5-stage pipeline similar to the one described in Chapter 7 and many I/O peripherals. The FE310 is packaged in a 48-lead, quad flat no-leads package. SiFive publishes a datasheet that describes many features and I/O registers; this chapter discusses only a subset of those features.

Table e9.1 shows the FE310 memory map. Upon start-up, the processor begins executing code from external flash memory at address 0x20000000. The memory map has room for up to 512 MiB of external flash, although current RED-V boards have much less: the RED-V RedBoard has 32 MiB of external flash, and the RED-V Thing Plus has 4 MiB. The chip also has 16 KiB of RAM, called a *data tightly integrated memory* (DTIM), at address 0x80000000. This RAM has a 2-cycle load latency and is used to hold variables. Various peripherals are memory-mapped between addresses 0x02000000 and 0x1FFFFFFF and will be described in detail in later sections. These peripherals include general-purpose I/O, three pulse-width modulation (PWM) blocks for generating output waveforms, and many serial ports to connect to external devices, including three serial peripheral interfaces (SPIs), two universal asynchronous receiver/transmitters (UARTs), and one inter-integrated circuit (I^2C) interface.

Figure e9.3 shows a simplified schematic of the RED-V RedBoard. The board receives 5V power from a USB power supply and regulators produce 3.3V and 1.8V for I/O, powering the low-power always-on core and miscellaneous functions.

The RISC-V microcontrollers from SiFive continue to advance. By the time you read this, a newer model might be available with a more advanced processor and a different set of embedded I/O. Nevertheless, the same principles discussed here apply to that microcontroller as well as other microcontrollers. You can expect to find the same types of I/O and peripherals. You will need to consult the datasheet to look up the mapping between the peripheral, the pin on the chip, and the pin on the board, as well as the memory-mapped I/O addresses (registers) associated with each peripheral. But, as described here, you will still write to configuration registers to initialize the peripheral and read and write data registers to communicate with the peripheral.

9.3.3 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. At a minimum, GPIO pins require memory-mapped I/O registers

Table e9.1 **FE310 Memory Map**

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Memory attributes: R: Read, W: Write, X: Execute, C: Cacheable, A: Atomics.
Reprinted with permission from Table 4 of the *FE310-G0002 Manual*, © 2019 SiFive, Inc.

to read input pin values, write output pin values, and set the direction of the pin. In many embedded systems, the GPIO pins can be shared with one or more special-purpose peripherals, so additional configuration registers are necessary to determine whether the pin is general- or special-purpose. Furthermore, the processor may generate interrupts when an event such as a rising or falling edge occurs on an input pin, and configuration registers may be used to specify the conditions for

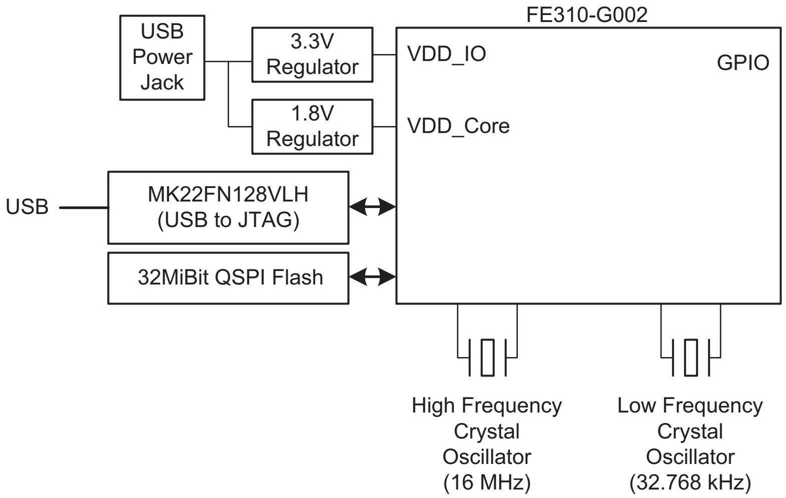

FE310-G002
USB Power Jack
3.3V Regulator
1.8V Regulator
VDD_IO
VDD_Core
GPIO
USB
MK22FN128VLH (USB to JTAG)
32MiBit QSPI Flash
High Frequency Crystal Oscillator (16 MHz)
Low Frequency Crystal Oscillator (32.768 kHz)

Figure e9.3 **RED-V board schematic**

an interrupt. Recall that the FE310 has 19 GPIO pins. This section will start with a basic example of controlling these pins and then will look at some of the special purposes for these pins.

Figure e9.4 shows three light-emitting diodes (LEDs) and three switches connected to six GPIO pins. The LEDs are wired to glow when driven to 1 and to turn off when driven to 0. The current-limiting (typically around 300 Ω) resistors are placed in series with the LEDs to set the brightness and to avoid overloading the current capability of the GPIO. The switches are wired to produce a 1 when closed and a 0 when open. As shown, the 1 kΩ pull-down resistors pull the pins down to 0 when the switches are open. Figure e9.4 indicates the (Arduino) pin numbers that are labeled on the board as well as the GPIO pin numbers.

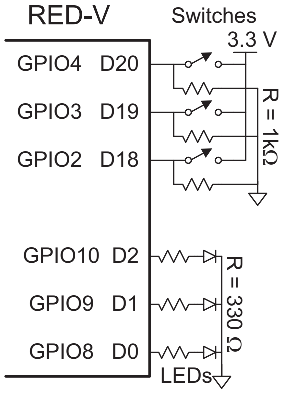


Figure e9.4 **GPIO example**

Table e9.2 lists the GPIO registers and their address offsets relative to the GPIO base address, 0x10012000, as shown in Table 51 of the *FE310-G002 Manual*. Let's first focus on the top four registers (i.e., memory-mapped I/O addresses). Each GPIO pin is mapped to one bit of the registers. Reading from the `input_val` (input value) register reads the values of the GPIO pins, and writing to the `output_val` (output value) register writes to the GPIO pins. Before reading or writing to the pins, the input and output enable registers (`input_en` and `output_en`) must be set to configure the pins as inputs or outputs and the hardware-driven function enable register (`iof_en`) must be cleared to configure the pins as GPIO controlled.

GPIO Memory-Mapped I/O

We illustrate how to use the GPIO pins by writing a program that reads the state of a switch and controls an LED using the GPIOs. The

Table e9.2 **GPIO register offsets**

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable*
0x08	output_en	Pin output enable*
0x0C	output_val	Output value
0x10	pue	Internal pull-up enable*
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x38	iof_en	HW-driven functions enable
0x3C	iof_sel	HW-driven functions selection
0x40	out_xor	Output XOR (invert)

Registers with * are asynchronously reset to 0 at start-up so that GPIO pins are inactive.
Reprinted with permission from Table 52 of the SiFive *FE310-G0002 Manual,* © 2019 SiFive, Inc.

five most important registers for interacting with the GPIO pins are, as described above, input_val, input_en, output_en, output_val, and iof_en at offsets of 0x0, 0x4, 0x8, 0xC, and 0x38 from the base address. Each register is 32 bits wide and could control up to 32 GPIOs, but only 19 GPIOs are physically present on this chip.

To read GPIO *n*, a program sets bit *n* of the input_en (input enable) register and then reads the input_val (input value) register and looks at bit *n*. Similarly, to drive GPIO *n*, a program sets bit *n* of the output_en (output enable) register and then writes the desired value to bit *n* of the output_val (output value) register. In both cases, bit *n* of the `iof_en` register must be cleared to ensure that the pin is driven by the GPIO controller instead of other hardware on the chip.

Code Example e9.1 illustrates a simple program that reads the value of the switch connected to GPIO19 and accordingly turns ON or OFF the on-board LED connected to GPIO5. The hardware setup is shown in Figure e9.5. To access the memory-mapped I/O, it first declares pointers to the five registers at the addresses mentioned above. Each pointer is of type `uint32_t*` because the registers contain unsigned 32-bit values. The program writes a 1 to bit 19 of the `input_en` register and a 1 to bit 5 of the `output_en` register to configure GPIO pin 19 as an input and GPIO pin 5 as an output. Notice how we use the shift operation `(1 << 19)` to set a 1 in bit 19 and OR it with the existing contents of the enable register to turn on that bit without affecting other bits that might already be turned on. Then, we write a 0 to bits 5 and 19 in the `iof_en` register to ensure that the pins are driven by the GPIO controller. To write a 0 to a bit, we AND `iof_en` with 1's in every position except that bit so that the desired bit is forced low and the other bits are not affected. Next, the program repeatedly reads the input pin and writes the output pin. To read the input pin, the program reads the `input_val` register, right-shifts the value by 19 (to move pin 19's value into bit 0), and performs a bitwise `AND` with `0x1` to retain only bit 0, leaving a single 0 or 1 corresponding to the value originally in bit 19. To write a high value to a bit of the `output_val` register, we use the `OR` operation, as we did to turn on a bit in the enable registers. To write a 0 to a bit in the `output_val` register, we use the same approach as described above for clearing bits in the `iof_en` register.

In the context of bit manipulation, "setting" means to write 1 to a bit and "clearing" means to write 0 to a bit.

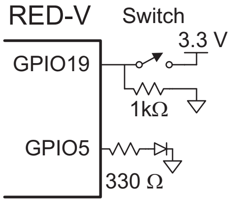


Figure e9.5 **LED output on GPIO pin 5 and switch input from GPIO pin 19**

Code Example e9.1 **SETTING GPIO OUTPUT BASED ON SWITCH INPUT**

```
#include <stdint.h>
int main(void) {
    volatile uint32_t *input_val  = (uint32_t*)0x10012000;
    volatile uint32_t *input_en   = (uint32_t*)0x10012004;
    volatile uint32_t *output_en  = (uint32_t*)0x10012008;
    volatile uint32_t *output_val = (uint32_t*)0x1001200C;
    volatile uint32_t *iof_en     = (uint32_t*)0x10012038;
    int val;

    *iof_en    &= ~(1 << 19);                // Pin 19 is a GPIO
    *input_en  |=  (1 << 19);                // Pin 19 is an input
    *iof_en    &= ~(1 << 5);                 // Pin 5 is a GPIO
    *output_en |=  (1 << 5);                 // Pin 5 is an output
    while (1)  {
        val = (*input_val >> 19) & 1;        // Read value on pin 19
        if (val) *output_val |= (1 << 5);    // Turn ON pin 5
        else     *output_val &= ~(1 << 5);   // TURN OFF pin 5
    }
}
```

Other GPIO Registers

Table e9.2 listed several other GPIO control registers of interest, particularly the pin drive strength (`ds`), internal pull-up enable (`pue`), and I/O function (`iof_sel` and `iof_en`) registers.

The `ds` register controls each pin's maximum output current. The default value (0) configures I_{OL}/I_{OH} as 15 to 16 mA, while setting a pin's `ds` to 1 increases that pin's output current modestly to 21 mA, which might be helpful to drive a brighter LED.

The `pue` register configures an internal pull-up resistor. Figure e9.4 showed an example of an external *pull-down* resistor. If the power and ground connections on the switch were reversed, the resistor would then be a *pull-up* resistor that drives the pin to 1 when the switch is not connected. In that case, when the switch was pressed, the pin would drop to 0. To save money and circuit board space, many microcontrollers contain internal pull-up resistors that can optionally be enabled in software. Writing a 1 to a bit of the `pue` register activates the internal pull-up resistor for the corresponding GPIO pin. According to Table 4.2 of the FE310-G002 datasheet, the pull-up current is 85 μA when the pin is at 0 V. Hence, the effective pull-up resistance is $3.3\,V/85\,\mu A = 39\,k\Omega$ ($V/I = R$).

As shown in Table e9.3, most GPIO pins can also perform a special function, such as acting as a serial port or a pulse-width modulation (PWM) output. We discuss these functions in detail later in this chapter. The `iof_sel` and `iof_en` registers together determine whether each pin is acting as a GPIO or as a special function. When `iof_en` is 0 (the default), the pin acts as a GPIO. When it is 1, it takes on the special function. The special function is chosen from Table e9.3 based on the `iof_sel` bit for that pin. For example, to use GPIO11 to generate a pulse-width modulated waveform, set bit 11 of `iof_sel` and `iof_en` to 1. Then, use the PWM registers to control the output. `iof_en` is mapped to address 0x10012038 and `iof_sel` to 0x1001203C. Table e9.3 lists 32 GPIOs; however, remember that the RED-V boards only include 19 GPIOs: GPIOs 0 to 5, 9 to 13, and 16 to 23.

9.3.4 Device Drivers

As we saw in Code Example e9.1, programmers can manipulate I/O devices directly by reading or writing the memory-mapped I/O registers. However, it is better programming practice to call functions that access the memory-mapped I/O. These functions are called *device drivers*. Some of the benefits of using device drivers include:

- The code is easier to read when it involves a clearly named function call rather than a write to bit fields at an obscure memory address.
- Somebody who is familiar with the deep workings of the I/O devices can write the device driver and casual users can call it without having to understand the details.

Table e9.3 GPIO pins special functions map

GPIO Number	IOF0	IOF1
0		PWM0_PWM0
1		PWM0_PWM1
2	SPI1_CS0	PWM0_PWM2
3	SPI1_DQ0	PWM0_PWM3
4	SPI1_DQ1	
5	SPI1_SCK	
6	SPI1_DQ2	
7	SPI1_DQ3	
8	SPI1_CS1	
9	SPI1_CS2	
10	SPI1_CS3	PWM2_PWM0
11		PWM2_PWM1
12	I2C0_SDA	PWM2_PWM2
13	I2C0_SCL	PWM2_PWM3
14		
15		
16	UART0_RX	
17	UART0_TX	
18	UART1_TX	
19		PWM1_PWM1
20		PWM1_PWM0
21		PWM1_PWM2
22		PWM1_PWM3
23	UART1_RX	
24		
25		
26	SPI2_CS0	
27	SPI2_DQ0	
28	SPI2_DQ1	
29	SPI2_SCK	
30	SPI2_DQ2	
31	SPI2_DQ3	

Reprinted with permission from Table 53 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

- The code is easier to port to another processor with different memory mapping or I/O devices because only the device driver must change.
- If the device driver is part of the operating system (OS), the OS can control access to physical devices shared among multiple programs running on the system and can manage security (e.g., so a malicious program can't read the keyboard while you are typing your password into a web browser).

This chapter will develop a simple device driver called EasyREDVIO to access FE310 peripherals so that you can understand what is happening under the hood in a device driver. To access all features of the FE310, users may prefer the Freedom Metal environment, which provides convenient software interfaces for controlling SiFive Core IP features and peripheral devices. Freedom Metal is powerful since it is written in such a way that its API will work on any device that has a Freedom Metal board support package (BSP). A BSP is a software package containing drivers and other commonly used routines. SiFive also provides the Freedom E software developer kit (SDK) and Freedom Studio, which allow users to develop software for any SiFive core.

EasyREDVIO and the code examples in this chapter can be downloaded from the textbook website (see the Preface). More information about Freedom Metal and documentation can be found at https://github.com/sifive/freedom-metal.

Example e9.2 DEVICE DRIVERS IN C

Accessing and modifying the values for memory-mapped I/O is accomplished by reading or writing to memory addresses. In assembly, this is done using `lw` and `sw` instructions. As illustrated in Code Example e9.2, C can do the same thing with pointers, but it is tedious and error-prone to declare pointers for every memory-mapped I/O register. A more natural way to describe and control memory-mapped I/O in C is using structures.

As discussed in Section C.8.5 in the appendix, structures in C are a way to group a collection of different data types into a single unit. Using structures in the context of memory-mapped registers allows communication with the I/O device using the name of a given register or field as opposed to a memory address. A C program can declare a structure for a memory-mapped peripheral, listing the registers in the order they appear in the memory map. It can then declare a pointer to such a structure and access the peripheral via the structure pointer.

Start the EasyREDVIO library by writing `pinMode`, `digitalRead`, and `digitalWrite` functions to configure a pin's direction and read or write it.

- The `pinMode` function takes two inputs: the pin number and the mode. For example, `pinMode(5, INPUT)` sets GPIO pin 5 as an input, and `pinMode(17, OUTPUT)` sets GPIO pin 17 as an output.
- `digitalRead` takes one input, the pin number, and returns the value of that pin. For example, `digitalRead(19)` reads the value of GPIO19.
- `digitalWrite` takes two inputs: the pin number and the value. For example, `digitalWrite(3, 1)` writes 1 to GPIO pin 3, and `digitalWrite(5, 0)` writes 0 to GPIO pin 5.

After writing these functions, write a C program that uses these functions to read the three switches and turn on the corresponding LEDs, using the hardware in Figure e9.4.

Solution The EasyREDVIO code is given below. The functions must choose which registers and bits within those registers to access. For example, to configure a pin as an input, pinMode must set that pin's bit in input_en and clear that pin's bit in output_en. digitalWrite handles writing either 1 or 0 by writing to output_val. digitalRead reads the desired bit of input_val.

The GPIO structure (struct) specifies the 32-bit registers by name. Two define statements then specify the base address of the GPIO (GPIO0_BASE) and instantiate a pointer of type GPIO located at that base address. Each of the 32-bit variables in the structure are then located in memory in ascending order from that base address.

Code Example e9.2 GPIO FOR SWITCHES AND LEDS

```
// EasyREDVIO.h
// Joshua Brake, David Harris, and Sarah Harris, 7 October 2020

#include <stdint.h>

#define INPUT  0
#define OUTPUT 1

// Define statements to map Arduino pin names to FE310 GPIO pin number according to Figure e9.2
#define D0  16
#define D1  17
#define D2  18
#define D3  19
#define D4  20
#define D5  21
#define D6  22
#define D7  23
#define D8  0
#define D9  1
#define D10 2
#define D11 3
#define D12 4
#define D13 5
#define D15 9
#define D16 10
#define D17 11
#define D18 12
#define D19 13

// Declare a GPIO structure defining the GPIO registers in the order they appear in Table e9.2
typedef struct {
    volatile uint32_t  input_val;     // (GPIO offset 0x00) Pin value
    volatile uint32_t  input_en;      // (GPIO offset 0x04) Pin input enable*
    volatile uint32_t  output_en;     // (GPIO offset 0x08) Pin output enable*
    volatile uint32_t  output_val;    // (GPIO offset 0x0C) Output value
    volatile uint32_t  pue;           // (GPIO offset 0x10) Internal pull-up enable*
    volatile uint32_t  ds;            // (GPIO offset 0x14) Pin drive strength
    volatile uint32_t  rise_ie;       // (GPIO offset 0x18) Rise interrupt enable
    volatile uint32_t  rise_ip;       // (GPIO offset 0x1C) Rise interrupt pending
    volatile uint32_t  fall_ie;       // (GPIO offset 0x20) Fall interrupt enable
    volatile uint32_t  fall_ip;       // (GPIO offset 0x24) Fall interrupt pending
    volatile uint32_t  high_ie;       // (GPIO offset 0x28) High interrupt enable
    volatile uint32_t  high_ip;       // (GPIO offset 0x2C) High interrupt pending
    volatile uint32_t  low_ie;        // (GPIO offset 0x30) Low interrupt enable
    volatile uint32_t  low_ip;        // (GPIO offset 0x34) Low interrupt pending
    volatile uint32_t  iof_en;        // (GPIO offset 0x38) HW-Driven functions enable
    volatile uint32_t  iof_sel;       // (GPIO offset 0x3C) HW-Driven functions selection
    volatile uint32_t  out_xor;       // (GPIO offset 0x40) Output XOR (invert)
    // Registers marked with * are asynchronously reset to 0 at startup
} GPIO;
```

```
// Define the base address of the GPIO registers (see Table e9.1) and a pointer to this
// structure
// The 0x…U notation in 0x10012000U indicates an unsigned hexadecimal number
#define GPIO0_BASE  (0x10012000U)
#define GPIO0 ((GPIO*) GPIO0_BASE)

// To access the members of the structure, the member-access operator -> is used.

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
            GPIO0->input_en   |=  (1 << gpio_pin);  // Sets a pin as an input
            GPIO0->output_en  &= ~(1 << gpio_pin);  // Clear output_en bit
            GPIO0->iof_en     &= ~(1 << gpio_pin);  // Disable IOF
            break;
        case OUTPUT:
            GPIO0->output_en  |=  (1 << gpio_pin);  // Set pin as an output
            GPIO0->input_en   &= ~(1 << gpio_pin);  // Clear input_en bit
            GPIO0->iof_en     &= ~(1 << gpio_pin);  // Disable IOF
            break;
    }
}

void digitalWrite(int gpio_pin, int val) {
    if (val) GPIO0->output_val |=  (1 << gpio_pin);
    else     GPIO0->output_val &= ~(1 << gpio_pin);
}

int digitalRead(int gpio_pin) {
    return (GPIO0->input_val >> gpio_pin) & 0x1;
}

// The program below reads switches and writes LEDs. It sets pins 2 to 4 as inputs (for the
// switches) and pins 7 to 9 as outputs (for the LEDs). It then continuously reads the
// switches and writes their values to the corresponding LEDs.

#include "EasyREDVIO.h"
int main(void) {
  // Set GPIO 4:2 as inputs
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);
  // Set GPIO 10:8 as outputs
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  while (1) { // Read each switch and write corresponding LED
    digitalWrite(8,  digitalRead(2));
    digitalWrite(9,  digitalRead(3));
    digitalWrite(10, digitalRead(4));
  }
}
```

9.3.5 Serial I/O

A microcontroller can send multiple bits to a peripheral device by using multiple wires or by sending multiple bits in series over a single wire. The former is called *parallel I/O* and the latter is called *serial I/O*. Serial I/O is popular, especially when pins are limited, because it uses few wires and is fast enough for many applications. Indeed, it is so popular that many standards for serial I/O have been established and microcontrollers offer dedicated hardware to easily send data via these standards. This section describes the SPI and UART standard serial interfaces.

Other common serial standards include inter-integrated circuit (I^2C), universal serial bus (USB), and Ethernet. I^2C (pronounced "I squared C") is a 2-wire interface with a clock and a bidirectional data pin; it is used in a fashion similar to SPI. USB and Ethernet are more complex, high-performance standards. The FE310 supports SPI, UART, and I^2C via on-board specialized peripherals.

The terms *master/slave* used to be common (instead of *controller/peripheral*), but they are now outdated. *Serial data out (SDO)* or *controller-out peripheral-in (COPI)* is now used in place of *master-out slave-in (MOSI)*. *Serial data in (SDI) or controller-in peripheral-out (CIPO)* is now used in place of *master-in slave-out (MISO)*.

Serial Peripheral Interface (SPI)

SPI (pronounced "S-P-I") is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: serial clock (SCK), serial data out (SDO), and serial data in (SDI). SPI connects a *controller* device to a *peripheral* device, as shown in Figure e9.6(a). The controller produces the clock. It initiates communication by sending clock pulses on SCK. The controller sends data from its SDO pin to the peripheral's SDI pin one bit per cycle, starting with the most significant bit. The peripheral may simultaneously respond with its SDO pin back to the controller's SDI pin. Figure e9.6(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable to alert the receiver that data is coming.

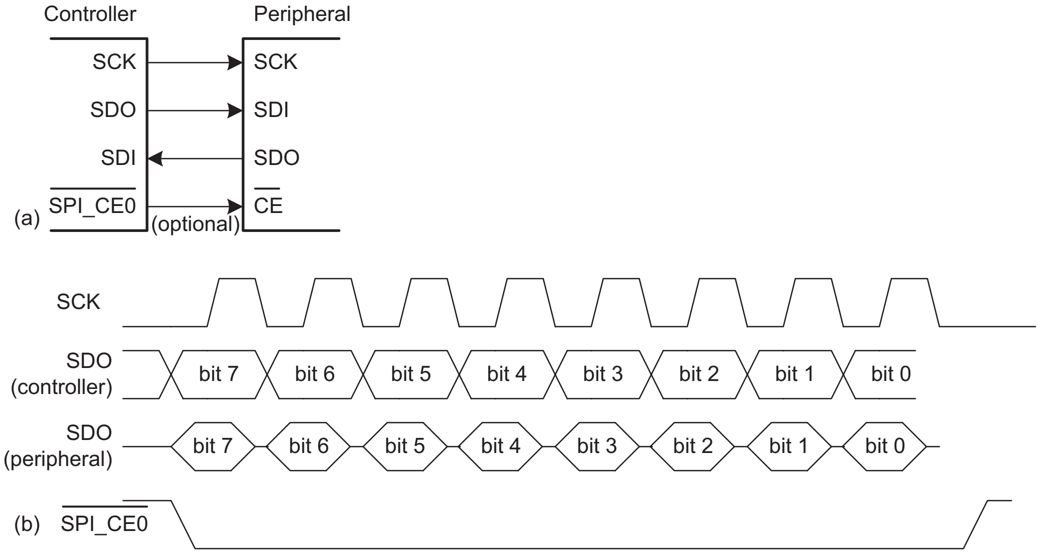


Figure e9.6 **SPI configuration: (a) SPI controller-peripheral connection diagram, (b) Example SPI data signals**

The FE310 has three SPI controller ports, but only two (SPI1 and SPI2) are available to the user. The remaining SPI controller port, SPI0, is used to communicate with external flash memory for program and data storage. This section describes the SPI1 controller port, which is accessible using GPIO pins 5:2. The SPI2 controller port is identical except that it is connected to different GPIO pins and its control registers are located at different memory addresses. To use pins for SPI rather than GPIO, their `iof_sel` bits should be set to 0 to select the SPI1 function and their `iof_en` bits should be set to 1 to give the SPI controller access to the pins. When the FE310 writes to the SPI `txdata` register, the data is transmitted serially to the peripheral. Simultaneously, data received from the peripheral is collected and the FE310 can read it from `rxdata` when the transfer is complete.

The term *port* refers to a pin or a group of associated pins.

SPI always sends data in both directions on each transfer. If the system only needs unidirectional communication, it can ignore the unwanted data. For example, if the controller only needs to send data to the peripheral, the byte received from the peripheral can be ignored. If the controller only needs to receive data from the peripheral, it must still trigger the SPI communication by sending an arbitrary byte that the peripheral will ignore. It can then read the data received from the peripheral. The SPI clock (SCK) only toggles while the controller is transmitting data.

SPI ports on a microcontroller normally offer a variety of configuration options to match the requirements of peripheral devices. When designing an interface to communicate with a particular peripheral device, the controller must be configured properly to ensure that the data being transmitted via the link is properly interpreted.

Two common configuration parameters are clock polarity (CPOL) and clock phase (CPHA). CPOL sets the level of the clock when it is idle and CPHA sets the clock edge when data (SDO and SDI) is sampled (and changed). If CPOL = 1, SCK remains high (1) when data is not being transmitted; if CPOL = 0, SCK remains low (0) when idle. If CPHA = 0, data are sampled on the leading edge (and change on the trailing edge) of SCK; if CPHA = 1, data are sampled on the trailing edge (and change on the leading edge) of SCK. The edge on which data changes is also referred to as the *shifting edge* because the underlying hardware is usually a shift register. Figure e9.7 shows the four possible combinations of CPHA and CPOL. The example from Figure e9.6 shows CPOL = 0 and CPHA = 0.

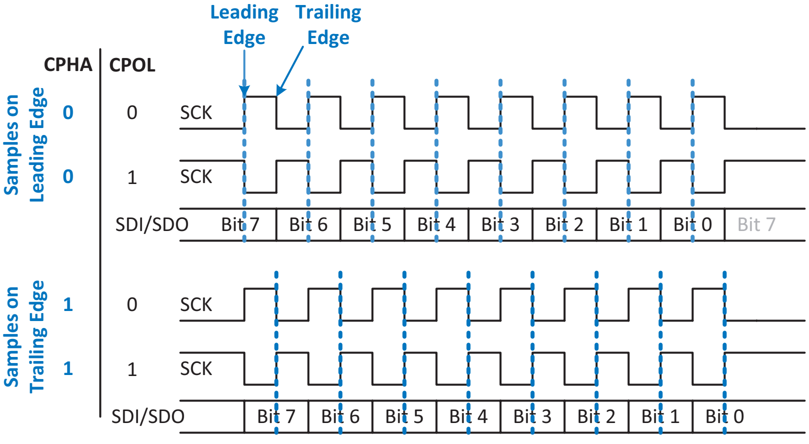


Figure e9.7 Timing diagram and configurations for SPI peripherals

Table e9.4 Memory map of SPI registers

Offset	Name	Description
`0x00`	`sckdiv`	Serial clock divisor
`0x04`	`sckmode`	Serial clock mode
`0x08`	Reserved	
`0x0C`	Reserved	
`0x10`	`csid`	Chip select ID
`0x14`	`csdef`	Chip select default
`0x18`	`csmode`	Chip select mode
`0x1C`	Reserved	
`0x20`	Reserved	
`0x24`	Reserved	
`0x28`	`delay0`	Delay control 0
`0x2C`	`delay1`	Delay control 1
`0x30`	Reserved	
`0x34`	Reserved	
`0x38`	Reserved	
`0x3C`	Reserved	
`0x40`	`fmt`	Frame format
`0x44`	Reserved	
`0x48`	`txdata`	Tx FIFO Data
`0x4C`	`rxdata`	Rx FIFO data
`0x50`	`txmark`	Tx FIFO watermark
`0x54`	`rxmark`	Rx FIFO watermark
`0x58`	Reserved	
`0x5C`	Reserved	
`0x60`	`fctrl`	SPI flash interface control*
`0x64`	`ffmt`	SPI flash instruction format*
`0x68`	Reserved	
`0x6C`	Reserved	
`0x70`	`ie`	SPI interrupt enable
`0x74`	`ip`	SPI interrupt pending

Reprinted with permission from Table 65 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Table e9.4 shows the control registers associated with SPI1, and Table e9.5 shows the fields of the key registers. `sckdiv` (see Table e9.4) configures the SPI clock frequency by specifying a divisor (`div`) for the selected input peripheral clock—on the RED-V board, the peripheral clock's default frequency is 16 MHz. The frequency of the SPI clock is given by $f_{sck} = \frac{f_{in}}{2(div + 1)}$. For example, if `div` = 15, then the serial clock is $f_{sck} = \frac{16\,\text{MHz}}{2(15+1)} = 500\,\text{kHz}$. If the frequency is too high (>~1 MHz on a breadboard or tens of MHz on an unterminated printed circuit board), the SPI connection may become unreliable due to reflections, crosstalk, or other signal integrity issues.

Table e9.5 **SPI register bitfields**

Serial Clock Divisor Register (sckdiv)				
Register Offset	0x0			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[11:0]	div	RW	0x3	Divisor for serial clock. div_width bits wide.
[31:12]	Reserved			

Serial Clock Mode Register (sckmode)				
Register Offset	0x4			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	pha	RW	0x0	Serial clock phase
1	pol	RW	0x0	Serial clock polarity
[31:2]	Reserved			

Transmit Data Register (txdata)				
Register Offset	0x48			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RW	0x0	Transmit data
[30:8]	Reserved			
31	full	RO	X	FIFO full flag

Receive Data Register (rxdata)				
Register Offset	0x4C			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RO	X	Received data
[30:8]	Reserved			
31	empty	RW	X	FIFO empty flag

Reprinted with permission from Tables 66, 67, 80, and 81 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Configuration registers have many unused or "reserved" bits. These bits might be used in a future version of the chip, so they should not be written lest they cause unintended consequences in the future.

sckmode controls the phase and polarity of the clock. sckmode uses only the two least significant bits. Bit 0 is CPHA and bit 1 is CPOL.

txdata is written to transmit a byte over the SPI channel, and rxdata is read to get the received byte. Only the least significant byte (LSB) written to txdata is transmitted. The SPI instances on the FE310 have 8-entry first-in-first-out (FIFO) buffers on both the transmit and receive data registers. This means that when data is written to the txdata register, it is placed in the FIFO buffer and the hardware within the SPI peripheral takes care of sending it out. The most significant bit (msb) of the txdata register is a flag bit called full, which is 1 when the FIFO is full and cannot receive any more data.

Care must be taken when reading data in from the FE310 SPI rxdata register. The SPI controller is designed such that the data in the register is removed from the receive FIFO when the register is read. To check if the rxdata register has valid data, the register should be read once and then the `empty` bit should be checked to determine if the data is valid. The programmer should take care to avoid reading the register more than once for each byte, as this will result in lost data.

The registers `csid`, `csdef`, and `csmode` are optionally used to control parameters related to the control and configuration of the chip select line. Alternatively, the chip select pin can be configured as a GPIO output pin and controlled in software through `digitalWrite`.

Some SPI registers pack multiple small fields of information into a single 32-bit word. In C, we can declare the number of bits of each field with a colon and number as part of a bitfield structure. Example e9.3 shows how to use bitfields and structures to define these registers.

Example e9.3 SENDING AND RECEIVING BYTES OVER SPI

Design a system to communicate between a FE310 controller and an FPGA peripheral over SPI. Sketch a schematic of the interface. Write the C code for the FE310 to send the character “A” and receive a character back. Write HDL code for an SPI peripheral on the FPGA. How could the peripheral be simplified if it only needs to receive data?

Solution Figure e9.8 shows the connection between the FE310 controller and the FPGA peripheral using SPI1. The pin numbers are obtained from the component datasheets (e.g., Table e9.3 for the FE310). Notice that both the pin numbers and signal names are shown on the diagram to indicate both the physical and logical connectivity. When the SPI connection is enabled, these pins cannot be used for GPIO.

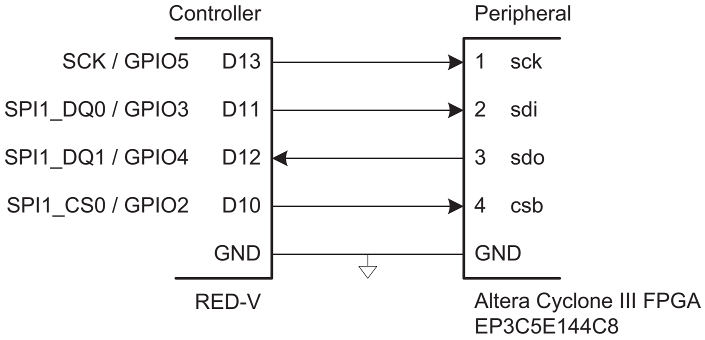


Figure e9.8 RED-V to Altera Cyclone FPGA connection diagram

The following code from EasyREDVIO.h is used to initialize the SPI channel and to send and receive a character. The file first declares the SPI bitfields and memory map. The `pinMode` function is generalized to support I/O functions as well as inputs and outputs. The function `spiSendReceive` completes a full SPI transaction sending and receiving a single byte. It initially checks to make sure that the transmit FIFO is not full and can accept another entry. If yes, it writes the character to the transmit FIFO to be shifted out. After transmitting, the `rxdata` register is read. Here, care must be taken because the empty flag bit of the `rxdata` register is updated whenever the register is read. So, the entire 32-bit `rxdata` register should be read. Then, after checking that the empty flag is not set (i.e., the data is valid), the received byte is returned.

Code Example e9.3 SPI FUNCTIONS

```
////////////////////////////////////////////////////////////////////////////////
// SPI Registers
////////////////////////////////////////////////////////////////////////////////
typedef struct {
    volatile uint32_t     div            :     12;   // Clock divisor
    volatile uint32_t                    :     20;
} sckdiv_bits;

typedef struct {
    volatile uint32_t     pha            :     1;    // Serial clock phase
    volatile uint32_t     pol            :     1;    // Serial clock polarity
    volatile uint32_t                    :     30;
} sckmode_bits;

...

typedef struct {
    volatile uint32_t     data           :     8;    // Transmit data
    volatile uint32_t                    :     23;
    volatile uint32_t     full           :     1;    // FIFO full flag
} txdata_bits;

typedef struct {
    volatile uint32_t     data           :     8;    // Received data
    volatile uint32_t                    :     23;
    volatile uint32_t     empty          :     1;    // FIFO empty flag
} rxdata_bits;

// Pin modes
#define INPUT  0
#define OUTPUT 1
#define GPIO_IOF0 2
#define GPIO_IOF1 3

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
            GPIO0->input_en    |=  (1 << gpio_pin);    // Set a pin as an input
            GPIO0->iof_en      &= ~(1 << gpio_pin);    // Disable IOF
            break;
        case OUTPUT:
            GPIO0->output_en   |=  (1 << gpio_pin);    // Set pin as an output
            GPIO0->iof_en      &= ~(1 << gpio_pin);    // Disable IOF
            break;
        case GPIO_IOF0:
            GPIO0->iof_en      |=  (1 << gpio_pin);    // Enable IOF
            GPIO0->iof_sel     &= ~(1 << gpio_pin);    // IO Function 0
            break;
        case GPIO_IOF1:
            GPIO0->iof_en      |= (1 << gpio_pin);     // Enable IOF
            GPIO0->iof_sel     |= (1 << gpio_pin);     // IO Function 1
            break;
    }
}
```

```
void spiInit(uint32_t clkdivide, uint32_t cpol, uint32_t cpha) {

    // Initially assigning SPI pins (GPIO 2-5) to HW I/O function 0
    pinMode(3, GPIO_IOF0); // SDO
    pinMode(4, GPIO_IOF0); // SDI
    pinMode(5, GPIO_IOF0); // SCK

    digitalWrite(2, 1);    // make sure CS0 doesn't pulse low
    pinMode(2, OUTPUT);    // CS0 is manually controlled

    SPI1->sckdiv.div = clkdivide; // Set the clock divisor

    SPI1->sckmode.pol = cpol;     // Set the polarity
    SPI1->sckmode.pha = cpha;     // Set the phase
}

/* Transmits a character (1 byte) over SPI and returns the received character.
 *    send: the character to send over SPI
 *    return value: the character received over SPI */
uint8_t spiSendReceive(uint8_t send) {
    while(SPI1->txdata.full);  // Wait until transmit FIFO is ready for new data
    SPI1->txdata.data = send;  // Transmit the character over SPI

    rxdata_bits rxdata;
    while (1) {
        rxdata = SPI1->rxdata;  // Read the rxdata register EXACTLY once
        if (!rxdata.empty) {    // If the empty bit was not set, return the data
            return (uint8_t)rxdata.data;
        }
    }
}
```

The C code in Code Example e9.4 initializes the SPI and then sends and receives a character. Using the formula $f_{clk} = \frac{f_{in}}{2(div+1)}$, where f_{in} is the 16 MHz `coreclk`, it sets the SPI clock to 500 kHz.

Code Example e9.4 SPI FUNCTIONS

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t volatile received;

  // Initialize the SPI
  // Clock divisor of div = 15, CPOL = 0, CPHA = 0
  spiInit(15, 0, 0);

  digitalWrite(2, 0);             // enable the peripheral (chip select = 0), if necessary
  received = spiSendReceive('A'); // Send letter A and receive byte
  digitalWrite(2, 1);             // turn off chip enable
}
```

If the peripheral needs to receive data only from the controller, it is a simple shift register, as shown in HDL Example e9.1. On each rising sck edge, a new sdi bit is shifted into the shift register, starting with the data's most significant bit. After eight clock cycles, the entire byte has been read into the shift register.

HDL Example e9.1 HDL FOR SPI PERIPHERAL (RECEIVER ONLY)

```
module spi_peripheral_receive_only(input  logic       sck, // From controller
                                   input  logic       sdi, // From controller
                                   output logic [7:0] q);  // Data received
  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

HDL Example e9.2 gives the SystemVerilog code for an SPI peripheral that can both send and receive data (i.e., an SPI transceiver), and Figure e9.9 shows its block diagram and timing with CPHA = CPOL = 0. The main component is still a shift register, shown on the right of Figure e9.9. The shift register parallel loads the byte to send (d[7:0]) into the shift register and then shifts out this data on sdo while it shifts in data transmitted from the controller (t[7:0]) on sdi. A counter, cnt, keeps track of how many bits have been sent/received. When sck is idle, cnt = 0 and the most significant bit of d (d[7]) sits on the sdo wire. One subtlety is that sdo can only change on the falling clock edge, so the sdo output (which is the most significant bit of the shift register, q[7], is delayed by half a clock cycle by the negative-edge triggered qdelayed register on the bottom left of Figure e9.9.

HDL Example e9.2 HDL FOR SPI PERIPHERAL

```
module spi_peripheral(input  logic       sck,   // From controller
                      input  logic       sdi,   // From controller
                      output logic       sdo,   // To controller
                      input  logic       reset, // System reset
                      input  logic [7:0] d,     // Data to send
                      output logic [7:0] q);    // Data received
  logic [2:0] cnt;
  logic  qdelayed;

  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else       cnt = cnt + 3'b1;
```

```
  // Loadable shift register
  // Loads d at the start, shifts sdi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], sdi} : {q[6:0], sdi};

  // Align sdo to falling edge of sck
  // Load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];

  assign sdo = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

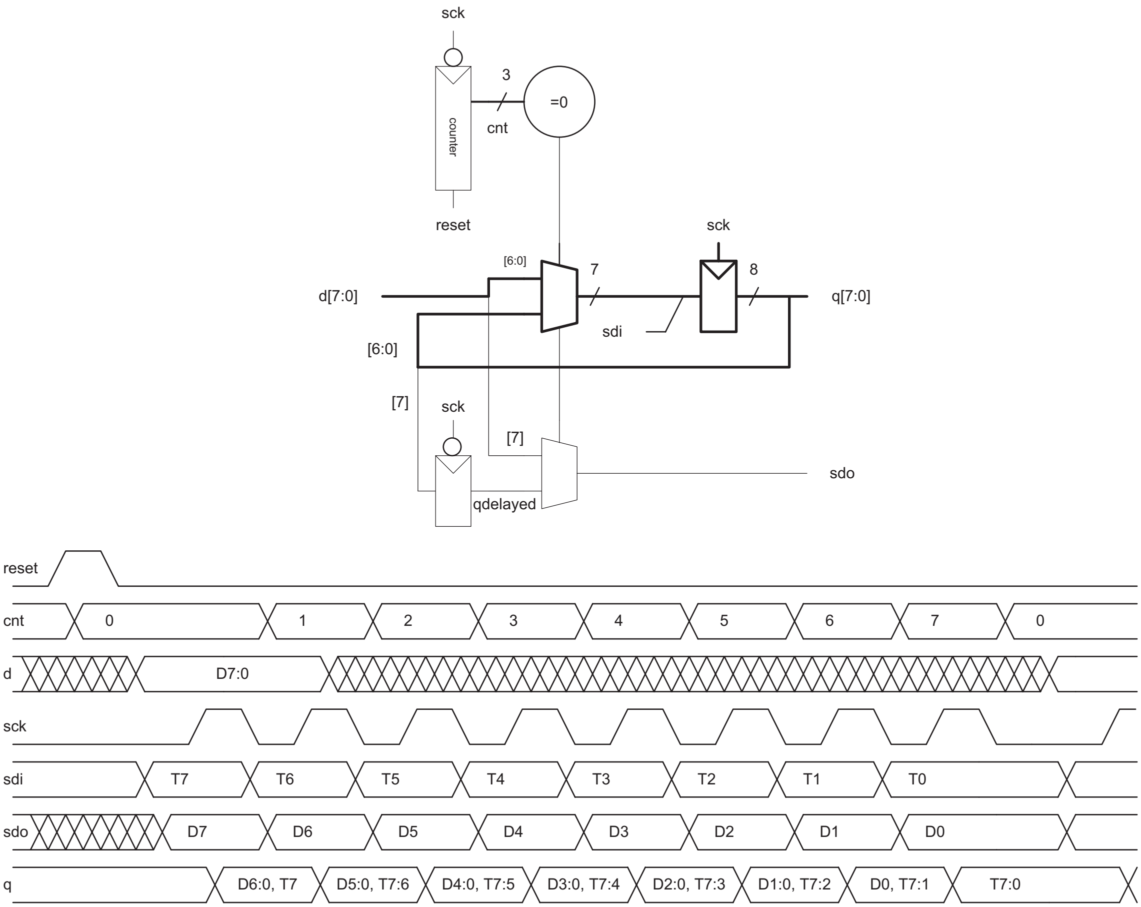


Figure e9.9 Block and timing diagram for SPI peripheral on FPGA

Universal Asynchronous Receiver/Transmitter (UART)

A UART (pronounced "you-art") is a serial I/O peripheral that communicates between two systems without sending a clock. Instead, the systems must agree in advance about what data rate to use and must each locally generate their own clocks. Hence, the transmission is asynchronous because the clocks are not synchronized. Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication. UARTs are used in protocols such as RS-232 and RS-485. For example, old computer serial ports use the RS-232C standard, introduced in 1969 by the Electronics Industries Associations. The standard originally envisioned connecting *data terminal equipment* (DTE) such as a mainframe computer to *data communication equipment* (DCE) such as a modem. Although a UART is relatively slow compared with SPI and prone to misconfiguration issues, the standards have been around for so long that they remain important today.

Figure e9.10(a) shows an asynchronous serial link. The DTE sends data to the DCE over the TX line and receives data back over the RX line. Figure e9.10(b) shows one of these lines sending a character at a data rate of 9600 baud. The lines idle at a logic "1" when not in use. Each character is sent as a start bit (0), 7 or 8 data bits, an optional parity bit, and one or more stop bits (1's). Most typically, start and stop bits and 8 bits of data are sent. The UART detects the falling transition from idle to start to lock on to the transmission at the appropriate time. Although seven data bits is sufficient to send an ASCII character, eight bits are normally used because they can convey an arbitrary byte of data.

The optional parity bit allows the system to detect if a bit was corrupted during transmission. It can be configured as *even* or *odd*; *even parity* means that the parity bit is chosen such that the total collection of data and parity has an even number of 1's. In other words, the parity bit is the XOR of the data bits. The receiver can then check if an even number of 1's was received and signal an error if not. *Odd parity* is the reverse.

A common choice is 1 start bit, 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information. Hence, signaling rates are referred to in units of baud rather than bits/sec. For example, 9600 baud indicates 9600 symbols/sec, or 960 characters/second. Both the transmitter and receiver must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled. This is a hassle, especially for nontechnical users, which is one of the reasons that USB has replaced UARTs in personal computer systems.

Typical baud rates include 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200. The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones. In contemporary systems, 9600 and 115200 are two of the most

Baud rate gives the *signaling rate*, measured in symbols per second, whereas bit rate gives the *data rate*, measured in bits per second. In a simple system like SPI, where each symbol is a data bit, the baud rate is equal to the bit rate. UARTs and some other signaling conventions require overhead bits in addition to the data. For example, a UART that adds start and stop bits for each 8 bits of data (i.e., 10 symbols per 8 bits of data) and operates at a baud rate of 9600 has a bit rate of (9600 symbols/second) × (8 bits/10 symbols) = 7680 bits/second = 960 characters/second.

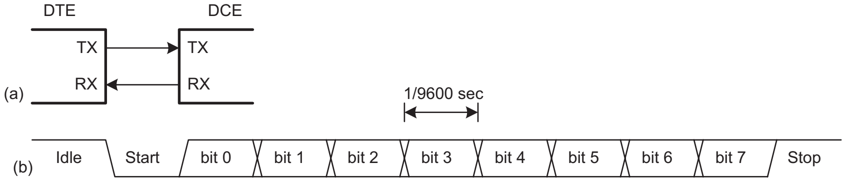


Figure e9.10 Asynchronous serial link

common baud rates; 9600 is encountered where speed does not matter, and 115200 is the fastest standard rate, though still slow compared with other modern serial I/O standards.

In the 1950's through 1970's, early hackers calling themselves *phone phreaks* learned to control the phone company switches by whistling appropriate tones. A 2600 Hz tone produced by a toy whistle from a Cap'n Crunch cereal box (Figure e9.11). could be exploited to place free long-distance and international calls.

Figure e9.11 Cap'n Crunch Bosun Whistle (Photograph by Evrim Sen, reprinted with permission.)

The RS-232 standard defines several additional signals. The request to send (RTS) and clear to send (CTS) signals can be used for *hardware handshaking*. They can be operated in either of two modes: flow control or simplex. In *flow control* mode, the DTE clears RTS to 0 when it is ready to accept data from the DCE. Likewise, the DCE clears CTS to 0 when it is ready to receive data from the DTE. Some datasheets use an overbar to indicate that they are active-low. In the older *simplex* mode, the DTE clears RTS to 0 when it is ready to transmit. The DCE replies by clearing CTS when it is ready to receive the transmission.

Some systems, especially those connected over a telephone line, also used data terminal ready (DTR), data carrier detect (DCD), data set ready (DSR), and ring indicator (RI) signals to indicate when equipment is connected to the line. These signals still show up in some connectors.

Handshaking refers to the negotiation between two systems. Typically, one system signals that it is ready to send or receive data and the other system acknowledges that request.

The original RS-232 standard recommended a massive 25-pin DB-25 connector, but PCs streamlined it to a male 9-pin DE-9 connector with the pinout shown in Figure e9.12(a). The cable wires normally connect straight across, as shown in Figure e9.12(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure e9.12(c) may be needed to swap RX and TX and complete the handshaking. As a final insult, some connectors are male and some are female. In summary, it can take a large box of cables and a certain amount of guesswork to connect two systems over RS-232, again explaining the shift to USB. Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS.

RS-232 represents a 0 electrically with 3 to 15 V and a 1 with −3 to −15 V; this is called *bipolar* signaling. A transceiver converts the digital logic levels of the UART to the positive and negative levels expected by RS-232 and also provides electrostatic discharge protection to protect the serial port from getting zapped when the user plugs in a cable. The MAX3232E is a popular transceiver compatible with both 3.3 and 5 V digital logic. It contains a charge pump that, in conjunction with external capacitors, generates ±5 V outputs from a single low-voltage power supply. Some serial peripherals intended for embedded systems omit the transceiver and just use 0 V for a 0 and 3.3 or 5 V for a 1; check the datasheet!

The FE310 has two UARTs, named UART0 and UART1. UART0 can be configured to operate on pins 16 and 17; UART1 operates on pins 18 and 23. To use these pins as a UART instead of as GPIOs, their corresponding `iof_sel` bits should be set to 0 (to select `IOF0`) and `iof_en` bits set to 1 to enable peripheral control. As with SPI, the FE310 must first configure the port. Unlike SPI, reading and writing can occur independently because either system may transmit without receiving and vice versa. UART0's registers are shown in Table e9.6.

To configure the UART, first set the baud rate. The UART uses the on-board TileLink bus clock, tlclk, as its clock source. For the FE310-G002, this bus clock is configured by default to be the same as the processor clock, coreclk, at 16 MHz. This clock signal must be divided down to produce the desired baud rate. The final baud rate is given by Equation 9.1:

$$f_{baud} = \frac{f_{in}}{div + 1} \qquad \text{(e9.1)}$$

The FE310 UART peripheral supports only 8-N-1 and 8-N-2 protocol configurations. Both protocols support 8 data bits and no parity bit, and the packets can be configured to have either one stop bit (in 8-N-1) or two stop bits (in 8-N-2). The stop bit configuration is set in the `txctrl` register using the `nstop` field. By default, `nstop` = `0`, which sets the peripheral to use one stop bit.

Data is transmitted and received using the `txdata` and `rxdata` registers, respectively. Both the transmit and receive registers are buffered by 8-entry, FIFO buffers. To transmit data, check that the full bit of the `txdata` register is 0, which indicates that there is room in the FIFO buffer for new data to be written. Then, write a byte to the data field in `txdata`. To read data, read the `rxdata` register and check that the empty bit is 0 to confirm that the byte in the data field is valid.

Example e9.4 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for an FE310 to communicate with a PC over a serial port at 115200 baud with 8 data bits, 1 stop bit, and no parity. The PC should be running a console program such as PuTTY[1] to read and write over the serial port. The program should ask the user to type a string. It should then indicate what the user typed.

Solution Figure e9.13(a) shows a basic schematic of the serial link illustrating the issues of level conversion and cabling. Because few PCs still have physical serial ports, we use a Plugable USB to RS-232 DB9 Serial Adapter from plugable.com shown in Figure e9.14 to provide a serial connection to the PC. The adapter connects to a female DE-9 connector soldered to wires that feed a transceiver, which

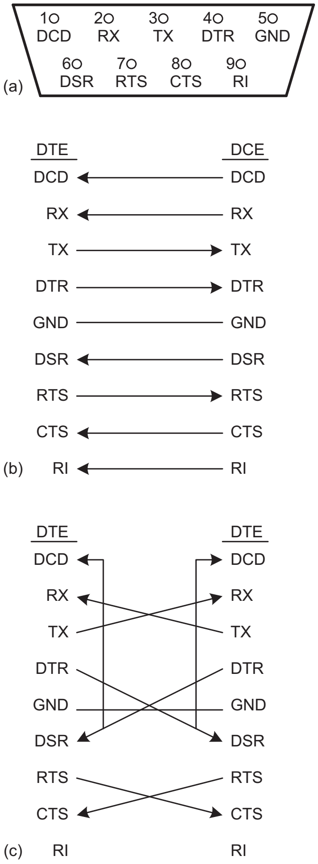


Figure e9.12 DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring

Table e9.6 UART memory mapped registers

	...
0x10013018	div
	...
0x1001300C	rxctrl
0x10013008	txctrl
0x10013004	rxdata
0x10013000	txdata
	...

Adapted and printed with permission from Table 55 of the SiFive *FE310-G002 Manual*, © 2019 SiFive, Inc.

[1] PuTTY is available for free download at www.putty.org.

converts the voltages from the bipolar RS-232 levels to the FE310's 3.3 V level. The FE310 and PC are both DTE, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the FE310 is not used, and the RTS and CTS on the DE9 connector are tied together so that the PC will shake its own hand.

Figure e9.13(b) shows an easier approach with an Adafruit 954 USB to TTL serial cable. The cable is directly compatible with 3.3V levels.

To configure PuTTY to work with the serial link, set *Connection type* to Serial and *Speed* to 115200. Set *Serial line* to the COM port assigned by the operating system to the Serial to USB Adapter. In

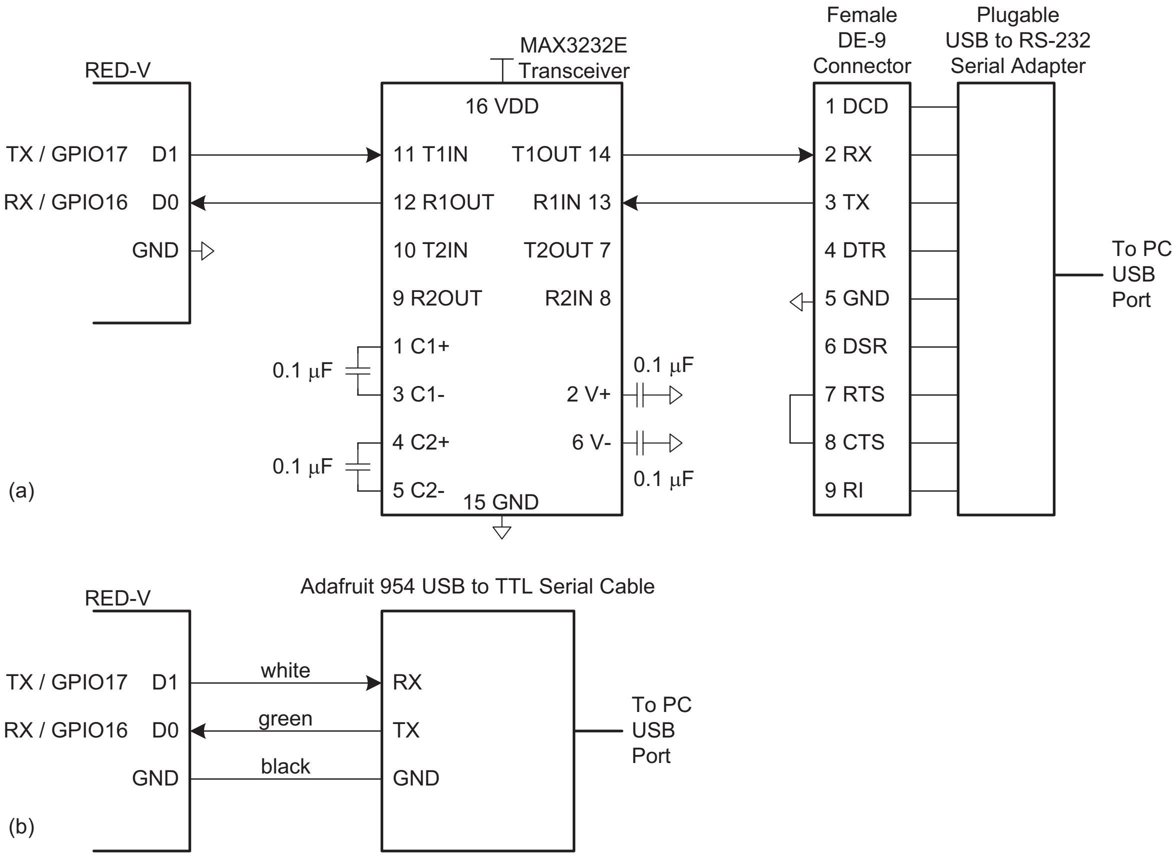


Figure e9.13 Serial communication link schematics: (a) serial communication via RS-232, (b) serial communication with USB to TTL serial cable

Windows, this can be found in the Device Manager; for example, it might be COM3. Under the *Connection* → *Serial* tab, set flow control to NONE or RTS/CTS. Under the Terminal tab, set Local Echo to Force On to have characters appear in the terminal as you type them.

The serial port device driver code in `EasyREDVIO.h` is shown in Code Example e9.5. The Enter key in the terminal program corresponds to a carriage return character represented as `\r` in C with an ASCII code of 0x0D. To advance to the beginning of the next line when printing, send both the `\n` and `\r` (new line and carriage return) characters.[2] The uartInit function configures the UART as described above. `getCharSerial` and `putCharSerial` read and write characters to the terminal, respectively, using the UART (Code Example e9.5).

Figure e9.14 **Plugable USB to RS-232 DB9 serial adapter**

Code Example e9.5 READING AND WRITING CHARACTERS (CHARS) TO A TERMINAL USING A UART

```
void uartInit(uint32_t baud) {
    uint32_t div = 16000000/baud-1;      // 16 MHz tileclock
    pinMode(16, GPIO_IOF0);
    pinMode(17, GPIO_IOF0);

    UART0->div.div = div;                // Set clock divisor
    UART0->txctrl.txen = 1;              // Enable transmitter
    UART0->txctrl.nstop = 1;             // Set one stop bit
    UART0->rxctrl.rxen = 1;              // Enable receiver
}

uint8_t getCharSerial(void) {
    uart_rxdata_bits rxdata;             // Create temporary variable to store register

    while(1) {
        rxdata = UART0->rxdata;          // Read register exactly once
        if(!rxdata.empty) {
            return (uint8_t)rxdata.data; // Check to see if the data is valid
        }
    }
}

void putCharSerial(uint8_t c) {
    while(UART0->txdata.full);           // Wait until ready to transmit
    UART0->txdata.data = c;
}
```

The `main` function in Code Example e9.6 demonstrates printing to the console and reading from the console using the `putStrSerial` and `getStrSerial` functions.

[2] PuTTY prints correctly even if the `\r` is omitted.

Code Example e9.6 READING AND WRITING STRINGS TO A TERMINAL USING A UART

```
#include "EasyREDVIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                          // Read an entire string until detecting
        str[i] = getCharSerial(); // Carriage return
    } while ((str[i++] != '\r') && (i < MAX_STR_LEN));  // Look for carriage return
    str[i-1] = 0;                 // Null-terminate the string
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {         // Iterate over string
        putCharSerial(str[i++]); // Send each character
    }
}

int main(void) {
   char str[MAX_STR_LEN];

   uartInit(115200); // initialize UART with baud rate

   while(1) {
       putStrSerial("Please type something: \r\n");
       getStrSerial(str);
       putStrSerial("You typed: ");
       putStrSerial(str);
       putStrSerial("\r\n");
   }
}
```

Communicating with the serial port from a C program on a PC is a bit of a hassle because serial port driver libraries are not standardized across operating systems. Other programming environments such as Python, MATLAB, or LabVIEW make serial communication painless.

9.3.6 Timers

Embedded systems commonly need to measure time. For example, a microwave oven needs a timer to keep track of the time of day and another to measure how long to cook. It might use yet another to generate pulses to the motor spinning the platter and a fourth to control the power setting by only activating the microwave's energy for a fraction of every second.

The FE310 has a system timer with a 64-bit free-running counter that increments according to an externally provided clock signal. On the RED-V, this clock source is a 32.768 kHz oscillator (conveniently 2^{15} Hz). Figure e9.16 shows the memory map for the system timer. It is located within the core-local interruptor (CLINT) block. `mtime` contains the 64-bit current value of the counter. It can be read or written; so, to restart the timer, a zero can be written. mtimecmp is a 64-bit register containing the timer comparison value and msip is the machine-mode software interrupt register. When the counter hits the value in mtimecmp, the least significant bit in the msip register is set to 1. Using the msip and mtimecmp registers is an efficient way to check that a delay has taken place. Table e9.7 shows the memory addresses for these registers.

If additional timers are needed, the PWM module (see Section 9.3.7.2) provides additional counters that can be used to measure precise delays.

Table e9.7 System timer registers

	...
0x0200BFFC	mtime (hi)
0x0200BFF8	mtime (lo)
	...
0x02004004	mtimecmp (hi)
0x02004000	mtimecmp (lo)
	...
0x02000000	msip
	...

Adapted and printed with permission from Table 24 of the SiFive *FE310-G002 Manual,* © 2019 SiFive, Inc.

Example e9.5 BLINKING LED

Write a program that blinks the status LED on the RED-V 5 times per second for 4 seconds.

Solution The delay function in EasyREDVIO (see Code Example e9.7) creates a delay of a specified number of milliseconds using the timer compare channel.

Code Example e9.7 DELAY FUNCTION

```
#define MTIME_CLK_FREQ 32768              // RTC frequency in Hz
volatile uint64_t *mtime = (uint32_t*) 0x0200BFF8;
void delay(int ms) {
    uint64_t doneTime = *mtime + (ms*MTIME_CLK_FREQ)/1000;
    while (*mtime < doneTime);            // Wait until time is reached
}
```

GPIO5 (D13) drives the activity LED on the RED-V board. The program in Code Example e9.8 sets this pin to be an output. It then turns the LED OFF and ON through a series of digital writes with a 200 ms repetition rate (5 Hz).

Code Example e9.8 BLINK ACTIVITY LED

```
#include "EasyREDVIO.h"

void main(void) {
    uint32_t i;

    pinMode(D13, OUTPUT);     // status led as output

    for(i = 0; i < 20; i++) {
        delay(100);
        digitalWrite(D13, 0); // turn led off
        delay(100);
        digitalWrite(D13, 1); // turn led on
    }
}
```

9.3.7 Analog I/O

The real world is an analog place. Many embedded systems need analog inputs and outputs to interface with the world. They use analog-to-digital converters (ADCs) to quantize analog signals into digital values and digital-to-analog-converters (DACs) to do the reverse. Figure e9.15 shows symbols for these components. Such converters are characterized by their resolution, dynamic range, sampling rate, and accuracy. For example, an ADC might have N = 12-bit resolution over a range V_{ref}^{-} to V_{ref}^{+} of 0 to 5 V with a sampling rate of f_s = 44 kHz and an accuracy of ±3 least significant bits (lsbs). Sampling rates are also listed as samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{\text{in}}(t)$ and the digital sample $X[n = t / f_s]$ is

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref}^{-}}{V_{ref}^{+} - V_{ref}^{-}} \tag{e9.2}$$

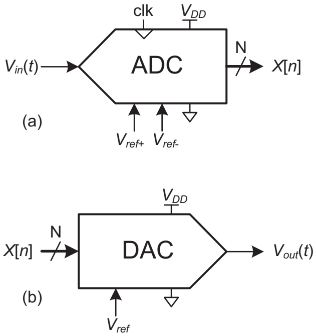


Figure e9.15 **ADC and DAC symbols**

For example, an input voltage of 2.5V (half of full scale) would correspond to $2^{12}/2$ (half of the maximum value), that is, an output of $100000000000_2 = 0x800 = 2^{11} = 2048$, with an uncertainty of up to 3 lsbs.

Similarly, a DAC might have N = 16-bit resolution over a full-scale output range of V_{ref} = 2.56 V. It produces an output of

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref} \tag{e9.3}$$

Many microcontrollers have built-in ADCs of moderate performance. For higher performance (e.g., 16-bit resolution or sampling rates in excess of 1 MHz), it is often necessary to use a separate ADC connected to the microcontroller. Fewer microcontrollers have built-in DACs, so separate chips must be used to convert digital values to an analog voltage. However, microcontrollers often produce analog outputs using a technique called pulse-width modulation (PWM).

D/A Conversion

The FE310 has no general-purpose DAC. This section describes D/A conversion using external DACs and illustrates interfacing the FE310 with other chips over parallel and serial ports. The next section achieves the same result using PWM.

Some DACs accept the N-bit digital input on a parallel interface with N wires, while others accept it over a serial interface, such as SPI. Some DACs require both positive and negative power supply voltages, while others operate off of a single supply. Some support a flexible range of supply voltages, while others demand a specific voltage. The input logic levels should be compatible with the digital source. Some DACs produce a voltage output proportional to the digital input, while others produce a current output; an operational amplifier may be needed to convert this current to a voltage in the desired range.

In this section, we use the Linear Technology LTC1450 12-bit parallel DAC and the Microchip MCP4801 8-bit serial DAC. Both produce voltage outputs, run off a single 5 to 15 V power supply, use $V_{IH} = 2.4V$ such that they are compatible with 3.3 V I/O, come in DIP packages that make them easy to breadboard, and are easy to use. The LTC1450 produces an output on a scale of 0 to 2.048 V or 0 to 4.095 V depending on the gain setting, consumes 2 mW, comes in a 24-pin package, and has a 4 μs settling time, permitting an output rate of 250 ksamples/second. The datasheet is at analog.com. The MCP4801 produces an output on a scale of 0 to 2.048 V or 0 to 4.096 V, consumes less than 2 mW, comes in an 8-pin package, and has a 4.5 μs settling time. Its SPI operates at a maximum of 20 MHz. The datasheet is at microchip.com.

Example e9.6 ANALOG OUTPUT WITH EXTERNAL DACS

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a RED-V, an LTC1450, and an MCP4801.

Solution The circuit is shown in Figure e9.16. Two DAC chips are used in this example. Both DACs use a 5 V power supply and have a 0.1 μF decoupling capacitor to reduce power supply noise.

The LTC1450 DAC has 12 data inputs, D0 to D11, that specify the analog voltage to generate on VOUT. In our example, we use only 8-bit precision, so we tie the four least significant bits, D0 to D3, to ground. To load data into the DAC, the RED-V puts the desired value on D4 to D11. Then, the RED-V drives the active-low write (WR) pin low to write the data to the DAC. CLR is tied to VCC because we don't need to clear the input data latches. LDAC, the low-asserted load DAC signal, is tied to GND to load data every time WR goes low.

The MCP4801 connects to the RED-V via SPI1. In addition to the standard SPI signals, the MCP4801 has an analog output voltage pin (VOUT) and two active-low control inputs: hardware shutdown (SHDN) and latch DAC (LDAC). SHDN is used to turn off the output driving circuitry and save power when the output value is not needed. The LDAC latches the input values when it is low. To send data to the MCP4801, a 16-bit value is sent over SPI: bits 11 to 4 hold D7 to D0; bit 13 is the gain selection (1x if set to 1, 2x if set to 0); and bit 12 controls SHDN (0 shuts down the output, 1 allows VOUT to drive an analog value). In this case, SHDN is controlled in software, so it is left floating (not driven) in the circuit.

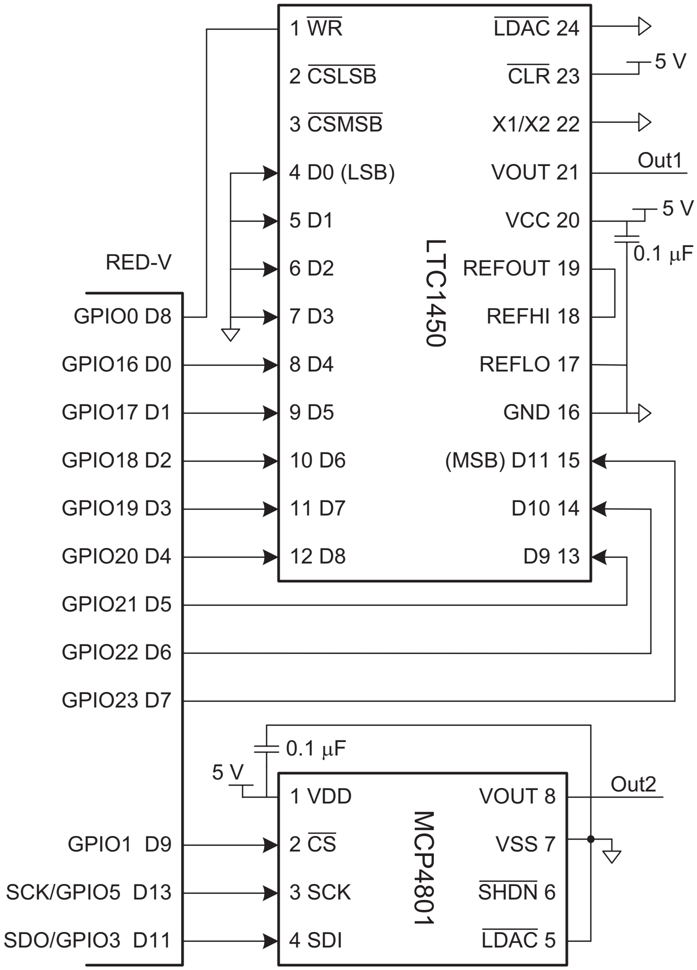


Figure e9.16 DAC parallel and serial interfaces to a RED-V board

The program for driving both DACs is shown in Code Example e9.9. The program configures the 8 parallel port pins as outputs and also configures GPIO0 as an output to drive the `WR` signal on the LTC1450 and GPIO1 to drive the chip select signal on the MCP4801. It initializes the SPI to 500 kHz. `initWaveTables` precomputes an array of sample values for the sine and triangle waves. It then updates the serial DAC. Then, the program delays until the timer indicates that it is time for the next sample. The maximum frequency of the generated waveforms is set by the time to send each point in the `genWaves` function, which is limited by the SPI transmission time.

Code Example e9.9 GENERATING A SINE WAVE USING A DAC

```c
#include "EasyREDVIO.h"
#include <math.h> // required to use the sine function

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];

#define SHDNn_Pos 12
#define Gain_Pos  13

int parallelPins[8] = {D0, D1, D2, D3, D4, D5, D6, D7};

void initWaveTables(void) {
  int i;
  for (i = 0; i<NUMPTS; i++) {
    sine[i] = 127*(sin(2*3.14159*i/NUMPTS) + 1);  // 8-bit scale
    if (i < NUMPTS/2) triangle[i] = i*255/NUMPTS;  // 8-bit scale
    else triangle[i] = 254 - i*255/NUMPTS;
  }
}

void genWaves(int freq) {
  int i, j;
  int delay_cycles = MTIME_CLK_FREQ/(NUMPTS*freq);

  for (i = 0; i < 2000; i++){
    for (j = 0; j < NUMPTS; j++) {
      uint64_t doneTime = *mtime + delay_cycles; // Set sample period

      // Load serial DAC
      digitalWrite(1, 0);       // enable chip (chip select: CS = 0)
      // Set SHDNn to active (bit 12) and gain to 1 (bit 13)
      volatile uint16_t sine_samp_dac = ((uint16_t) sine[j] << 4) \
                                        |(1 << SHDNn_Pos) | (1 << Gain_Pos);
      spiSendReceive16(sine_samp_dac);
      digitalWrite(1, 1);       // disable chip (chip select: CS = 1)

      // Load parallel DAC
      digitalWrite(0, 1);       // No load while changing inputs
      digitalWrites(parallelPins, 8, triangle[j]);
      digitalWrite(0, 0);       // Load new points into DACs
      while(*mtime < doneTime); // Wait for mtime_cmp to hit
    }
  }
}

int main(void) {
  pinsMode(parallelPins, 8, OUTPUT); // Set pins connected to the AD558 as outputs
  pinMode(0, OUTPUT);                // Make pin 0 an output to control LOAD
  pinMode(1, OUTPUT);                // Make pin 1 an output to control CE
  spiInit(15, 0, 0);                 // Initialize the SPI
  initWaveTables();
  genWaves(100);
}
```

Pulse-Width Modulation

Another way for a digital system to generate an analog output is with *pulse-width modulation* (PWM), in which a periodic output is pulsed high for part of the period and low for the remainder. The duty cycle is the fraction of the period for which the pulse is high (pulse width/period), as shown in Figure e9.17. The average value of the output is

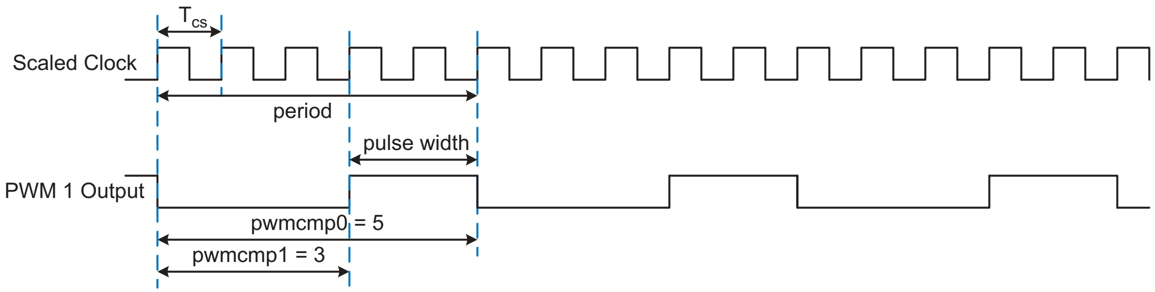


Figure e9.17 **Pulse-width modulated (PWM) signal**

proportional to the duty cycle. For example, if the output swings between 0 and 3.3 V and has a duty cycle of 25%, the average value will be 0.25 × 3.3 = 0.825 V. Low-pass filtering a PWM signal eliminates the oscillation and leaves a signal with the desired average value. Thus, PWM is an effective way to produce an analog output if the pulse rate is much higher than the analog output frequencies of interest. Other applications of PWM include making square wave audio tones and digital control of a motor or light at partial power or brightness.

The FE310 has three PWM peripherals and, as shown in Table e9.3, each PWM has four PWM outputs, for a total of 12 available PWM outputs. The outputs on PWM0 have 8-bit precision, and PWM1 and PWM2 have 16-bit precision. In this section, we show how to use PWM2, but configuring and using the other two PWM peripherals follows similar steps. PWM2 has four outputs (PWM2_PWM0, PWM2_PWM1, PWM2_PWM2, PWM2_PWM3) that are available on pins GPIO10-13 using pin function IOF1.

PWMs have several waveform generation modes, but we focus on generating PWM waveforms such as those in Figure e9.17. To do this, the peripheral is configured in a repeating mode in which comparator 0 (`pwmcmp0`) sets the period and comparator 1 (`pwmcmp1`) sets the low time. These times are in units of a scaled clock period, T_{cs}. For example, as shown in Figure e9.17, if the scaled clock period is 0.5 μs (2 MHz) and `pwmcmp0` = 5, then PWM2_PWM1 (pin 11) will oscillate at a period of 5 × 0.5 μs = 2.5 μs (400 kHz). If `pwmcmp1` = 3, then the duty cycle is 1−(3/5) = 40%.

Table e9.8 shows the memory map for the PWM2 registers. In this section, we describe the steps needed to configure the PWM1_PWM1 output; the other PWMs and their outputs are configured using a similar procedure.

Table e9.9 shows the bitfields in the PWM configuration register `pwmcfg`. Note that most bits are not cleared on system reset,

Table e9.8 **PWM2 configuration registers**

0x1002502C	pwmcmp3
0x10025028	pwmcmp2
0x10025024	pwmcmp1
0x10025020	pwmcmp0
...	...
0x10025010	pwms
0x1002500C	...
0x10025008	pwmcount
0x10025004	...
0x10025000	pwmcfg
	...

Adapted and printed with permission from Table 89 of the SiFive *FE310-G002 Manual*, © 2019 SiFive, Inc.

Table e9.9 PWM configuration register fields

PWM Configuration Register (pwmcfg)				
Register Offset	0x0			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	pwmscale	RW	X	PWM Counter scale
[7:4]	Reserved			
8	pwmsticky	RW	X	PWM Sticky - disallow clearing pwmcmp$X$ip bits
9	pwmzerocmp	RW	X	PWM Zero - counter resets to zero after match
10	pwmdeglitch	RW	X	PWM Deglitch - latch pwmcmp$X$ip within same cycle
11	Reserved			
12	pwmenalways	RW	0x0	PWM enable always - run continuously
13	pwmenoneshot	RW	0x0	PWM enable one shot - run one cycle
[15:14]	Reserved			
16	pwmcmp0center	RW	X	PWM0 Compare Center
17	pwmcmp1center	RW	X	PWM1 Compare Center
18	pwmcmp2center	RW	X	PWM2 Compare Center
19	pwmcmp3center	RW	X	PWM3 Compare Center
[23:20]	Reserved			
24	pwmcmp0gang	RW	X	PWM0/PWM1 Compare Gang
25	pwmcmp1gang	RW	X	PWM1/PWM2 Compare Gang
26	pwmcmp2gang	RW	X	PWM2/PWM3 Compare Gang
27	pwmcmp3gang	RW	X	PWM3/PWM0 Compare Gang
28	pwmcmp0ip	RW	X	PWM0 Interrupt Pending
29	pwmcmp1ip	RW	X	PWM1 Interrupt Pending
30	pwmcmp2ip	RW	X	PWM2 Interrupt Pending
31	pwmcmp3ip	RW	X	PWM3 Interrupt Pending

Reprinted with permission from Table 91 of the SiFive *FE310-G0002 Manual,* © 2019 SiFive, Inc.

so it is prudent to start by resetting all bits to 0, then writing a 1 to pwmenalways and pwmzerocmp to configure the PWM to generate a repeating waveform with the period set by pwmcmp0.

The scaled clock frequency f_{scaled} is the base bus clock frequency of $f_{\text{base}} = 16\,\text{MHz}$ divided by $2^{\texttt{pwmscale}}$, where pwmscale is a 4-bit number in the range of 0 to 15 in the pwmcfg register. The PWM frequency is $f_{\text{pwm}} = f_{\text{scaled}}/\texttt{pwmcmp0} = f_{\text{base}}/(\texttt{pwmcmp0} \times 2^{\texttt{pwmscale}})$. As described above, the duty cycle is $1-(\texttt{pwmcmp1}/\texttt{pwmcmp0})$. Many possible choices exist for pwmscale and pwmcmp0 to give a desired PMW frequency. However, the PWM frequency resolution (error between desired and actual frequency) is best when pwmscale is as small as possible and pwmcmp0 is as large as possible, within the constraint that pwmcmp0 is an unsigned 16-bit number (i.e., it cannot exceed 65,535).

Example e9.7 PULSE-WIDTH MODULATION (PWM)

Choose `pwmscale` and `pwmcmp0` to blink an LED at 1.2 Hz. Repeat the question to generate a tone of 1190 Hz.

Solution To illustrate this, suppose that we wished to blink an LED at f_{pwm} = 1.2 Hz. $f_{pwm} = f_{base}/(\texttt{pwmcmp0} \times 2^{\texttt{pwmscale}})$. Thus, choose `pwmscale` = 8 and `pwmcmp0` = 52083.33 to get the desired frequency with f_{scaled} = 16 MHz/2^8 = 62.5 KHz. `pwmcmp0` is a 16-bit register, so we must round to 52083, giving an actual f_{pwm} = 16 MHz/(52083 × 2^8) = 1.20000768 Hz, which is very close to the desired frequency and comparable to the 10 parts per million accuracy of a typical quartz crystal clock reference.

On the other hand, suppose that we wanted a 1190 Hz output. If we didn't change `pwmscale`, `pwmcmp0` would need to be 52.521. Rounding to 53 gives an actual f_{pwm} = 16 MHz/(53 × 2^8) = 1179.2 Hz, an error of about 10 Hz, or 0.91%. If we needed a more accurate output frequency, we could reduce `pwmscale` to 0 and increase `pwmcmp0` to 13445, obtaining f_{pwm} = 16 MHz/(13445 × 2^0) = 1190.03 Hz.

A PWM device driver could have `pwmInit()` and `pwm(int freq, float duty)` functions. `pwmInit` would set the appropriate pin for the PWM peripheral and set the bits in the `pwmcfg` register. The `pwm` function would choose the appropriate `pwmscale`, `pwmcmp0`, and `pwmcmp1` to generate a waveform with the specified frequency and duty cycle. Writing these functions is similar to writing the SPI or UART device driver; details are left as an exercise for the reader.

A/D Conversion

Many microcontrollers have at least one built-in ADC, but the FE310 does not. This section describes A/D conversion using an external converter similar to the external DACs described in the prior section.

Example e9.8 ANALOG INPUT WITH AN EXTERNAL ADC

Interface a 10-bit MCP3002 A/D converter to an FE310 using SPI and print the input value. Set a full-scale voltage of 3.3 V. Search for the datasheet on the web for full details of operation.

Solution Figure e9.18 shows a schematic of the connection between the FE310 and the MCP3002 ADC and Code Example e9.10 shows the driver code. The MCP3002 uses VDD as its full-scale reference: that is, VDD (Pin 8) is connected to 3.3 V. It can accept a 3.3 to 5.5 V supply; we choose 3.3 V. The ADC

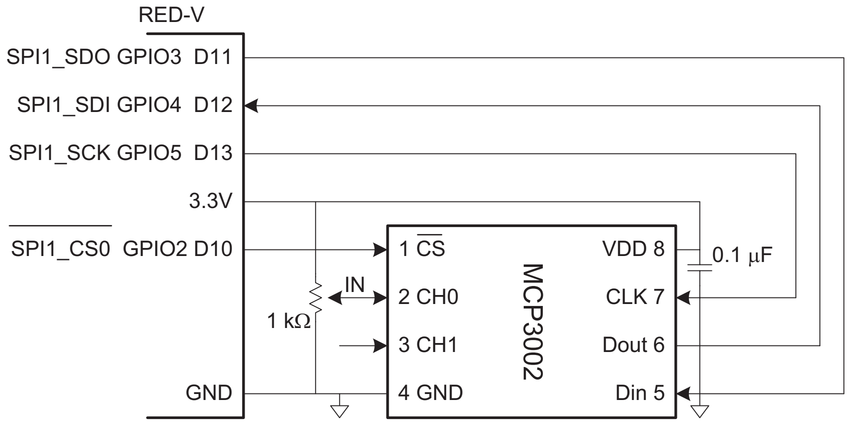


Figure e9.18 **Reading an analog input using an external ADC**

has two input channels, CH0 and CH1. We connect channel 0 to a potentiometer (not shown in the figure) that we rotate to adjust the input voltage between 0 and 3.3 V.

The FE310 code (see Code Example e9.10) initializes the SPI and repeatedly reads and prints samples. According to the datasheet, the FE310 must send the 16-bit quantity 0x6000 over SPI to read CH0 and will receive the 10-bit result back in the bottom 10 bits of the 16-bit result. Since we cannot directly set the FE310 to transmit 16-bit frames, we can put together two 8-bit packets without raising the chip select line in between. Although the SPI peripheral has the option to control the chip select line automatically, here we manually configure GPIO2 as an output and toggle it appropriately at the beginning of the transmission (writing the chip select line to 0) and the end of the transmission (writing the chip select line to 1).

Code Example e9.10 CODE FOR INTERFACING WITH ADC

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t sample;
  spiInit(15, 0, 0);    // Initialize the SPI
                        // Clock divisor of div = 15, CPOL = 0, CPHA = 0
  pinMode(D10,  OUTPUT);
  while(1) {
    digitalWrite(D10, 0);
    spiSendReceive('0x60');
    sample = spiSendReceive('0x00');
    digitalWrite(D10, 1);
    printf("Read %d\n", sample);
    delay(200);
   }
}
```

9.3.8 Interrupts

So far, we have relied on *polling*, in which the program continually checks a value until an event occurs such as data arriving on a UART or a timer reaching its compare value. This can be a waste of the processor's power and makes it difficult to write programs that do interesting work while simultaneously waiting for events to occur.

Most microcontrollers support *interrupts*. When an event occurs, the microcontroller stops the executing program and jumps to an interrupt handler that responds to the interrupt. After handling the interrupt, the processor then returns to the user program and seamlessly continues where it was interrupted. Interrupts are the hardware exceptions discussed in Section 6.6.2.

The FE310 has a core-local interruptor (CLINT) that handles timer and software interrupts. Software interrupts are used for interprocessor communication and debugging. The FE310 also has a platform-level interrupt controller (PLIC) that collects interrupts from other peripherals. In a multiprocessor system, the PLIC routes the peripheral interrupt to an appropriate processor to handle it.

Example e9.9 shows how to blink an LED using interrupts instead of polling.

Example e9.9 BLINKING AN LED WITH A TIMER INTERRUPT

We configure local interrupts on the FE310 using the CLINT. For the FE310-G002 chip used on the RED-V RedBoard and RED-V Thing Plus, information on how to use interrupts is provided in Chapters 8 to 10 of the *FE310-G002 Manual*. The basic configuration procedure for local interrupts through the CLINT is outlined below.

1. Write a trap handler to handle execution whenever an interrupt or exception is triggered. The main purpose of the trap handler is to figure out what interrupt or exception was triggered and then to perform the desired operation in response.
2. Configure `mtvec`, a control and status register (CSR), with the address of the trap handler and the mode (direct or vectored).
3. Enable the specific interrupt (e.g., from the timer)
4. Globally enable all interrupts.

After defining constants and function pointer arrays, the code declares the global trap handler function `handle_trap()`, as shown in Code Example e9.11. This function is called whenever we trigger a trap (interrupt or exception). Its job is to figure out which trap triggered the call and jump to the correct interrupt or exception handler. The trap handler performs two tasks. First, it uses a mask (`MCAUSE_INT_MASK`) to check the most significant bit of the `mcause` register, which indicates whether the trap is an interrupt (generated from a device external to the core) or an exception (generated internally in the core). The structure of mcause is shown in Table e9.10 and the listing of interrupt and exception codes is shown in Table 6.6. Then, it uses an additional mask (MCAUSE_CODE_MASK) to determine the trap code and jumps to the appropriate interrupt or exception handler based on the index of the interrupt_handler or exception_handler function pointer arrays.

Code Example e9.11 SETTING UP THE TRAP HANDLER

```
// Function pointer arrays for interrupt and exception handlers
#define MAX_INTERRUPTS 16
void (*interrupt_handler[MAX_INTERRUPTS])();
void (*exception_handler[MAX_INTERRUPTS])();

// Masks for isolating interrupt vs. exception and the relevant code
#define MCAUSE_INT_MASK 0x80000000  // If [31] = 1 interrupt, else exception
#define MCAUSE_CODE_MASK 0x7FFFFFFF // low bits show code

// Declaration for interrupt handler. Declared with attribute interrupt which
// maps to GCC helper function.
void handle_trap(void) __attribute((interrupt));

// Define trap handler
void handle_trap() {
    unsigned long mcause_value = read_csr(mcause);
    if (mcause_value & MCAUSE_INT_MASK) {
        // Branch to interrupt handler here
        // Index into 32-bit array containing addresses of functions
        interrupt_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
    else {
        // Branch to exception handler here
        exception_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
}
```

Next, we define an *interrupt service routine* (*ISR*) for the timer. This is a function that contains instructions we want to execute whenever we get a timer interrupt. In this example, we call this function timer_handler(). It reads the current value of the GPIO pin driving the on-board LED (D13/GPIO5) and toggles the state using digitalWrite().

Table e9.10 mcause register fields

Bits	Field Name	Description
[9:0]	Exception Code	A code identifying the most recent exception
[30:10]	Reserved	
31	Interrupt	1 if trap was caused by an interrupt; 0 otherwise

Reprinted with permission from Table 22 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Then, it resets the timer by calling `reset_timer()`, which sets the current count in the `mtime` register to 0 and resets the count value at which the next interrupt should be triggered.

Code Example e9.12 TIMER ISR AND FUNCTION TO RESET TIMER

```
void timer_handler() {
    volatile int pin_val = (GPIO0->output_val >> D13) & 1; // Read the current output state
    if(pin_val) digitalWrite(D13, LOW);
    else digitalWrite(D13, HIGH);
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));
}

void reset_timer(int count_val) {
    *MTIME = 0;
    *MTIMECMP = count_val;
}
```

Unlike the other registers we have used in this chapter, most of the registers related to the CLINT—such as `mtvec`, `mie`, and `mstatus`—are not memory mapped. These registers are called *control and status registers* (*CSRs*). To manipulate CSRs, we must use the RISC-V assembly instructions CSR read (`csrr`) and CSR write (`csrw`). These instructions can conveniently be wrapped in C macros to enable us to more easily interact with them.

Code Example e9.13 MACROS FOR WRITING AND READING CSRs

```
// Macros for reading and writing the control and status registers (CSRs)
#define read_csr(reg) ({ unsigned long __tmp; \
 asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
 __tmp; })

#define write_csr(reg, val) ({ \
 asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })
```

After setting up the trap handler, we register it by placing its address in the `mtvec` register. Its structure is shown in Table e9.11. `mtvec` is a 32-bit register where bits [31:2] hold bits [31:2] of the address of the trap handler function (bits [1:0] are automatically assumed to be 0 since the instructions must be word aligned in the memory). Bits [1:0] of `mtvec` are instead used to configure whether the exceptions are handled in direct or vector mode. In direct mode, regardless of what interrupt or exception fires, we jump to the function address indicated by `mtvec[31:2]`. This is the mode we will use here. In vectored mode, we jump to different memory addresses depending on what interrupt is triggered.

After configuring the trap handler and setting up the timer interrupt service routine, we finish by enabling the machine timer interrupt by setting bit 7, the machine timer interrupt enable (`MTIE`) bit, in the machine interrupt enable `mie` register and by enabling interrupts globally by

Table e9.11 `mtvec` **register fields**

Bits	Field Name	Description
[1:0]	MODE	Sets the interrupt processing mode to direct (`00`) or vectored (`10`)
[31:2]	BASE[31:2]	Base address of the `trap_handler`

Adapted and printed with permission from Table 18 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Code Example e9.14 FUNCTIONS TO REGISTER TRAP HANDLER BY WRITING TO `mtvec`

```
void register_trap_handler(void *func) {
    // Set mtvec[31:2] to interrupt handler function address
    // The two lsbs are not meaningful because instructions are aligned to 4 bytes
    // Set mtvec[1:0] to 00 for direct mode.
    write_csr(mtvec, ((unsigned long) func) & ~(0b11));
}
```

setting bit 3, the machine interrupt enable (`MIE`) bit, in the machine status register (`mstatus`). Simple helper functions for globally enabling and disabling interrupts are shown in Code Example e9.15. Complete details about the structure of `mstatus` and `mie` can be found in Tables 17 and 20 in the SiFive *FE310-G002 Manual*.

Code Example e9.15 GLOBALLY ENABLE OR DISABLE INTERRUPTS

```
void enable_interrupts() {
    // Set bit 3 in mstatus (MIE) to enable machine interrupts
    write_csr(mstatus, read_csr(mstatus) | (1 << 3));

void disable_interrupts() {
    // Clear bit 3 in mstatus (MIE) to disable machine interrupts
    write_csr(mstatus, read_csr(mstatus) & ~(1 << 3));
}
```

Finally, we put all the pieces together and call the functions we built in our main function, as shown in Code Example e9.16. Here, because our application is interrupt driven, we don't do anything in the main `while` loop.

Care should be taken when developing safety- or timing-critical applications with interrupts as they are asynchronous events and can be triggered at any time during program execution. You as a programmer should consider what bugs may be introduced by an interrupt triggering at an inopportune time. If you have a segment of code where you want to avoid being interrupted, you can disable interrupts (i.e., clear `MIE` in

Code Example e9.16 BLINK LED WITH TIMER INTERRUPTS

```
#include "EasyREDVIO.h"

// CLINT memory map pointers
#define MTIMECMP ((uint64_t *) 0x02004000UL)
#define MTIME ((uint64_t *) 0x0200BFF8UL)

#define BLINK_FREQ 4 // This is an arbitrary constant used to specify the LED blink
frequency

int main(void) {

    // Set LED pin as an output
    pinMode(D13, OUTPUT);

    // Register interrupt handler.
    // The machine timer interrupt is exception code 7 as shown in  so we put the
    // timer_handler() function at index 7 of the array.
    interrupt_handler[7] = timer_handler;

    // Set up interrupt by configuring mtvec
    register_trap_handler(handle_trap);

    // Reset timer
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));

    // Enable timer interrupt
    write_csr(mie, read_csr(mie) | (1 << 7));

    enable_interrupts();

    while(1) {
    };

    return 0;
}
```

`mstatus`) when executing the instructions and then re-enable interrupts when finished (i.e., set `MIE` in `mstatus`).

9.4 OTHER MICROCONTROLLER PERIPHERALS

Microcontrollers frequently interface with other external peripherals. This section describes a variety of common examples, including character-mode liquid crystal displays (LCDs), VGA monitors, Bluetooth wireless links, and motor controllers.

9.4.1 Character LCDs

A character LCD is a small liquid crystal display capable of showing one or a few lines of text. They are commonly used in the front panels of appliances such as cash registers, laser printers, and fax machines that need to display a limited amount of information. They are easy to interface with a microcontroller over parallel, RS-232, or SPI interfaces. Crystalfontz America sells a wide variety of character LCDs ranging from 8 columns × 1 row to 40 columns × 4 rows with choices of color, backlight, 3.3 or 5 V operation, and daylight visibility. Their LCDs can cost $20 or more in small quantities, but prices come down to under $5 in high volume.

This section shows how to interface the RED-V board to the Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD shown in Figure e9.19. The interface is an 8-bit parallel interface, which is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi.

Figure e9.20 shows the LCD connected to a RED-V board over an 8-bit parallel interface (inputs D0-D7 on the LCD). The LCD logic operates at 5 V but is compatible with 3.3 V inputs from the RED-V board. The LCD contrast is set by a second voltage (input to pin 3, VO) produced using a potentiometer; it is usually most readable at a setting of 4.2 to 4.8 V. The LCD receives three control signals: RS (1 for characters, 0 for instructions), $R/\overline{W}$ (1 to read from the display, 0 to write), and E (pulsed high for at least 250 ns to enable the LCD when the next data byte is ready to be written to it). In addition to sending data bits, the data lines (D0–D7) are used to set LCD configurations when RS = 0 (i.e., when in instruction mode). When read, LCD port D7 returns the busy flag, which is 1 when the LCD is busy and 0 when it is ready to accept another instruction or byte of data.

To initialize the LCD, the RED-V board must write a sequence of instructions to the LCD as given in Table e9.12. Instructions are written by making RS = 0 and $R/\overline{W} = 0$, putting the value on the eight data lines, and pulsing E for at least 250 ns. Data bytes are written by doing the same thing except making RS = 1. After sending an instruction or data byte, the processor must wait for at least a specified amount of time (or sometimes until the busy flag is clear) before sending another instruction or data byte. The busy flag (D7) is read by making RS = 0 and $R/\overline{W} = 1$ and pulsing E for at least 250 ns. Remember that GPIO23 must also be temporarily set as an input when reading the busy flag (D7).

After configuration is complete, the LCD is ready to accept text to display. Write text to the LCD by making RS = 1 and $R/\overline{W} = 0$, putting

Figure e9.19 **Crystalfontz CFAH2002A-TMI 20 × 2 character LCD**
© 2012 Crystalfontz America; reprinted with permission.

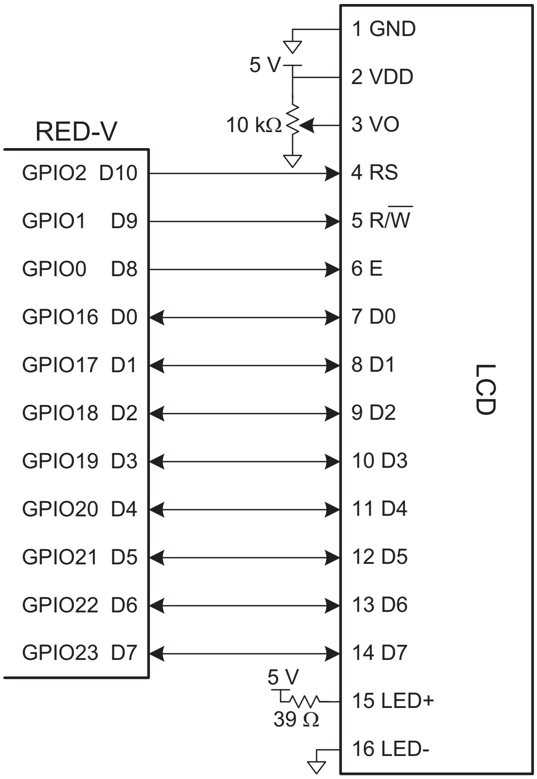


Figure e9.20 **Parallel LCD interface**

the value on the eight data lines, and pulsing E for at least 250 ns. After each character, the RED-V must wait for the busy bit to clear before sending another character. It may also send the instruction 0x01 to clear the display or 0x02 to return to the home position in the upper left.

Example e9.10 LCD CONTROL

Write a program to print "I love LCDs" to the Crystalfontz CFAH2002A-TMI character display.

Solution The program in Code Example e9.17 writes "I love LCDs" to the display by initializing the display and then sending the characters.

9.4.2 VGA Monitor

A more flexible display option is to drive a computer monitor. This section explains the low-level details of driving a VGA (*video graphics array*) monitor directly from an FPGA.

Code Example e9.17 WRITING “I LOVE LCDS” TO LCD

```
#include "EasyREDVIO.h"

int LCD_IO_Pins[] = {D0, D1, D2, D3, D4, D5, D6, D7};

typedef enum {INSTR, DATA} mode;
#define RS D10
#define RW D9
#define E  D8

char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md == DATA));      // set instr/data mode
    digitalWrite(RW, 1);                // RWbar = read mode
    digitalWrite(E, 1);                 // pulse enable
    delay(1);                           // wait for LCD response
    c = digitalReads(LCD_IO_Pins, 8);   // read a byte from parallel port
    digitalWrite(E, 0);                 // turn off enable
    delay(1);
    return c;
}

void lcdBusyWait(void) {
    char state;
    do {
       state = lcdRead(INSTR);
    } while(state & 0x80);
}

void lcdWrite(char val, mode md) {
    pinsMode(LCD_IO_Pins, 8, OUTPUT);
    digitalWrite(RS, (md == DATA));     // set instr/data mode. OUTPUT = 1, INPUT = 0
    digitalWrite(RW, 0);                // set RW pin to write  (RW = 0)
    digitalWrites(LCD_IO_Pins, 8, val);// write the char to the parallel port
    digitalWrite(E, 1); delay(1);       // pulse E
    digitalWrite(E, 0); delay(1);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delay(1);
}

void lcdPrintString(char* str) {
    while (*str ! = 0) {
        lcdWrite(*str, DATA); lcdBusyWait();
        str++;
    }
}

void lcdInit(void) {
    pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
    // send initialization routine:
    delay(15);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

int main(void) {
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

Table e9.12 LCD initialization sequence

Code (D7-D0)	Purpose	Wait (µs)
(apply VDD)	Allow device to turn on	15000
`0x30`	Set 8-bit mode	4100
`0x30`	Set 8-bit mode again	100
`0x30`	Set 8-bit mode yet again	Until busy flag is clear
`0x3C`	Configure 2 lines and 5 × 8 dot font	Until busy flag is clear
`0x08`	Turn display OFF	Until busy flag is clear
`0x01`	Clear display	1530
`0x06`	Set entry mode that increments cursor after each character	Until busy flag is clear
`0x0C`	Turn display ON with no cursor	

(These are instructions: so RS = 0 and $\mathbf{R}/\overline{\boldsymbol{W}} = 0$.)

The VGA monitor standard was introduced in 1987 for the IBM PS/2 computers, with a 640 × 480 pixel resolution on a *cathode ray tube* (CRT) and a 15-pin connector conveying color information with analog voltages. Modern LCD monitors have higher resolution but remain backward compatible with the VGA standard.

In a CRT, an electron gun scans across the screen from left to right, exciting fluorescent material to display an image. Color CRTs use three different phosphors for red, green, and blue, and three electron beams. The strength of each beam determines the intensity of each color in the pixel. At the end of each scan line, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scan lines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. This entire process repeats about 60 to 75 times per second to refresh the fluorescence and give the visual illusion of a steady image. Modern displays typically use LCD technology, which doesn't require the same electron scan gun but uses the same VGA interface timing for compatibility.

In a 640 × 480 pixel VGA monitor, the full frame is actually 800 pixels × 525 horizontal scan lines as shown in Figure e9.21, but only 480 of the scan lines and 640 pixels per scan line actually convey the image, while the remainder are black. A scan line begins with a 48-pixel *back porch*, the blank section on the left edge of the screen. It then contains 640 active pixels, followed by a blank 16-pixel *front porch* at the right edge of the screen and a 96-pixel clock horizontal sync (hsync) pulse to rapidly move

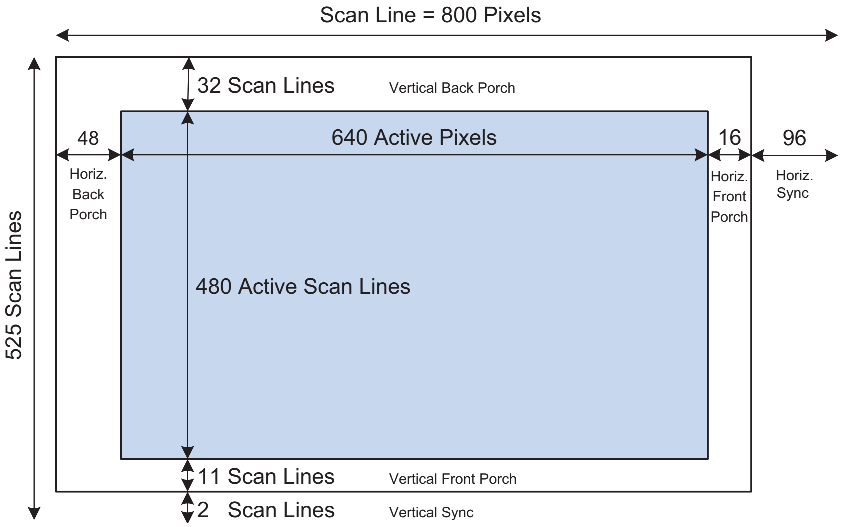


Figure e9.21 VGA frame

the gun back to the left edge. In the vertical direction, the screen starts with a 32-scan line back porch at the top, followed by 480 active scan lines, a front porch of 11 scan lines at the bottom and a 2-scan line vertical sync (vsync) pulse to return to the top to start the next frame. For a 640 × 480 pixel VGA monitor refreshed at 59.52 Hz, the pixel clock operates at 800 × 525 × 59.52 = 25 MHz, so each pixel is 40 ns wide.

Figure e9.22(a) shows the timing of each of the scan lines. The entire scan line is 32 μs long. Figure e9.22(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. A new frame is drawn approximately 60 times per second. Higher resolutions use a faster pixel clock, up to 388 MHz for 2048 × 1536 refreshed at 85 Hz. For example, a 1024 × 768 display refreshed at 60 Hz can be achieved with a 65 MHz pixel clock.

Figure e9.23 shows the pinout for a female connector coming from a video source. Pixel information is conveyed with three analog voltages for red, green, and blue. Each voltage ranges from 0 to 0.7V, with more positive indicating brighter. The voltages should be 0 during the front and back porches. The video signal must be generated in real time at high speed, which is difficult on a microcontroller but easy on an FPGA. A simple black-and-white display could be produced by driving all three color pins with either 0 or 0.7V using a voltage divider connected to a digital output pin. A color monitor, on the other hand, uses a *video DAC* with three separate D/A converters to independently drive the three color pins.

Figure e9.24 shows an FPGA driving a VGA monitor through an ADV7125 triple 8-bit video DAC. The DAC receives 8 bits of R, G, and B from the FPGA. It also receives a SYNC_b signal that is driven active low whenever HSYNC or VSYNC are asserted. The video DAC

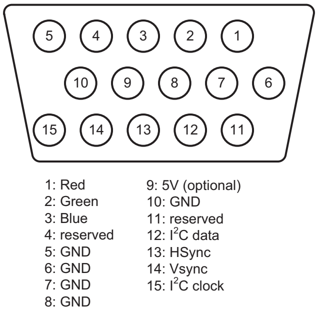


Figure e9.23 **VGA connector pinout**

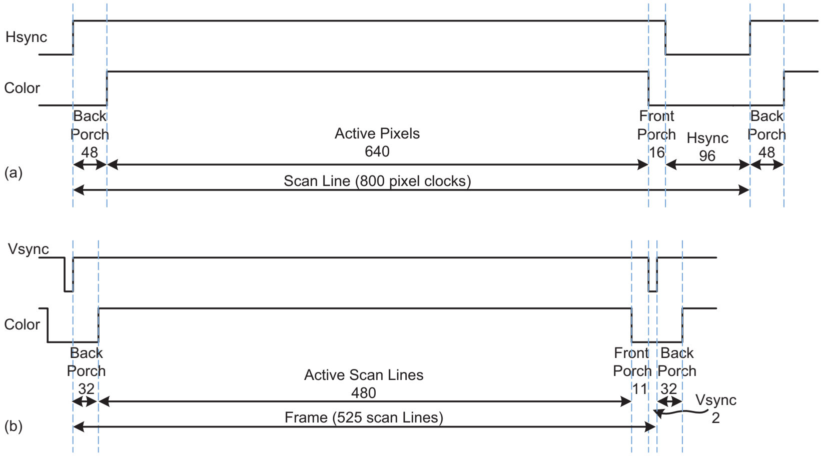


Figure e9.22 **VGA timing: (a) horizontal, (b) vertical**

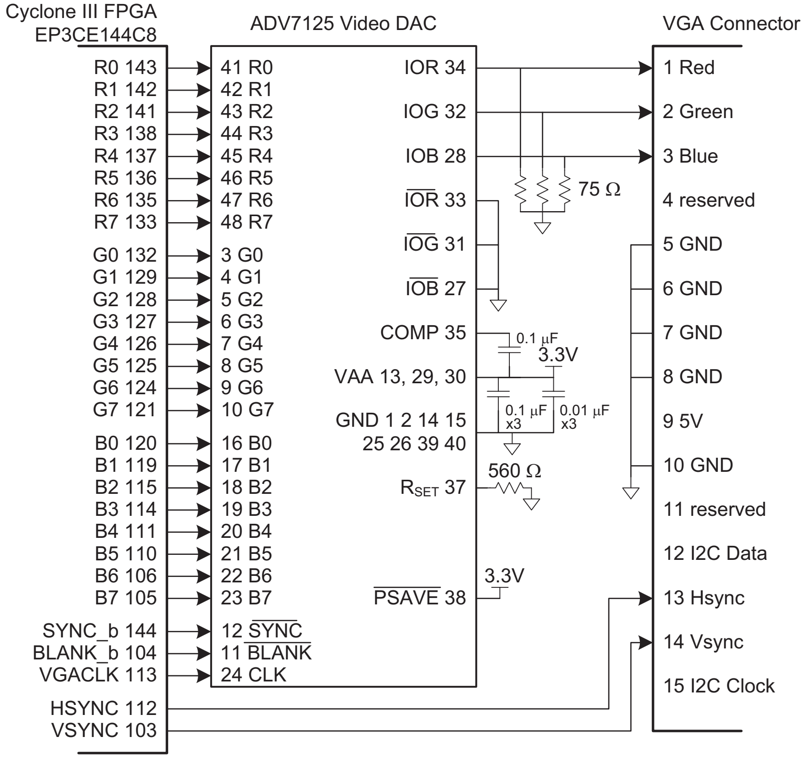


Figure e9.24 **FPGA driving VGA cable through video DAC**

produces three output currents to drive the red, green, and blue analog lines, which are normally 75 Ω transmission lines parallel terminated at both the video DAC and the monitor. The R$_{SET}$ resistor sets the scale of the output current to achieve the full range of color. The clock rate depends on the resolution and refresh rate; it may be as high as 330 MHz with a fast-grade ADV7125JSTZ330 model DAC.

Example e9.11 VGA MONITOR DISPLAY

Write HDL code to display text and a green box on a VGA monitor using the circuitry from Figure e9.24.

Solution The code assumes a system clock frequency of 50 MHz and uses a clock divider to generate the 25 MHz VGA clock. You could also use a PLL to generate the clock. PLL configuration varies among FPGAs; for the Cyclone III, the frequencies are specified with Altera's megafunction wizard. Alternatively, the VGA clock could be provided directly from a signal generator.

The VGA controller counts through the columns and rows of the screen, generating the `hsync` and `vsync` signals at the appropriate times. It also produces a `blank_b` signal that is asserted low to draw black when the coordinates are outside the 640 × 480 active region.

The video generator produces red, green, and blue color values based on the current (x, y) pixel location. (0, 0) represents the upper left corner. The generator draws a set of characters on the screen, along with a green rectangle. The character generator draws an 8 × 8-pixel character, giving a screen size of 80 × 60 characters. It looks up the character from a ROM, where it is encoded in binary as 6 columns by 8 rows. The other two columns are blank. The bit order is reversed by the SystemVerilog code because the leftmost column in the ROM file is the most significant bit, while it should be drawn in the least significant x-position.

Figure e9.25c shows a photograph of the VGA monitor while running this program. The rows of letters alternate red and blue. A green box overlays part of the image.

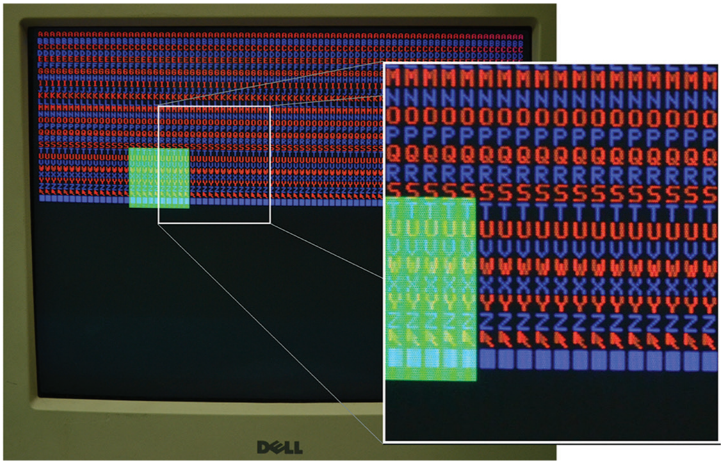


Figure e9.25 VGA output

HDL Example e9.3 vga.sv

```
module vga(input  logic clk, reset,
           output logic vgaclk,            // 25 MHz VGA clock
           output logic hsync, vsync,
           output logic sync_b, blank_b,   // to monitor & DAC
           output logic [7:0] r, g, b);    // to video DAC

  logic [9:0] x, y;

  // divide 50 MHz input clock by 2 to get 25 MHz clock
  always_ff @(posedge clk, posedge reset)
    if (reset) vgaclk = 1'b0;
    else       vgaclk = ~vgaclk;

  // generate monitor timing signals
  vgaController vgaCont(vgaclk, reset, hsync, vsync, sync_b, blank_b, x, y);

  // user-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);

endmodule


module vgaController #(parameter HBP     = 10'd48,   // horizontal back porch
                                 HACTIVE = 10'd640,  // number of pixels per line
                                 HFP     = 10'd16,   // horizontal front porch
                                 HSYN    = 10'd96,   // horizontal sync pulse = 60 to move
                                                     // electron gun back to left
                                 // number of horizontal pixels (i.e., clock cycles)
                                 HMAX    = HBP + HACTIVE + HFP + HSYN, //48+640+16+96=800:
                                 VBP     = 10'd32,   // vertical back porch
                                 VACTIVE = 10'd480,  // number of lines
                                 VFP     = 10'd11,   // vertical front porch
                                 VSYN    = 10'd2,    // vertical sync pulse = 2 to move
                                                     // electron gun back to top
                                 // number of vertical pixels (i.e., clock cycles)
                                 VMAX    = VBP + VACTIVE + VFP  + VSYN) //32+480+11+2=525:

    (input  logic vgaclk, reset,
     output logic hsync, vsync, sync_b, blank_b,
     output logic [9:0] hcnt, vcnt);

     // counters for horizontal and vertical positions
     always @(posedge vgaclk, posedge reset) begin
       if (reset) begin
         hcnt <= 0;
         vcnt <= 0;
       end
       else  begin
         hcnt++;
          if (hcnt == HMAX) begin
           hcnt <= 0;
               vcnt++;
               if (vcnt == VMAX)
                 vcnt <= 0;
          end
       end
     end


     // compute sync signals (active low)
     assign hsync  = ~( (hcnt >= (HACTIVE + HFP)) & (hcnt < (HACTIVE + HFP + HSYN)) );
     assign vsync  = ~( (vcnt >= (VACTIVE + VFP)) & (vcnt < (VACTIVE + VFP + VSYN)) );
     assign sync_b = 1'b0;   // this should be 0 for newer monitors
                             // for older monitors, use: assign sync_b = hsync & vsync;
     // force outputs to black when not writing pixels
     assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule
```

```
module videoGen(input logic [9:0] x, y, output logic [7:0] r, g, b);
  logic pixel, inrect;

  // given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue. Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00, {8{pixel}}};
  assign g      = inrect   ? 8'hFF : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;            // a line read from the ROM

  // initialize ROM with characters from text file
  initial $readmemb("charrom.txt", charrom);

  // index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}];  // subtract 65 because A
                                                // is entry 0
  // reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

HDL Example e9.4 charrom.txt: CONTENTS OF THE CHARACTER ROM

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

King Bluetooth
(So… this is actually Olof Kindgren, but we imagine King Bluetooth looked similar. Photo reprinted with permission.)
The Bluetooth standard is named for King Harald Bluetooth of Denmark, a 10th century monarch who unified the warring Danish tribes. This wireless standard is only partially successful at unifying a host of competing wireless protocols!

Table e9.13 Bluetooth classes

Class	Transmitter Power (mW)	Range (m)
1	100	100
2	2.5	10
3	1	5

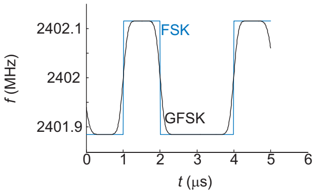


Figure e9.26 FSK and GFSK waveforms

9.4.3 Bluetooth Wireless Communication

Many standards are now available for wireless communication, including Wi-Fi, ZigBee, and Bluetooth. The standards are elaborate and require sophisticated integrated circuits, but a growing assortment of modules abstract away the complexity and give the user a simple interface for wireless communication. One of these modules is the BlueSMiRF, which is an easy-to-use Bluetooth wireless interface that can be used instead of a serial cable.

Bluetooth is a wireless standard initially developed by Ericsson in 1994 for low-power, moderate-speed communication over distances of 5 to 100 meters, depending on the transmitter power level. It is commonly used to connect an earpiece to a cellphone or a keyboard to a computer. Unlike infrared communication links, it does not require a direct line of sight between devices.

Bluetooth operates in the 2.4 GHz unlicensed industrial-scientific-medical (ISM) band. It defines 79 radio channels spaced at 1 MHz intervals starting at 2402 MHz. It hops between these channels in a pseudo-random pattern to avoid consistent interference with other devices, such as wireless routers operating in the same band. As given in Table e9.13, Bluetooth transmitters are classified at one of three power levels, which dictate the range and power consumption. In the basic rate mode, it operates at 1 Mbit/sec using Gaussian frequency shift keying (FSK). In ordinary FSK, each bit is conveyed by transmitting a frequency of $f_c \pm f_d$, where f_c is the center frequency of the channel and f_d is an offset of at least 115 kHz. The abrupt transition in frequencies between bits consumes extra bandwidth. In Gaussian FSK, the change in frequency is smoothed to make better use of the spectrum. Figure e9.26 shows the frequencies being transmitted for a sequence of 0's and 1's on a 2402 MHz channel using FSK and GFSK.

A BlueSMiRF Silver module, shown in Figure e9.27(a), contains a Class 2 Bluetooth radio, modem, and interface circuitry on a small card with a serial interface. It communicates with another Bluetooth device, such as a laptop with built-in Bluetooth, or a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a RED-V and a PC similar to the link from Figure e9.13 but without the cable. The wireless link is compatible with the same software as is the wired link.

Figure e9.28 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the RED-V and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

The BlueSMiRF defaults to 115.2 kbaud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic

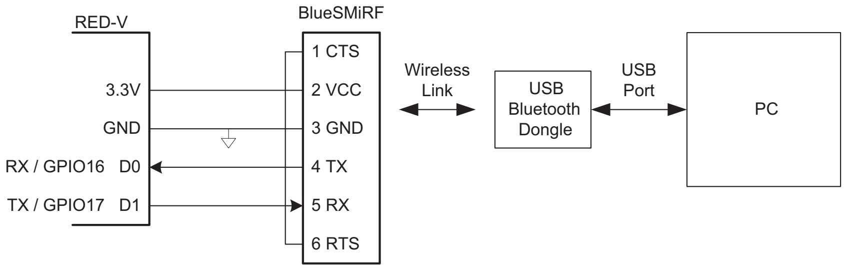


Figure e9.28 **BlueSMiRF RED-V to PC link**

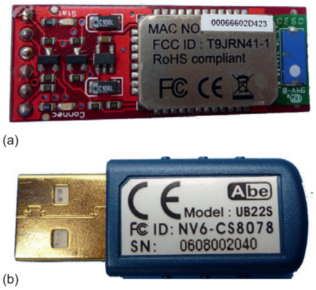


Figure e9.27 **(a) BlueSMiRF module and (b) USB dongle**

levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

To use the interface, plug a USB Bluetooth dongle into a PC. Power up the RED-V and BlueSMiRF. The red STAT light will flash on the BlueSMiRF, indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with the BlueSMiRF. The default passkey for the BlueSMiRF is 1234. Take note of which COM port is assigned to the dongle. Then communication can proceed just as it would over a serial cable. Note that the dongle typically operates at 9600 baud and that PuTTY must be configured accordingly.

9.4.4 Motor Control

Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, typically on the order of 1 A. Thus, a microcontroller's GPIO cannot drive them directly and a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. Motors also require a *shaft encoder* if the user wants to know the current position of the motor. *Servo motors* accept a pulse-width modulated signal to specify their position over a limited range of angles. They are very easy to interface but are not as powerful and are not suited to continuous rotation. *Stepper motors* accept a sequence of pulses, each of which rotates the motor by a fixed angle, called a *step*. They are more expensive and still need an H-bridge to drive the high current, but the position can be precisely controlled.

Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.

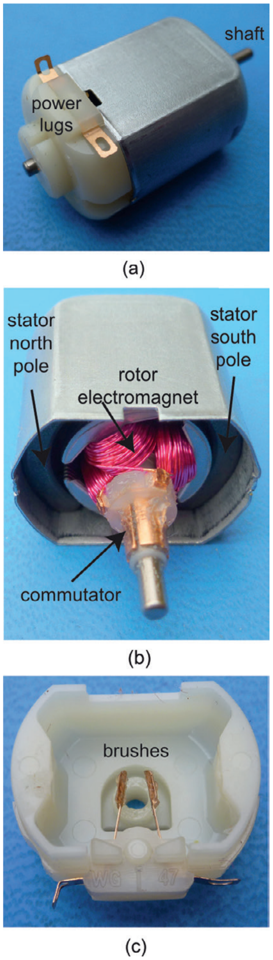


Figure e9.29 **DC motor**

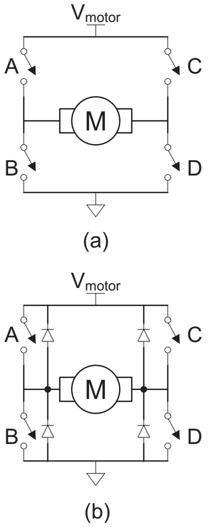


Figure e9.30 **H-bridge**

DC Motors

Figure e9.29 shows the structure of a brushed DC motor. The motor is a two-terminal device. It contains permanent stationary magnets called the *stator* and a rotating electromagnet called the *rotor* or *armature* connected to the shaft. The front end of the rotor connects to a split metal ring called a *commutator*. Metal brushes attached to the power lugs (input terminals) rub against the commutator, providing current to the rotor's electromagnet. This induces a magnetic field in the rotor that causes the rotor to spin to become aligned with the stator field. Once the rotor has spun part way around and approaches alignment with the stator, the brushes touch the opposite sides of the commutator, reversing the current flow and magnetic field and causing it to continue spinning indefinitely.

DC motors tend to spin at thousands of rotations per minute (RPM) at very low torque. Most systems add a gear train to reduce the speed to a more reasonable level and increase the torque. Look for a gear train designed to mate with your motor. Pittman manufactures a wide range of high-quality DC motors and accessories, while inexpensive toy motors are popular among hobbyists.

A DC motor requires substantial current and voltage to deliver significant power to a load. The current should also be reversible so the motor can spin in both directions. Most microcontrollers cannot produce enough current to drive a DC motor directly. Instead, they use an H-bridge, which conceptually contains four electrically controlled switches, as shown in Figure e9.30(a). It is called an H-bridge because the configuration of switches mimics the letter H. If switches A and D are closed, current flows from left to right through the motor and it spins in one direction. If B and C are closed, current flows from right to left through the motor and it spins in the other direction. If A and C or B and D are closed, the voltage across the motor is forced to 0, causing the motor to actively brake. If none of the switches are closed, the motor will coast to a stop. The switches in an H-bridge are *power transistors*, that is, they can carry high currents of one or more Amps. The H-bridge also contains some digital logic to conveniently control the switches. The microcontroller supplies a low-current digital input to control the H-bridge high-current output.

When the motor current changes abruptly, the inductance of the motor's electromagnet will induce a large voltage spike that could damage the power transistors. Therefore, many H-bridges also have protection diodes in parallel with the switches, as shown in Figure e9.30(b). If the inductive kick drives either terminal of the motor above V_{motor} or below ground, the diodes will turn ON and clamp the voltage at a safe level. H-bridges can dissipate large amounts of power, so a heat sink may be necessary to keep them cool.

Example e9.12 AUTONOMOUS VEHICLE

Design a system in which a RED-V board controls two drive motors for a robot car. Write a library of functions to initialize the motor driver and to make the car drive forward and back, turn left or right, and stop. Use PWM to vary the voltage output and, thus, control the speed of the motors.

Solution Figure e9.31 shows a pair of DC motors controlled by a RED-V via a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5 to 36 V motor supply V_{CC2}; it has $V_{IH} = 2\text{V}$ and, hence, is compatible with the 3.3 V I/O from the RED-V. It can deliver up to 1 A of current to each of two motors. V_{motor} should come from a separate battery pack; the 5 V output of the RED-V cannot supply enough current to drive most motors and the RED-V could be damaged.

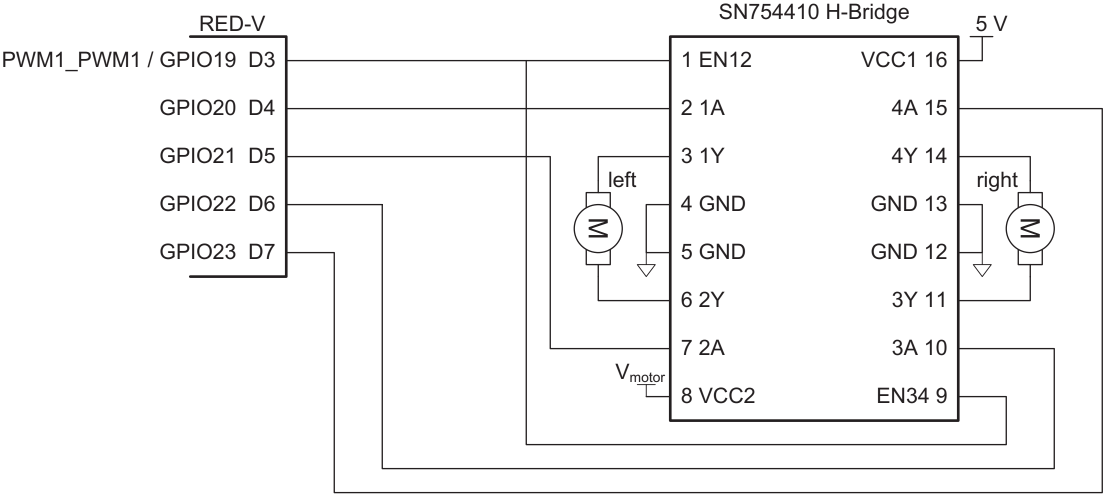


Figure e9.31 Motor control with dual H-bridge

Table e9.14 describes how the inputs to each H-bridge control a motor. The microcontroller drives the enable signals with a PWM signal to control the speed of the motors. It drives the four other pins to control the direction of each motor.

The PWM is configured to work at about 5 kHz with a duty cycle ranging from 0% to 100%. Any PWM frequency far higher than the motor's bandwidth will give the effect of smooth movement. Note that the relationship between duty cycle and motor speed is nonlinear and that below some duty cycle, the motor will not move at all.

Code Example e9.18 shows how to use PWM control with the dual H-bridge configuration shown in Figure e9.31 to drive two DC motors.

Table e9.14 H-Bridge control

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

Code Example e9.18 DC MOTOR DRIVER

```
#include "EasyREDVIO.h"

// Motor Constants
#define EN D3
#define MOTOR_1A D4
#define MOTOR_2A D5
#define MOTOR_3A D6
#define MOTOR_4A D7

void setMotorLeft(int dir) {  // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir){  // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1); // both motors drive forward
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0); // both motors drive backward
}

void left(void) {
    setMotorLeft(0); setMotorRight(1); // left back, right forward
}

void right(void) {
    setMotorLeft(1); setMotorRight(0); // right back, left forward
}

void halt(void) {  // turn both motors off
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                     // ensure motors are not spinning
    pwmInit(EN, 1, 255);        // turn on PWM
    analogWrite(200);           // default to partial power
}

int  main(void) {
    initMotors();
    while(1)
    {
        forward();
        delay(5000);
        backward();
        delay(5000);
        left();
        delay(5000);
        right();
        delay(5000);
        halt();
    }
}
```

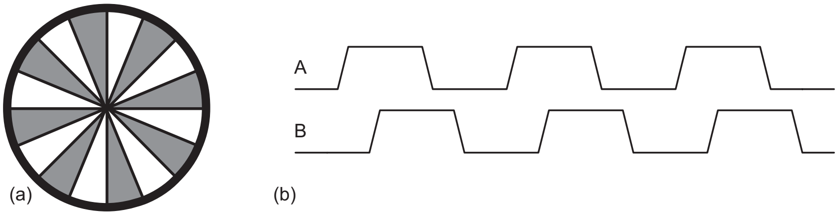


Figure e9.32 Shaft encoder (a) disk, (b) quadrature outputs

In the previous example, there is no way to measure the position of each motor. Two motors are unlikely to be exactly matched, so one is likely to turn slightly faster than the other, causing the robot to veer off course. To solve this problem, some systems add shaft encoders. Figure e9.32(a) shows a simple shaft encoder consisting of a disk with slots attached to the motor shaft. An LED is placed on one side and a light sensor is placed on the other side. The shaft encoder produces a pulse every time the gap rotates past the LED. A microcontroller can count these pulses to measure the total angle that the shaft has turned. By using two LED/sensor pairs spaced half a slot width apart, an improved shaft encoder can produce quadrature outputs shown in Figure e9.32(b) that indicate the direction the shaft is turning, as well as the angle by which it has turned. Sometimes shaft encoders add another hole to indicate when the shaft is at an index position.

Servo Motor

A servo motor is a DC motor integrated with a gear train, a shaft encoder, and some control logic so that it is easier to use. It has a limited rotation, typically 180°. Figure e9.33 shows a servo with the lid removed to reveal the gears. A servo motor has a 3-pin interface with power (typically 5 V), ground, and a control input. The control input is typically a 50 Hz pulse-width modulated signal. The servo's control logic drives the shaft to a position determined by the duty cycle of the control input. The servo's shaft encoder is typically a rotary potentiometer that produces a voltage dependent on the shaft position.

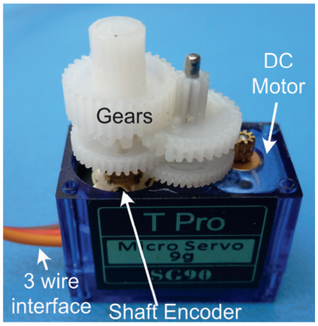


Figure e9.33 SG90 servo motor

In a typical servo motor with 180 degrees of rotation, a pulse width of 1 ms drives the shaft to 0°, 1.5 ms to 90°, and 2 ms to 180°. For example, Figure e9.34 shows a control signal with a 1.5 ms pulse width. Driving the servo outside its range may cause it to hit mechanical stops and be damaged. The servo's power comes from the power pin rather than the control pin, so the control can connect directly to a microcontroller without an H-bridge. Servo motors are commonly used in remote-control model airplanes and small robots because they are small, light, and convenient. Finding a motor with an adequate datasheet can

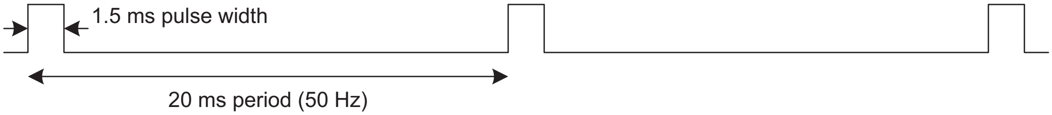


Figure e9.34 Servo control waveform

be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

Example e9.13 SERVO MOTOR

Design a system in which a RED-V drives a servo motor to a desired angle.

Solution Figure e9.35 shows a diagram of the connection to an SG90 servo motor, including the colors of the wires on the servo cable. The servo operates off of a 4.0 to 7.2 V power supply. It can draw as much as 0.5 A if it must deliver a large amount of force but may run directly off the RED-V power supply if the load is light. A single wire carries the PWM signal, which can be provided at 5 or 3.3 V logic levels. The code configures the PWM generation and computes the appropriate duty cycle for the desired angle. It cycles through positioning the servo at 0°, 90°, and 180°.

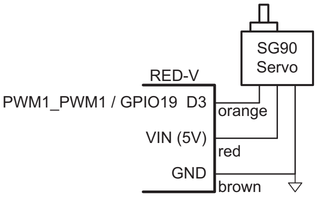


Figure e9.35 Servo motor control

Code Example e9.19 SERVO MOTOR DRIVER

```
#include "EasyREDVIO.h"

void genPulseMicroseconds(uint16_t pulse_len_us) {
    PWM1->pwmcmp1.pwmcmp = pulse_len_us;
}

void setServo(float angle) {
    volatile uint16_t pulse_len_us = (uint16_t) (1000 + (angle / 180) * 1000);
    genPulseMicroseconds(pulse_len_us);
}

int main(void) {
    uint32_t scale = 4; // Set scale to get 16e6/2^4 = 1 MHz count speed for 1 us accuracy
    float freq = 50.0;
    volatile uint32_t pwm_period_count = (uint32_t) (1/freq * 1e6); // Period for PWM in
                                                                     // microseconds

    pwmInit(D3, scale, pwm_period_count);
    while(1) {
        setServo(0.0);
        delay(1000);
        setServo(90.0);
        delay(1000);
        setServo(180.0);
        delay(1000);
    }
}
```

It is also possible to convert an ordinary servo into a continuous rotation servo by carefully disassembling it, removing the mechanical stop, and replacing the potentiometer with a fixed voltage divider. Many websites show detailed directions for particular servos. The PWM will then control the velocity rather than position, with 1.5 ms indicating stop, 2 ms indicating full speed forward, and 1 ms indicating full speed backward. A continuous rotation servo may be more convenient and less expensive than a simple DC motor combined with an H-bridge and gear train.

Stepper Motor

A stepper motor advances in discrete steps as pulses are applied to alternate inputs. The step size is usually a few degrees, allowing precise positioning and continuous rotation. Small stepper motors generally come with two sets of coils called *phases* wired in *bipolar* or *unipolar* fashion. Bipolar motors are more powerful and less expensive for a given size but require an H-bridge driver, while unipolar motors can be driven with transistors acting as switches. This section focuses on the more efficient bipolar stepper motor.

Figure e9.36(a) shows a simplified two-phase bipolar motor with a 90-degree step size. The rotor is a permanent magnet with one north and one south pole. The stator is an electromagnet with two pairs of coils comprising the two phases. Two-phase bipolar motors thus have four terminals. Figure e9.36(b) shows a symbol for the stepper motor modeling the two coils as inductors. Practical motors add gearing to reduce the output step size and increase torque.

Figure e9.37 shows three common drive sequences for a two-phase bipolar motor. Figure e9.37(a) illustrates *wave drive*, in which the coils are energized in the sequence AB–CD–BA–DC. Note that BA means that the winding AB is energized with current flowing in the opposite direction; this is the origin of the name *bipolar*. The rotor turns by 90 degrees at each step. Figure e9.37(b) illustrates *two-phase-on drive*, following the pattern (AB, CD)–(BA, CD)–(BA, DC)–(AB, DC). (AB, CD) indicates that both coils AB and CD are energized simultaneously. The rotor again turns by 90 degrees at each step, but aligns itself halfway between the two pole positions. This gives the highest torque operation because both coils are delivering power at once. Figure e9.37(c) illustrates *half-step drive*, following the pattern (AB, CD)–CD–(BA, CD)–BA–(BA, DC)–DC–(AB, DC)–AB. The rotor turns by 45 degrees at each half-step. The rate at which the pattern advances determines the speed of the motor. To reverse the motor direction, the same drive sequences are applied in the opposite order.

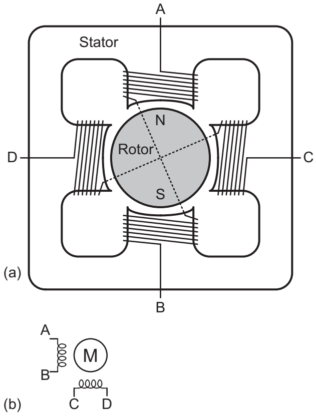


Figure e9.36 Two-phase bipolar motor: (a) simplified diagram, (b) symbol

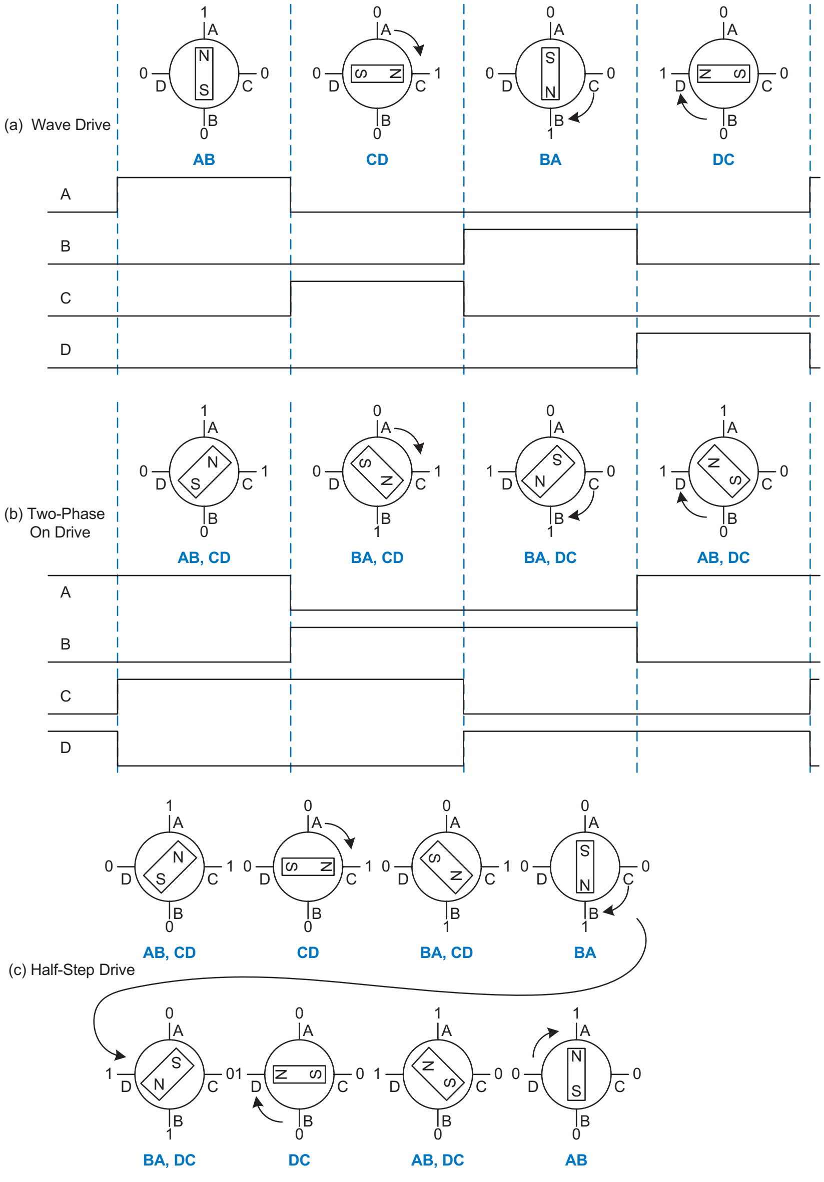


Figure e9.37 Bipolar motor drive

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure e9.39 shows an AIRPAX LB82773-M1 bipolar stepper motor with a 7.5-degree step size. The motor operates off 5 V and draws 0.8 A through each coil.

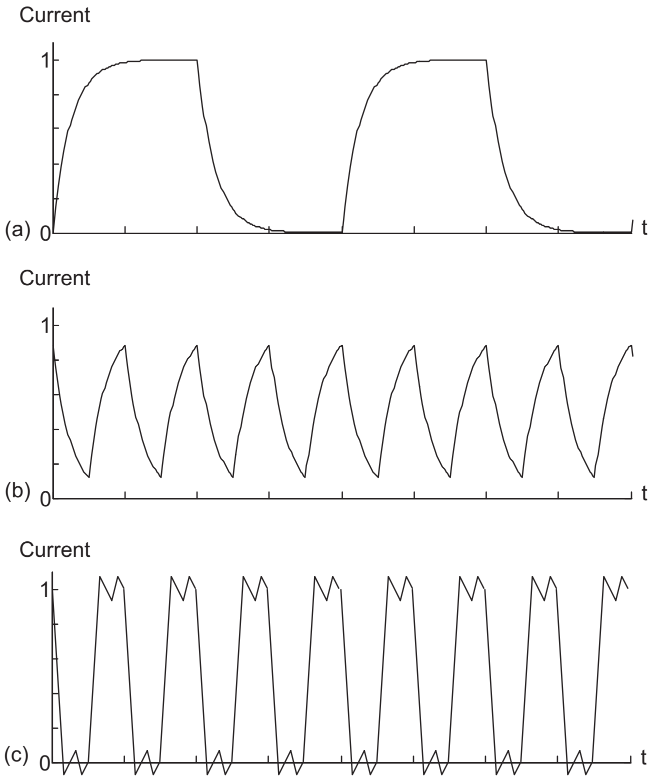


Figure e9.38 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

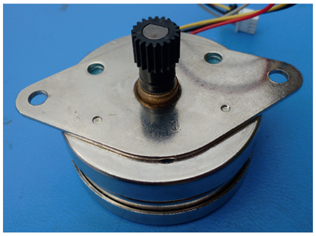


Figure e9.39 **AIRPAX LB82773-M1 bipolar stepper motor**

The torque in the motor is proportional to the coil current. This current is determined by the voltage applied and by the inductance L and resistance R of the coil. The simplest mode of operation is called *direct voltage drive* or *L/R drive*, in which the voltage V is directly applied to the coil. The current ramps up to $I = V/R$ with a time constant set by L/R, as shown in Figure e9.38(a). This works well for slow speed operation. However, at higher speed, the current doesn't have enough time to ramp up to the full level, as shown in Figure e9.38(b), and the torque drops off.

A more efficient way to drive a stepper motor is by pulse-width modulating a higher voltage. The high voltage causes the current to ramp up to full current more rapidly; then, it is turned off (during the off portion of the PWM duty cycle) to avoid overloading the motor. The voltage is then modulated or *chopped* to maintain the current near the desired level. This is called *chopper constant current drive* and is shown in Figure e9.38(c). The controller uses a small resistor in series with the motor to sense the current being applied by measuring the voltage drop and applies an enable signal to the H-bridge to turn off the drive when the current reaches the desired level. In principle, a microcontroller could generate the right waveforms, but it is easier to use a stepper motor controller. The L297 controller from ST Microelectronics is a convenient choice, especially when coupled with the L298 dual H-bridge with current sensing pins and a 2 A peak power capability. Unfortunately, the L298 is not available in a DIP package, so it is harder to breadboard. ST's application notes AN460 and AN470 are valuable references for stepper motor designers.

Example e9.14 BIPOLAR STEPPER MOTOR DIRECT WAVE DRIVE

Design a system to drive an AIRPAX bipolar stepper motor at a specified speed and direction using direct wave drive.

Solution Figure e9.40 shows the bipolar stepper motor driven directly by an H-bridge with the same interface as the DC motor. Note that VCC2 must supply enough voltage and current to meet the motor's demands or the motor may skip steps as the rotation rate increases.

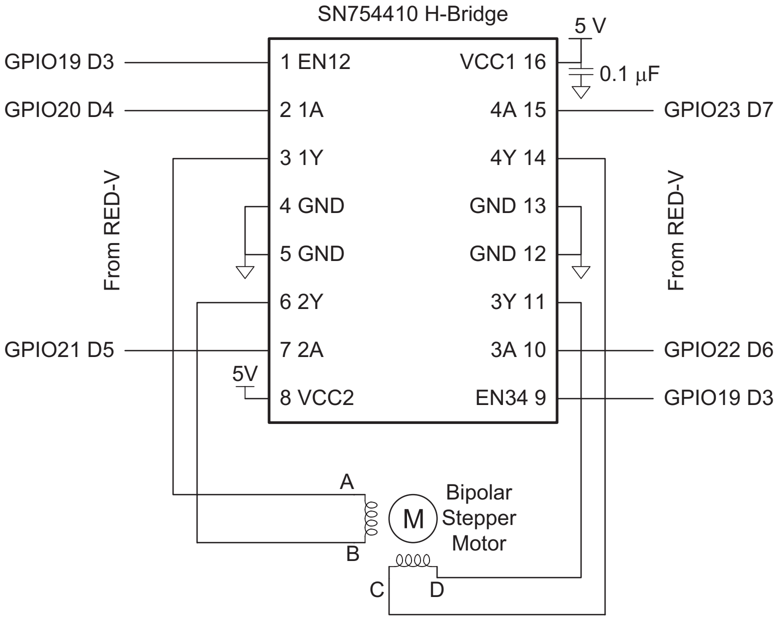


Figure e9.40 **Bipolar stepper motor direct drive with H-bridge**

Code Example e9.20 STEPPER MOTOR DRIVER

```
#include "EasyREDVIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MILLIS_PER_SEC 1000
#define DEG_PER_REV 360

int stepperPins[] = {19, 22, 23, 20, 21};
int curStepState; // Keep track of the current position of stepper motor

void stepperInit(void) {
  pinsMode(stepperPins, 5, OUTPUT);
  curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001}; //{2A, 1A, 4A, 3A, EN}
  int step = 0;

  unsigned int millisPerStep = (SECS_PER_MIN * MILLIS_PER_SEC * STEPSIZE) /
                               (rpm * DEG_PER_REV);

  for (step = 0; step < steps; step++) {
    digitalWrites(stepperPins, 5, sequence[curStepState]);
    if (dir == 0) curStepState = (curStepState + 1) % 4;
    else  curStepState = (curStepState + 3) % 4;
    delay(millisPerStep);
  }
}
int main(void) {
  stepperInit();
  stepperSpin(1, 12000, 120); // Spin 60 revolutions at 120 rpm
}
```

9.5 SUMMARY

Most processors use memory-mapped I/O to communicate with the real world. Microcontrollers offer a range of basic peripherals including general-purpose, serial, and analog I/O and timers.

This chapter has provided many specific examples of I/O using the FE310 RISC-V microcontroller on a SparkFun RED-V RedBoard. Embedded system designers continually encounter new processors and peripherals. The general principle for incorporating simple embedded I/O is to consult the datasheet to identify the peripherals that are available and which pins and memory-mapped I/O registers are involved. Then, it is usually straightforward to write a simple device driver that initializes the peripheral's registers and transmits or receives data.

For more complex standards such as USB, writing a device driver is a highly specialized undertaking best done by an expert with detailed knowledge of the device and the USB protocol stack. Casual designers should select a processor that comes with proven device drivers and example code for the devices of interest.

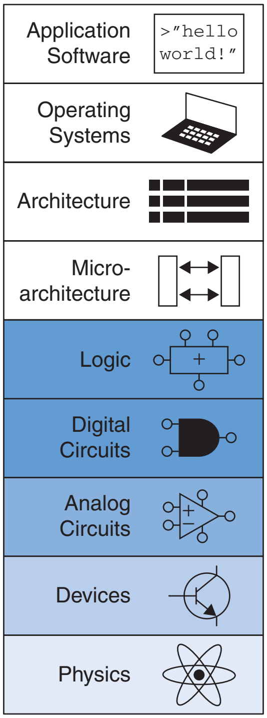
Digital System Implementation

eA

A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book; however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

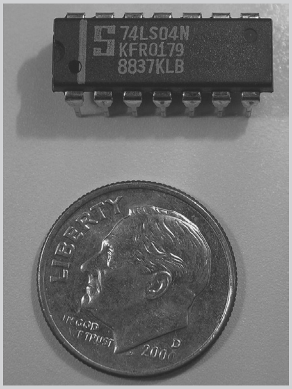
Digital systems are usually built using one or more chips. One strategy is to connect multiple chips containing individual logic gates or larger elements, such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.



A.2 74xx LOGIC

In the 1970's and 1980's, many digital systems were built from simple chips, each containing a handful of logic gates. For example, the 7404 chip contains six NOT gates, the 7408 contains four AND gates, and the 7474 contains two flip-flops. These chips are collectively referred to as *74xx-series* logic. They were sold by many manufacturers, typically for 10 to 25 cents per chip. These chips are now largely obsolete, but they are still handy for simple digital systems or class projects because they are so inexpensive and easy to use. 74xx-series chips are commonly sold in 14-pin *dual inline packages* (*DIPs*).

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00015-5



74LS04 inverter chip in a 14-pin dual inline package. The part number is on the first line. LS indicates the logic family (see Section A.6). The N suffix indicates a DIP package. The large S is the logo of the manufacturer, Signetics. The bottom two lines of gibberish are codes indicating the batch in which the chip was manufactured.

A.2.1 Logic Gates

Figure eA.1 shows the pinout diagrams for a variety of popular 74xx-series chips containing basic logic gates. These are sometimes called *small-scale integration* (*SSI*) chips because they are built from a few transistors. The 14-pin packages typically have a notch at the top or a dot on the top left to indicate orientation. Pins are numbered starting with 1 in the upper left and going counterclockwise around the package. The chips need to receive power ($V_{DD} = 5\,V$) and ground ($GND = 0\,V$) at pins 14 and 7, respectively. The number of logic gates on the chip is determined by the number of pins. Note that pins 3 and 11 of the 7421 chip are not connected (NC) to anything. The 7474 flip-flop has the usual D, CLK, and Q terminals. It also has a complementary output, $\overline{Q}$. Moreover, it receives asynchronous set (also called preset, or PRE) and reset (also called clear, or CLR) signals. These are active low; in other words, the flop sets when $\overline{PRE} = 0$, resets when $\overline{CLR} = 0$, and operates normally when $\overline{PRE} = \overline{CLR} = 1$. Low-asserted signals are often written in text as CLRb or CLRbar.

A.2.2 Other Functions

The 74xx series also includes somewhat more complex logic functions, including those shown in Figures eA.2 and eA.3. These are called *medium-scale integration* (*MSI*) chips. Most use larger packages to accommodate more inputs and outputs. Power and ground are still provided at the upper right and lower left, respectively, of each chip. A general functional description is provided for each chip. See the manufacturer's datasheets for complete descriptions.

A.3 PROGRAMMABLE LOGIC

Programmable logic consists of arrays of circuitry that can be configured to perform specific logic functions. We have already introduced three forms of programmable logic: programmable read-only memories (PROMs), programmable logic arrays (PLAs), and field-programmable gate arrays (FPGAs). This section shows chip implementations for each of these. Configuration of these chips may be performed by blowing on-chip fuses to connect or disconnect circuit elements. This is called *one-time programmable* (*OTP*) logic because, once a fuse is blown, it cannot be restored. Alternatively, the configuration may be stored in a memory that can be reprogrammed at will. Reprogrammable logic is convenient in the laboratory because the same chip can be reused during development.

A.3.1 PROMs

As discussed in Section 5.5.7, PROMs can be used as lookup tables. A 2^N-word × M-bit PROM can be programmed to perform any

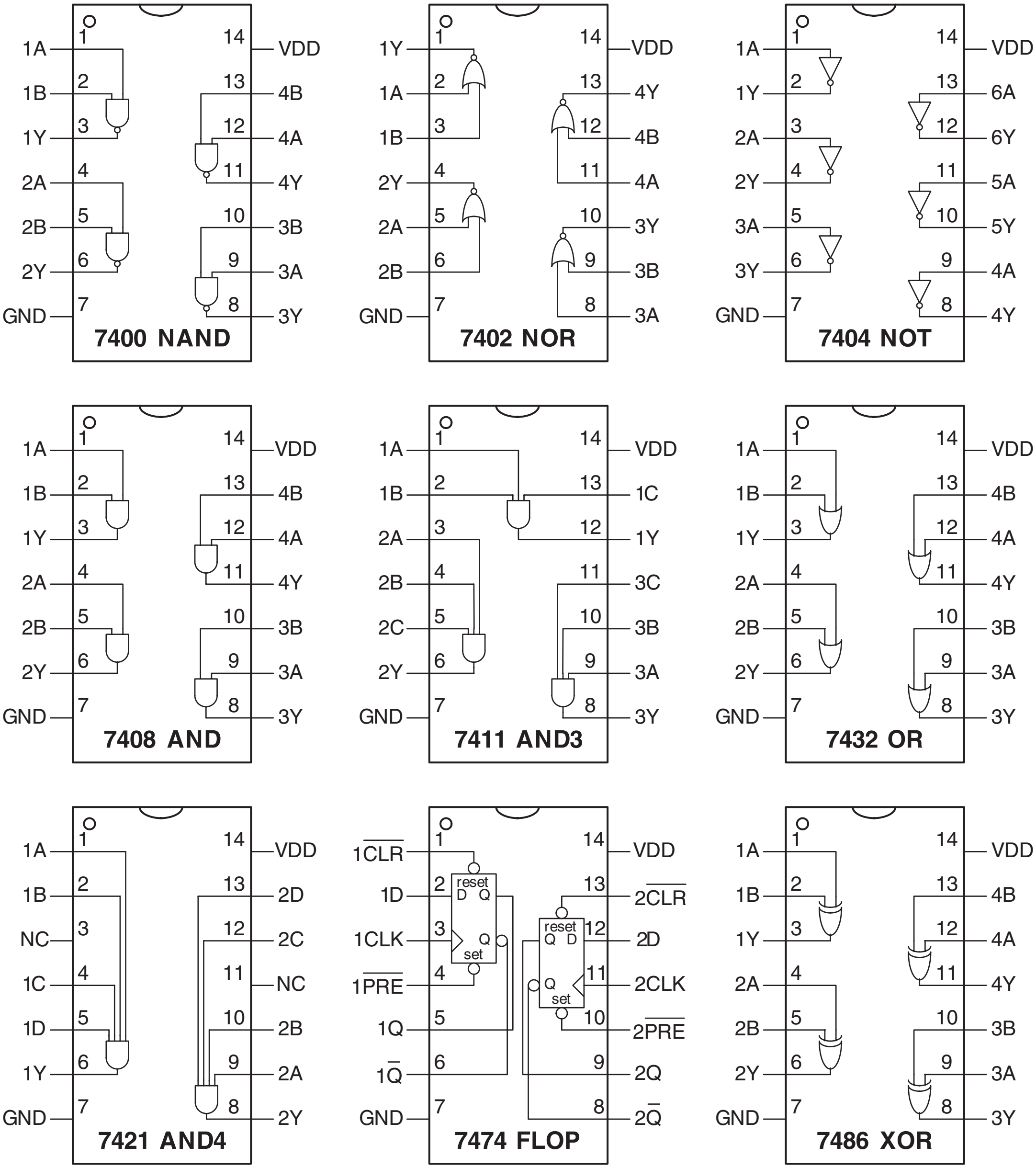
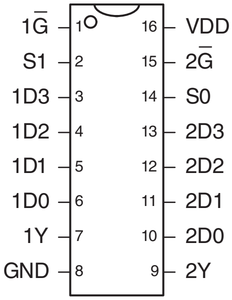


Figure eA.1 Common 74xx-series logic gates

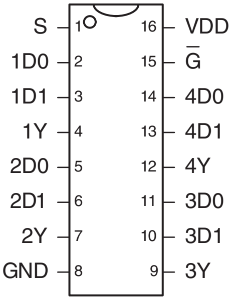


74153 4:1 Mux

Two 4:1 Multiplexers

$D_{3:0}$:	data
$S_{1:0}$:	select
Y:	output
Gb:	enable

```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```

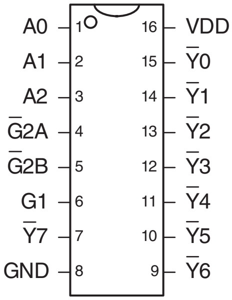


74157 2:1 Mux

Four 2:1 Multiplexers

$D_{1:0}$:	data
S:	select
Y:	output
Gb:	enable

```
always_comb
  if  (Gb) 1Y = 0;
  else     1Y = S ? 1D[1]  : 1D[0];
  if  (Gb) 2Y = 0;
  else     2Y = S ? 2D[1]  : 2D[0];
  if  (Gb) 3Y = 0;
  else     3Y = S ? 3D[1]  : 3D[0];
  if  (Gb) 4Y = 0;
  else     4Y = S ? 4D[1]  : 4D[0];
```

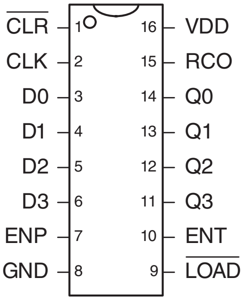


74138 3:8 Decoder

3:8 Decoder

$A_{2:0}$:	address
$Yb_{7:0}$:	output
G1:	active high enable
G2:	active low enables

G1	$\overline{G2A}$	$\overline{G2B}$	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



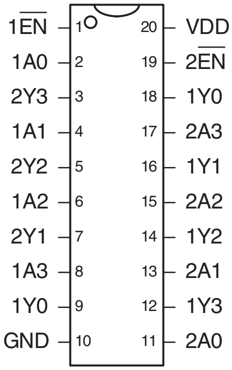
74161/163 Counter

4-bit Counter

CLK:	clock
$Q_{3:0}$:	counter output
$D_{3:0}$:	parallel input
CLRb:	async reset (161)
	sync reset (163)
LOADb:	load Q from D
ENP, ENT:	enables
RCO:	ripple carry out

```
always_ff @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```

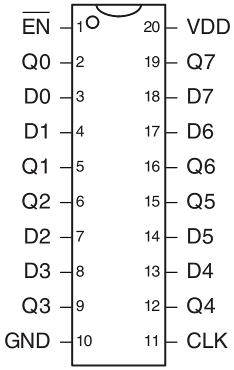


74244 Tristate Buffer

8-bit Tristate Buffer

$A_{3:0}$:	input
$Y_{3:0}$:	output
ENb:	enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```



74377 Register

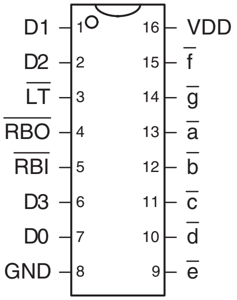
8-bit Enableable Register

CLK:	clock
$D_{7:0}$:	data
$Q_{7:0}$:	output
ENb:	enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Note: SystemVerilog variable names cannot start with numbers, but the names in the example code in Figure eA.2 are chosen to match the manufacturer's datasheet.

Figure eA.2 Medium-scale integration chips



7447 7-Segment Decoder

7-segment Display Decoder

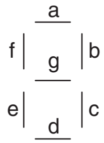
$D_{3:0}$: data

a...f : segments (low = ON)

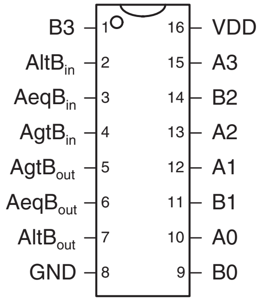
LTb: light test

RBIb: ripple blanking in

RBOb: ripple blanking out



$\overline{RBO}$	$\overline{LT}$	$\overline{RBI}$	D3:0	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	$\overline{g}$
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0



7485 Comparator

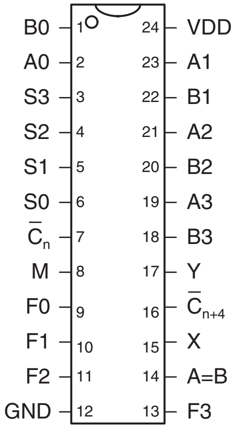
4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data

rel_{in} : input relation

rel_{out} : output relation

```
always_comb
   if  (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```



74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs

$Y_{3:0}$: output

$F_{3:0}$: function select

M : mode select

Cb_n : carry in

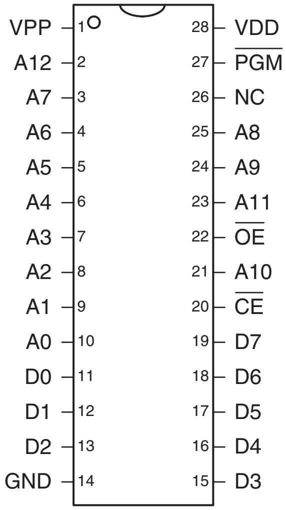
Cb_{nplus4} : carry out

AeqB : equality (in some modes)

X,Y : carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A         :  A              + ~Cbn;
    0001: Y = M ? ~(A | B)   :  A      +  B    + ~Cbn;
    0010: Y = M ? (~A) & B   :  A      + ~B    + ~Cbn;
    0011: Y = M ? 4'b0000    :  4'b1111        + ~Cbn;
    0100: Y = M ? ~(A & B)   :  A      + (A & ~B) + ~Cbn;
    0101: Y = M ? ~B         : (A | B) + (A & ~B) + ~Cbn;
    0110: Y = M ? A ^ B      :  A      - B        -  Cbn;
    0111: Y = M ? A & ~B     : (A & ~B)           -  Cbn;
    1000: Y = M ? ~A + B     :  A      + (A & B)  + ~Cbn;
    1001: Y = M ? ~(A ^ B)   :  A      +  B       + ~Cbn;
    1010: Y = M ? B          : (A | ~B) + (A & B) + ~Cbn;
    1011: Y = M ? A & B      : (A & B)            + ~Cbn;
    1100: Y = M ? 1          :  A      +  A       + ~Cbn;
    1101: Y = M ? A | ~B     : (A | B) +  A       + ~Cbn;
    1110: Y = M ? A | B      : (A | ~B) +  A      + ~Cbn;
    1111: Y = M ? A          :  A                 -  Cbn;
  endcase
```

Figure eA.3 More medium-scale integration (MSI) chips



8 KB EPROM

$A_{12:0}$:	address input
$D_{7:0}$:	data output
CEb :	chip enable
OEb :	output enable
PGMb :	program
VPP :	program voltage
NC :	no connection

```
assign D = (~CEb & ~OEb) ? ROM[A]
                         : 8'bz;
```

Figure eA.4 2764 8 KiB EPROM

combinational function of N inputs and M outputs. Design changes simply involve replacing the contents of the PROM rather than rewiring connections between chips. Lookup tables are useful for small functions but become prohibitively expensive as the number of inputs grows.

For example, the classic 2764 8-KiB (64-kib) erasable PROM (EPROM) is shown in Figure eA.4. The EPROM has 13 address lines to specify one of the 8 Ki words and 8 data lines to read the byte of data at that word. The chip enable and output enable must both be asserted for data to be read. The maximum propagation delay is 200 ps. In normal operation, $\overline{PGM} = 1$ and VPP is not used. The EPROM is usually programmed on a special programmer that sets $\overline{PGM} = 0$, applies 13 V to VPP, and uses a special sequence of inputs to configure the memory.

Modern PROMs are similar in concept but have much larger capacities and more pins. Flash memory is the cheapest type of PROM, selling for about $0.10 per gigabyte in 2021. Prices have historically declined by 30% to 40% per year.

A.3.2 PLAs

As discussed in Section 5.6.1, PLAs contain AND and OR planes to compute any combinational function written in sum-of-products form. The AND and OR planes can be programmed using the same techniques for PROMs. A PLA has two columns for each input and one column for each output. It has one row for each minterm. This organization is more efficient than a PROM for many functions, but the array still grows excessively large for functions with numerous I/Os and minterms.

Many different manufacturers have extended the basic PLA concept to build *programmable logic devices* (*PLDs*) that include registers.

The 22V10 is one of the most popular classic PLDs. It has 12 dedicated input pins and 10 outputs. The outputs can come directly from the PLA or from clocked registers on the chip. The outputs can also be fed back into the PLA. Thus, the 22V10 can directly implement finite state machines (FSMs) with up to 12 inputs, 10 outputs, and 10 bits of state. The 22V10 costs about $1.35 in quantities of 100. PLDs have been rendered mostly obsolete by the rapid improvements in capacity and cost of FPGAs.

A.3.3 FPGAs

As discussed in Section 5.6.2, FPGAs consist of arrays of configurable *logic elements* (*LEs*), also called *configurable logic blocks* (*CLBs*), connected with programmable wires. The LEs contain small lookup tables and flip-flops. FPGAs scale gracefully to extremely large capacities, with thousands of lookup tables. Xilinx and Intel FPGAs (formerly Altera) are two of the leading FPGA manufacturers.

Lookup tables and programmable wires are flexible enough to implement any logic function. However, they are an order of magnitude less efficient in speed and cost (chip area) than hard-wired versions of the same functions. Thus, FPGAs often include specialized blocks, such as memories, multipliers, and even entire microprocessors.

Figure eA.5 shows the design process for a digital system on an FPGA. The design is usually specified with a hardware description language (HDL), although some FPGA tools also support schematics. The design is then simulated. Inputs are applied and compared against expected outputs to *verify* that the logic is correct. Usually, some debugging is required. Next, logic *synthesis* converts the HDL into Boolean functions. Good synthesis tools produce a schematic of the functions. The prudent designer examines these schematics and any warnings produced during synthesis to ensure that the desired logic was produced. Sometimes, sloppy coding leads to circuits that are much larger than intended or to circuits with asynchronous logic. When the synthesis step completes successfully, the FPGA tool *maps* the functions onto the LEs of a specific chip. The *place and route* tool determines which functions go in which lookup tables and how they are wired together. Wire delay increases with length, so critical circuits should be placed close together. If the design is too big to fit on the chip, it must be reengineered. *Timing analysis* compares the timing constraints (e.g., an intended clock speed of 100 MHz) against the actual circuit delays and reports any errors. If the logic is too slow, it may have to be redesigned or pipelined differently. When the design is correct, a file is generated specifying the contents of all the LEs and the programming of all the wires on the FPGA. Many FPGAs store this *configuration* information in static RAM that must be reloaded each time the FPGA is turned on. The FPGA can

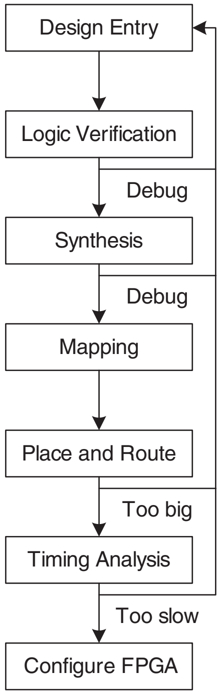


Figure eA.5 FPGA design flow

download this information from a computer in the laboratory or can read it from a nonvolatile ROM when power is first applied.

Example eA.1 FPGA TIMING ANALYSIS

Alyssa P. Hacker is using an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. Her design is implemented as an FSM, and she is using a Cyclone IV FPGA. According to the datasheet, the FPGA has the timing characteristics shown in Table eA.1.

Alyssa would like her FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which her FSM could possibly run?

Solution At 100 MHz, the cycle time, T_c, is 10 ns. Alyssa uses Equation eA.1 to figure out the minimum combinational propagation delay, t_{pd}, at this cycle time:

$$t_{pd} \leq 10\,\text{ns} - (0.199\,\text{ns} + 0.076\text{ ns}) = 9.725\,\text{ns} \tag{eA.1}$$

With a combined LE and wire delay of 381 ps + 246 ps = 627 ps, Alyssa's FSM can use at most 15 consecutive LEs (9.725/0.627) to implement the next-state logic.

The fastest speed at which an FSM will run on this Cyclone IV FPGA is when it is using a single LE for the next-state logic. The minimum cycle time is

$$T_c \geq 381\,\text{ps} + 199\,\text{ps} + 76\,\text{ps} = 656\,\text{ps} \tag{eA.2}$$

Therefore, the maximum frequency is 1.5 GHz.

Altera advertised the Cyclone IV FPGA with 14,400 LEs for $25 in 2021. In large quantities, medium-sized FPGAs typically cost several

Table eA.1 Cyclone IV timing

Name	Value (ps)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

dollars. The largest FPGAs cost hundreds or even thousands of dollars. The cost has declined at approximately 30% per year, so FPGAs are becoming extremely popular.

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cellphone chips are common examples of ASICs. The ASIC designer places transistors to form logic gates and wires the gates together. Because the ASIC is hardwired for a specific function, it is typically several times faster than an FPGA and occupies an order of magnitude less chip area (and, hence, cost) than an FPGA with the same function. However, the *masks* specifying where transistors and wires are located on the chip cost hundreds of thousands of dollars to produce. The fabrication process usually requires 6 to 12 weeks to manufacture, package, and test the ASICs. If errors are discovered after the ASIC is manufactured, the designer must correct the problem, generate new masks, and wait for another batch of chips to be fabricated. Hence, ASICs are suitable only for products that will be produced in large quantities and whose function is well defined in advance.

Figure eA.6 shows the ASIC design process, which is similar to the FPGA design process of Figure eA.5. Logic verification is especially important because correction of errors after the masks are produced is expensive. Synthesis produces a *netlist* consisting of logic gates and connections between the gates. The gates in this netlist are placed, and the wires are routed between gates. When the design is satisfactory, masks are generated and used to fabricate the ASIC. A single speck of dust can ruin an ASIC, so the chips must be tested after fabrication. The fraction of manufactured chips that work is called the *yield*. It is typically 50% to 90% depending on the size of the chip and the maturity of the manufacturing process. Finally, the working chips are placed in packages, as will be discussed in Section A.7.

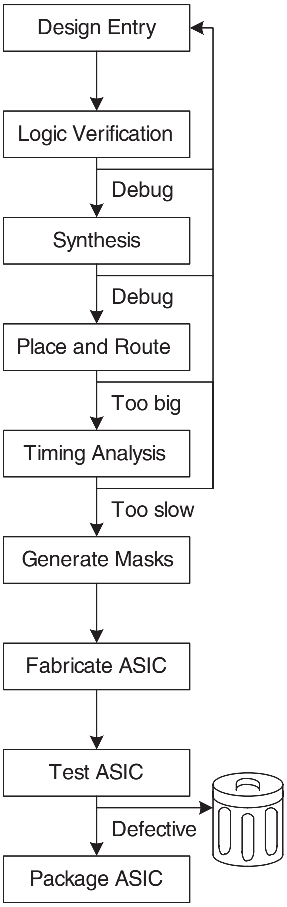


Figure eA.6 **ASIC design flow**

A.5 DATASHEETS

Integrated circuit manufacturers publish *datasheets* that describe the functions and performance of their chips. It is essential to read and understand the datasheets. One of the leading sources of errors in digital systems comes from misunderstanding the operation of a chip.

Datasheets are usually available from the manufacturer's website. If you cannot locate the datasheet for a part and do not have clear documentation from another source, don't use the part. Some of the entries in the datasheet may be cryptic. Often, the manufacturer publishes databooks containing datasheets for many related parts. The beginning of the data book has additional explanatory information. This information can usually be found on the Web with a careful search.

This section dissects the Texas Instruments (TI) datasheet for a 74HC04 inverter chip. The datasheet is relatively simple but illustrates many of the major elements. TI still manufacturers a wide variety of 74xx-series chips. In the past, many other companies built these chips as well, but the market is consolidating as the sales decline.

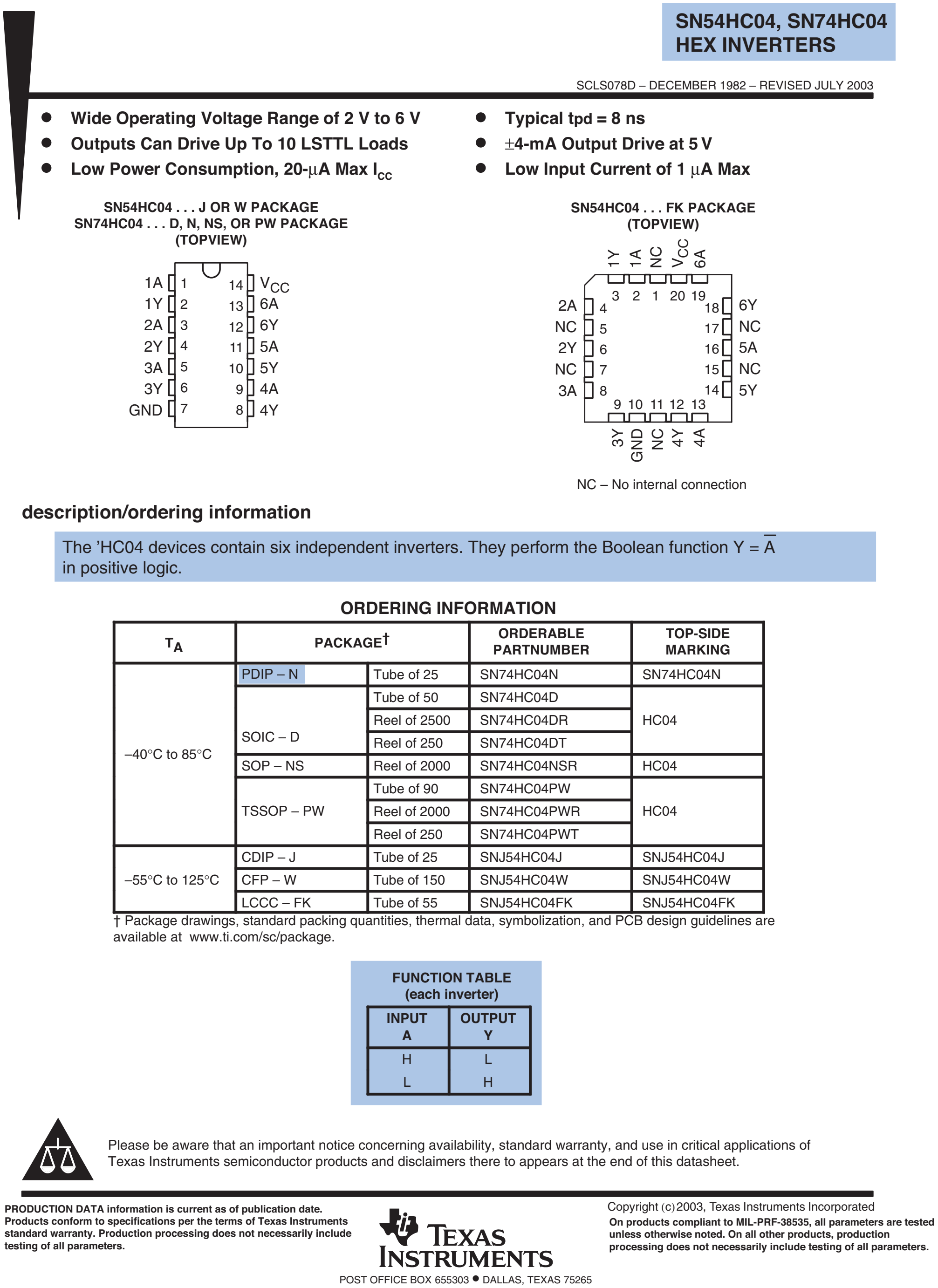
Figure eA.7 shows the first page of the datasheet. Some of the key sections are highlighted in blue. The title is SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS means that the chip contains six inverters. SN indicates that TI is the manufacturer. Other manufacturer codes include MC for Motorola and DM for National Semiconductor. You can generally ignore these codes because all of the manufacturers build compatible 74xx-series logic. HC is the logic family (high-speed CMOS). The logic family determines the speed and power consumption of the chip, but not the function. For example, the 7404, 74HC04, and 74LS04 chips all contain six inverters, but they differ in performance and cost. Other logic families are discussed in Section A.6. The 74xx chips operate across the commercial or industrial temperature range (0°C to 70°C or −40°C to 85°C, respectively), whereas the 54xx chips operate across the military temperature range (−55° to 125°C) and sell for a higher price but are otherwise compatible.

The 7404 is available in many different packages; it is important to order the one you intended when you make a purchase. The packages are distinguished by a suffix on the part number. N indicates a *plastic dual inline package* (*PDIP*), which fits in a breadboard or can be soldered in through-holes in a printed circuit board. Other packages are discussed in Section A.7.

The function table shows that each gate inverts its input. If A is HIGH (H), Y is LOW (L) and vice versa. The table is trivial in this case but is more interesting for more complex chips.

Figure eA.8 shows the second page of the datasheet. The logic diagram indicates that the chip contains inverters. The *absolute maximum* section indicates conditions beyond which the chip could be destroyed. In particular, the power supply voltage (V_{CC}, also called V_{DD} in this book) should not exceed 7V. The continuous output current should not exceed 25 mA. The *thermal resistance* or impedance, θ_{JA}, is used to calculate the temperature rise caused by the chip's dissipating power. If the *ambient* temperature in the vicinity of the chip is T_A and the chip dissipates P_{chip}, then the temperature on the chip itself at its *junction* with the package is

$$T_J = T_A + P_{\text{chip}}\,\theta_{JA} \qquad \text{(eA.3)}$$



SN54HC04, SN74HC04 HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- Wide Operating Voltage Range of 2 V to 6 V
- Outputs Can Drive Up To 10 LSTTL Loads
- Low Power Consumption, 20-µA Max I_{CC}
- Typical tpd = 8 ns
- ±4-mA Output Drive at 5 V
- Low Input Current of 1 µA Max

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)

1A 1
1Y 2
2A 3
2Y 4
3A 5
3Y 6
GND 7
14 VCC
13 6A
12 6Y
11 5A
10 5Y
9 4A
8 4Y

SN54HC04 . . . FK PACKAGE
(TOPVIEW)

1Y
1A
NC
VCC
6A
3 2 1 20 19
2A 4
NC 5
2Y 6
NC 7
3A 8
18 6Y
17 NC
16 5A
15 NC
14 5Y
9 10 11 12 13
3Y
GND
NC
4Y
4A

NC – No internal connection

description/ordering information

The ’HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

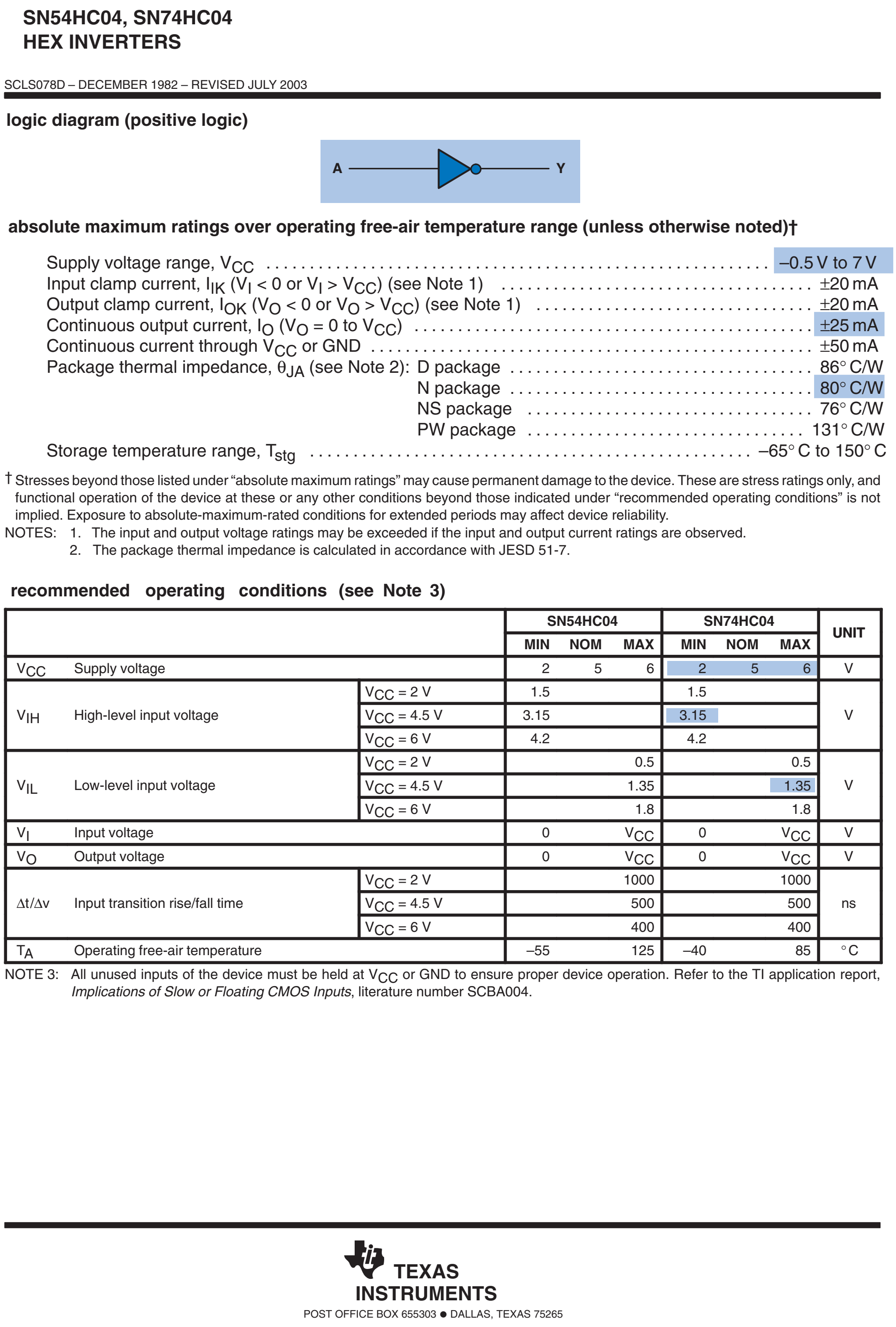
Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this datasheet.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

On products compliant to MIL-PRF-38535, all parameters are tested unless otherwise noted. On all other products, production processing does not necessarily include testing of all parameters.

Figure eA.7 7404 datasheet page 1



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC}	–0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1)	±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1)	±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC})	±25 mA
Continuous current through V_{CC} or GND	±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package	86° C/W
N package	80° C/W
NS package	76° C/W
PW package	131° C/W
Storage temperature range, T_{stg}	–65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
∆t/∆v	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 ● DALLAS, TEXAS 75265

Figure eA.8 7404 datasheet page 2

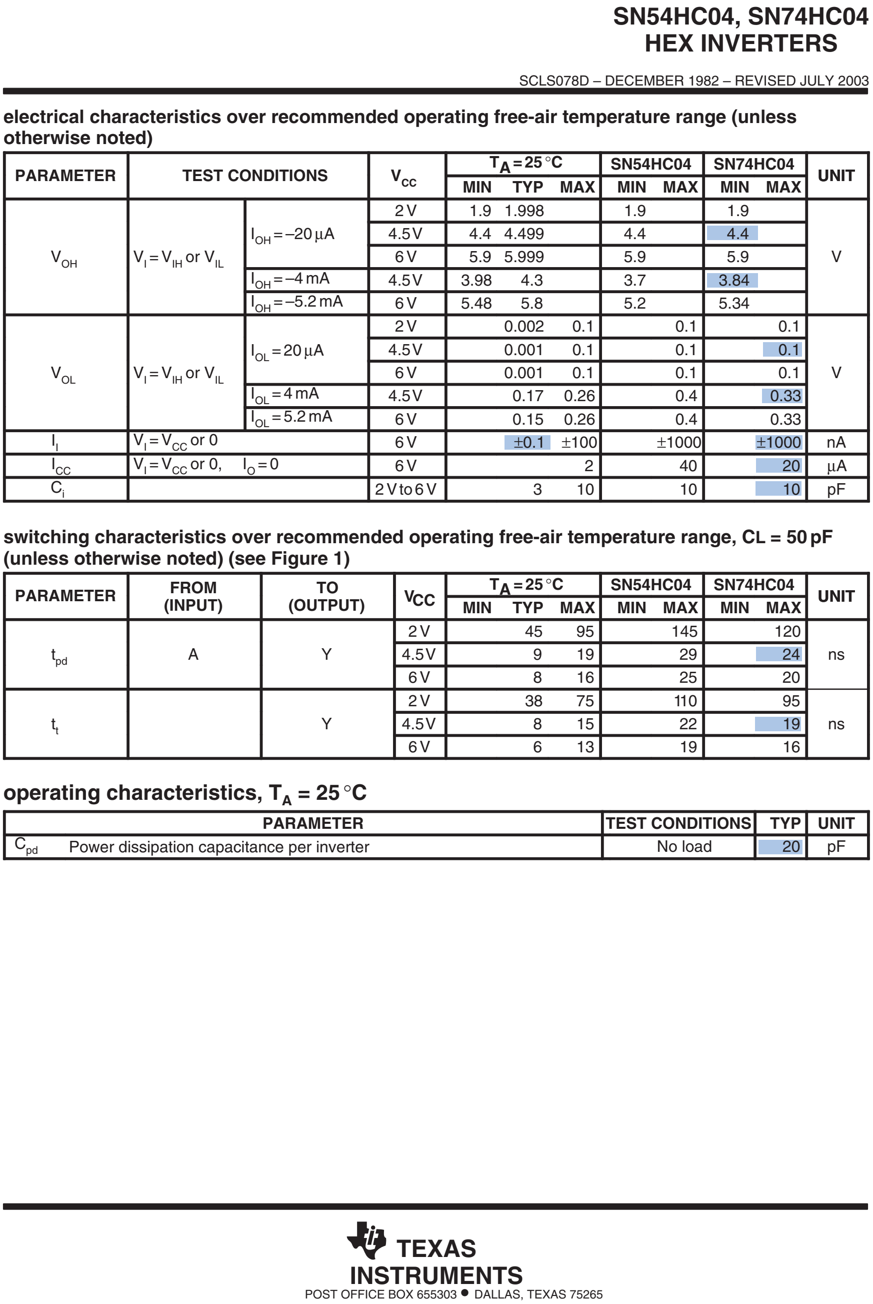
For example, if a 7404 chip in a plastic DIP package is operating in a hot box at 50°C and consumes 20 mW, the junction temperature will climb to 50°C + 0.02 W × 80°C/W = 51.6°C. Internal power dissipation is seldom important for 74xx-series chips, but it becomes important for modern chips that dissipate tens of watts or more.

The *recommended operating conditions* define the environment in which the chip should be used. Within these conditions, the chip should meet specifications. These conditions are more stringent than the absolute maximums. For example, the power supply voltage should be between 2 and 6 V. The input logic levels for the HC logic family depend on V_{DD}. Use the 4.5 V entries when V_{DD} = 5 V to allow for a 10% droop in the power supply caused by noise in the system.

Figure eA.9 shows the third page of the datasheet. The *electrical characteristics* describe how the device performs when used within the recommended operating conditions if the inputs are held constant. For example, if $\mathrm{V_{CC}}$ = 5 V (and droops to 4.5 V) and the output current I_{OH}/I_{OL} does not exceed 20 μ*A*, V_{OH} = 4.4 V and V_{OL} = 0.1 V in the worst case. If the output current increases, the output voltages become less ideal because the transistors on the chip struggle to provide the current. The HC logic family uses CMOS transistors that draw very little current. The current into each input is guaranteed to be less than 1000 nA and is typically only 0.1 nA at room temperature. The *quiescent* power supply current (I_{DD}) drawn while the chip is idle is less than 20 μ*A*. Each input has less than 10 pF of capacitance.

The *switching characteristics* define how the device performs when used within the recommended operating conditions if the inputs change. The *propagation delay*, t_{pd}, is measured from when the input passes through 0.5 V_{CC} to when the output passes through 0.5 V_{CC}. If V_{CC} is nominally 5 V and the chip drives a capacitance of less than 50 pF, the propagation delay will not exceed 24 ns (and typically will be much faster). Recall that each input may present 10 pF, so the chip cannot drive more than five identical chips at full speed. Indeed, stray capacitance from the wires connecting chips cuts further into the useful load. The *transition time*, also called the *rise/fall time*, is measured as the output transitions between 0.1 V_{CC} and 0.9 V_{CC}.

Recall from Section 1.8 that chips consume both *static* and *dynamic power*. Static power is low for HC circuits. At 85°C, the maximum quiescent supply current is 20 μ*A*. At 5 V, this gives a static power consumption of 0.1 mW. The dynamic power depends on the capacitance being driven and the switching frequency. The 7404 has an internal power dissipation capacitance of 20 pF per inverter. If all six inverters on the 7404 switch at 10 MHz and drive external loads of 25 pF, then the dynamic power given by Equation 1.4 is $\frac{1}{2}(6)(20\,\mathrm{pF} + 25\,\mathrm{pF})(5^2)(10\,\mathrm{MHz}) = 33.75\,\mathrm{mW}$ and the maximum total power is 33.85 mW.



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	$T_A = 25\,°C$			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	$I_{OH} = -20\,\mu A$	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		$I_{OH} = -4\,mA$	4.5 V	3.98	4.3		3.7		3.84		
		$I_{OH} = -5.2\,mA$	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	$I_{OL} = 20\,\mu A$	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		$I_{OL} = 4\,mA$	4.5 V		0.17	0.26		0.4		0.33	
		$I_{OL} = 5.2\,mA$	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, $I_O = 0$		6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	$T_A = 25\,°C$			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, $T_A = 25\,°C$

PARAMETER	TEST CONDITIONS	TYP	UNIT
C_{pd} Power dissipation capacitance per inverter	No load	20	pF

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 ● DALLAS, TEXAS 75265

Figure eA.9 7404 datasheet page 3

A.6 LOGIC FAMILIES

The 74xx-series logic chips have been manufactured using many different technologies, called *logic families*, that offer different speed, power, and logic level trade-offs. Other chips are usually designed to be compatible with some of these logic families. The original chips, such as the 7404, were built using bipolar transistors in a technology called *transistor-transistor logic* (*TTL*). Newer technologies add one or more letters after the 74 to indicate the logic family, such as 74LS04, 74HC04, or 74AHCT04. Table eA.2 summarizes the most common 5-V logic families.

Advances in bipolar circuits and process technology led to the *Schottky* (*S*) and *Low-Power Schottky* (*LS*) families. Both are faster than TTL. Schottky draws more power, whereas Low-Power Schottky draws less. *Advanced Schottky* (*AS*) and *Advanced Low-Power Schottky* (*ALS*) have improved speed and power compared with S and LS. *Fast* (*F*) logic is faster and draws less power than AS. All of these families provide more current for LOW outputs than for HIGH outputs and, hence, have

Table eA.2 Typical specifications for 5-V logic families

	Bipolar/TTL						CMOS		CMOS/TTL Compatible	
Characteristic	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH} (V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL} (V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH} (V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL} (V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH} (mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL} (mA)	16	20	8	20	8	20	4	8	4	8
I_{IL} (mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH} (mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD} (mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{Pd} (pF)	n/a						20	12	20	14
cost* (US $)	obsolete	0.63	0.25	0.53	0.32	0.22	0.12	0.12	0.12	0.12

*Per unit in quantities of 1000 for the 7408 from Texas Instruments in 2012.

asymmetric logic levels. They conform to the “TTL” logic levels: V_{IH} = 2 V, V_{IL} = 0.8 V, V_{OH} > 2.4 V, and V_{OL} < 0.5 V.

As CMOS circuits matured in the 1980’s and 1990’s, they became popular because they draw very little power supply or input current. The *High-Speed CMOS* (*HC*) and *Advanced High-Speed CMOS* (*AHC*) families draw almost no static power. They also deliver the same current for HIGH and LOW outputs. They conform to the “CMOS” logic levels: V_{IH} = 3.15 V, V_{IL} = 1.35 V, V_{OH} > 3.8 V, and V_{OL} < 0.44 V. Unfortunately, these levels are incompatible with TTL circuits because a TTL HIGH output of 2.4 V may not be recognized as a legal CMOS HIGH input. This motivates the use of *High-Speed TTL-compatible CMOS* (*HCT*) and *Advanced High-Speed TTL-compatible CMOS* (*AHCT*), which accept TTL input logic levels and generate valid CMOS output logic levels. These families are slightly slower than their pure CMOS counterparts. All CMOS chips are sensitive to *electrostatic discharge* (*ESD*) caused by static electricity. Ground yourself by touching a large metal object before handling CMOS chips, lest you zap them.

The 74xx-series logic is inexpensive. The newer logic families are often cheaper than the obsolete ones. The LS family is widely available and robust and is a popular choice for laboratory or hobby projects that have no special performance requirements.

The 5-V standard collapsed in the mid-1990’s, when transistors became too small to withstand the voltage. Moreover, lower voltage offers lower power consumption. Now 3.3, 2.5, 1.8, 1.2, and even lower voltages are commonly used. The plethora of voltages raises challenges in communicating between chips with different power supplies. Table eA.3 lists some of the low-voltage logic families. Not all 74xx parts are available in all of these logic families.

All of the low-voltage logic families use CMOS transistors, the workhorse of modern integrated circuits. They operate over a wide range of V_{DD}, but the speed degrades at lower voltage. *Low-Voltage CMOS* (*LVC*) logic and *Advanced Low-Voltage CMOS* (*ALVC*) logic are commonly used at 3.3, 2.5, or 1.8 V. LVC withstands inputs up to 5.5 V, so it can receive inputs from 5-V CMOS or TTL circuits. *Advanced Ultra-Low-Voltage CMOS* (*AUC*) is commonly used at 2.5, 1.8, or 1.2 V and is exceptionally fast. Both ALVC and AUC withstand inputs up to 3.6 V, so they can receive inputs from 3.3 V circuits.

FPGAs often offer separate voltage supplies for the internal logic, called the *core*, and for the input/output (I/O) pins. As FPGAs have advanced, the core voltage has dropped from 5 to 3.3, 2.5, 1.8, and 1.2 V to save power and avoid damaging the very small transistors. FPGAs have configurable I/Os that can operate at many different voltages to be compatible with the rest of the system.

Table eA.3 Typical specifications for low-voltage logic families

	LVC			ALVC			AUC		
V_{dd} (V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd} (ns)	4.1	6.9	9.8	2.8	3	?*	1.8	2.3	3.4
V_{IH} (V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL} (V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH} (V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL} (V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O (mA)	24	8	4	24	12	12	9	8	3
I_I (mA)		0.02			0.005			0.005	
I_{DD} (mA)		0.01			0.01			0.01	
C_{pd} (pF)	10	9.8	7	27.5	23	?*	17	14	14
cost (US $)		0.17			0.20			not available	

*Delay and capacitance not available at the time of writing.

A.7 SWITCHES AND LIGHT-EMITTING DIODES

Digital circuits need inputs and outputs. The most basic way to provide inputs is to wire them directly to power or ground, but a more interesting way is with switches that the user can control. The most basic way to check outputs is with a voltmeter, but a more interesting way is with light-emitting diodes (LEDs) that glow if the output is TRUE. Switches and LEDs are analog components used in this digital application. This section describes how they operate.

A.7.1 Switches

Figure eA.10 shows symbols for single-pole single-throw (SPST) and single-pole double-throw (SPDT) switches. Single-pole indicates that the switch has a single output, and single-throw means that the switch can connect to one terminal. Double-throw indicates that the output can connect to either of two terminals.

An SPST switch or button is either open or closed by default, which blocks or allows current to flow, respectively. To produce a digital logic level, the switch is normally placed in series with a resistor R, as shown in Figure eA.11(a). When the switch is open, the resistor pulls Y down

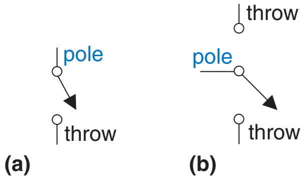


Figure eA.10 (a) SPST and (b) SPDT switches

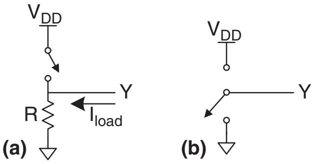


Figure eA.11 Digital inputs from switches: (a) SPST and resistor, (b) SPDT and no resistor

toward ground (logic 0). If the circuitry attached to pin Y presents a small load current I_{load}, the actual voltage on node Y will be $V_Y = I_{\text{load}}R$. When the switch is closed, the top of the resistor connects to V_{DD} and, thus, pulls Y up to V_{DD} (logic 1). A current of V_{DD}/R flows through the resistor, causing a static power consumption of $P = V_{DD}^2/R$. The resistor should be chosen according to the Goldilocks principle: not too big, not too small, but just right. If the resistor is too large, the voltage at Y, V_Y, might not fall low enough when the switch is open (and the circuitry connected to Y draws some current). If the resistor is too small, the circuit dissipates excessive and wasteful power (or might even melt down). As a practical matter, this often means the resistor is selected in the 1- to 10-kΩ range such that the power is small and leakage causes a negligible disturbance to the output voltage. However, if extremely low-power operation is needed, a larger resistor would be appropriate and the leakage current drawn by the load (connected to Y) must be considered more carefully so that the output voltage remains a valid low logic level.

An alternative to the SPST switch and resistor is a single-pole, double-throw switch (SPDT). Figure eA.11(b) shows the SPDT switch with an output that connects to either V_{DD} or GND to produce a strong high or low value without requiring a resistor and without consuming static power (assuming that Y is connected to the input of a CMOS gate). However, SPDT switches are often more expensive.

A.7.2 LEDs

An LED can be approximated as OFF when the voltage across the diode is less than some voltage V_D and ON when the voltage is greater than V_D. V_D varies from about 1.7 to 2.3 V depending on the color of the diode and the ambient temperature, but 2.0 V is a reasonable average for rough calculations. When the diode is off, it draws negligible current. When it is ON, it draws as much current, I_D, as it can, and the brightness is proportional to the current. However, if the current is too great, the diode or the device sourcing the current will burn out. Hence, diodes are normally used with current-limiting resistors, as shown in Figure eA.12. Ordinary small diodes typically are visible in indoor lighting when they carry more than 1 mA and glow nicely at 5 to 10 mA. Ultrabright LEDs might draw 100 mA or more.

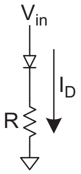


Figure eA.12 LED with current-limiting resistor

Figure eA.12 shows an LED being driven by a voltage V_{in} through a resistor R. The order of the resistor and LED does not matter because the same current flows through both in series. In a digital application, V_{in} typically comes from the output of a logic gate and may be in the range of V_{OH} to V_{DD}. The voltage across the resistor is $V_{\text{in}} - V_D$ (where V_D is the voltage drop across the diode) and it is also $R \times I_D$ because the current flowing through the diode also flows through the resistor. Equating these two, we find that $I_D = (V_{\text{in}} - V_D)/R$. If R is too large, not enough current will flow and the diode will be too dim. If R is too small, the diode will burn out or draw too much current from the component driving V_{in}. Hence, resistors in the low hundreds of ohms are common in digital circuits driving LEDs. Check the output current I_O specification on a datasheet to see how much current the digital output can deliver. For example, if a logic gate operates at 3.3 V and has a maximum output current of 5 mA under normal conditions, then pick $R = (3.3 - 2)\ \text{V} / 0.005\ \text{A} = 260\ \Omega$ to obtain the brightest light the driver can offer.

Remember that the absolute maximum specifications are those beyond which the component may take permanent damage, not those at which the component works correctly. Use the recommended operating condition data instead.

A.8 PACKAGING AND ASSEMBLY

Integrated circuits are typically placed in *packages* made of plastic or ceramic. The packages serve a number of functions, including connecting the tiny metal I/O pads of the chip to larger pins in the package for ease of connection, protecting the chip from physical damage, and spreading the heat generated by the chip over a larger area to help with cooling. The packages are placed on a breadboard or printed circuit board (PCB) and wired together to assemble the system.

A.8.1 Packages

Figure eA.13 shows a variety of integrated circuit packages. Packages can be generally categorized as *through-hole* or *surface mount* (*SMT*). Through-hole packages, as their name implies, have pins that can be inserted through holes in a PCB or into a socket. *Dual inline packages* (*DIPs*) have two rows of pins with 0.1-inch spacing between pins. *Pin grid arrays* (*PGAs*) support more pins in a smaller package by placing the pins under the package. SMT packages are soldered directly to the

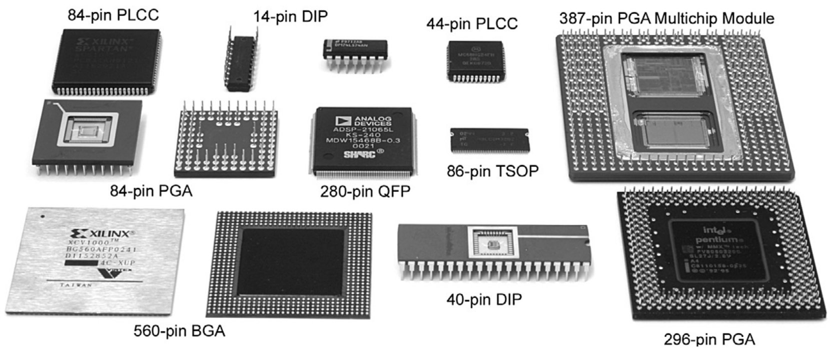


Figure eA.13 Integrated circuit packages surface of a PCB without using holes. Pins on SMT parts are called *leads*. The *thin small outline package* (*TSOP*) has two rows of closely spaced leads (typically, 0.02-inch spacing). *Plastic leaded chip carriers* (*PLCCs*) have J-shaped leads on all four sides, with 0.05-inch spacing. They can be soldered directly to a board or placed in special sockets. *Quad flat packs* (*QFPs*) accommodate a large number of pins, using closely spaced legs on all four sides. *Ball grid arrays* (*BGAs*) eliminate the legs altogether. Instead, they have hundreds of tiny solder balls on the underside of the package. They are carefully placed over matching pads on a PCB, then heated so that the solder melts and joins the package to the underlying board. Dual and quad flat packs with no leads (*DFNs* and *QFNs*) have pins on two or four sides of the chip, respectively, but the leads are under the package instead of on the outside of the package, which can conserve space on a PCB but also requires more tricky soldering.

A.8.2 Breadboards

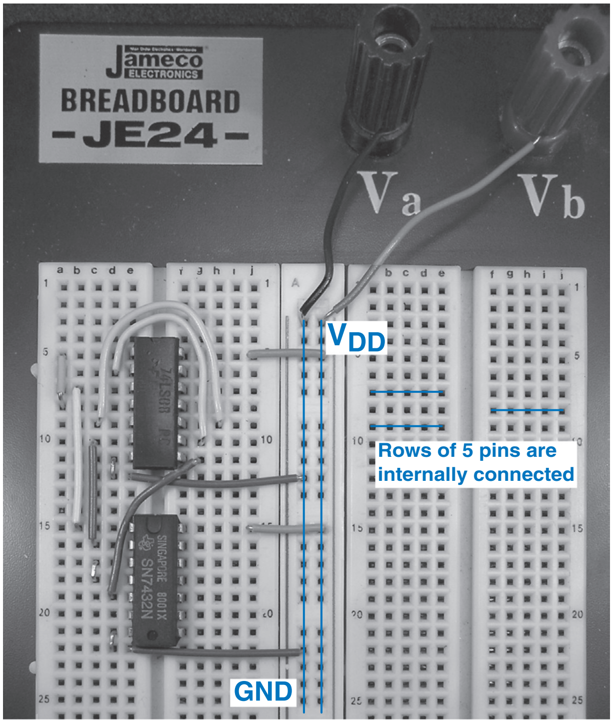
DIPs are easy to use for prototyping, because they can be placed in a *breadboard*. A breadboard is a plastic board containing rows of sockets, as shown in Figure eA.14. All five holes in a row are connected. Each pin of the package is placed in a hole in a separate row. Wires can be placed in adjacent holes in the same row to make connections to the pin. Breadboards often provide separate columns of connected holes running the height of the board to distribute power and ground.

Figure eA.14 shows a breadboard containing a majority gate built with a 74LS08 AND chip and a 74LS32 OR chip. The schematic of the circuit is shown in Figure eA.15. Each gate in the schematic is labeled with the chip (08 or 32) and the pin numbers of the inputs and outputs (see Figure eA.1). Observe that the same connections are made on the breadboard. The inputs are connected to pins 1, 2, and 5 of the 08 chip, and the output is measured at pin 6 of the 32 chip. Power and ground are connected to pins 14 and 7, respectively, of each chip, from the vertical power and ground columns that are attached to the banana plug receptacles, Vb and Va. Labeling the schematic in this way and checking off connections as they are made is a good way to reduce the number of mistakes made during breadboarding.

Unfortunately, it is easy to accidentally plug a wire in the wrong hole or have a wire fall out, so breadboarding requires a great deal of care (and usually some debugging in the laboratory). Breadboards are suited only to prototyping, not production.

A.8.3 Printed Circuit Boards

Instead of breadboarding, chip packages may be soldered to a *printed circuit board* (*PCB*). The PCB is formed of alternating layers of



Rows of 5 pins are internally connected

Figure eA.14 Majority circuit on breadboard

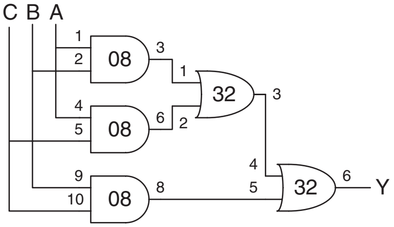


Figure eA.15 Majority gate schematic with chips and pins identified

conducting copper and insulating epoxy. The copper is etched to form wires called *traces*. Holes called *vias* are drilled through the board and plated with metal to connect between layers. PCBs are usually designed with *computer-aided design* (*CAD*) tools. You can etch and drill your own simple boards in the laboratory, or you can send the board design to a specialized factory for inexpensive mass production. Factories have turnaround times of days (or weeks, for cheap mass production runs) and typically charge a few hundred dollars in setup fees and a few dollars per board for moderately complex boards built in large quantities.

PCB traces are normally made of copper because of its low resistance. The traces are embedded in an insulating material, usually a green, fire-resistant plastic called FR4. A PCB also typically has copper power and ground layers, called *planes*, between signal layers. Figure eA.16 shows a cross-section of a PCB. The signal layers are on the top and bottom, and the power and ground planes are embedded in the center of the board. The power and ground planes have low resistance, so they distribute stable power to components on the board. They also make the capacitance and inductance of the traces uniform and predictable.

Figure eA.17 shows a PCB for a 1970's vintage Apple II+ computer. At the top is a 6502 microprocessor. Beneath are six 16-kib ROM chips

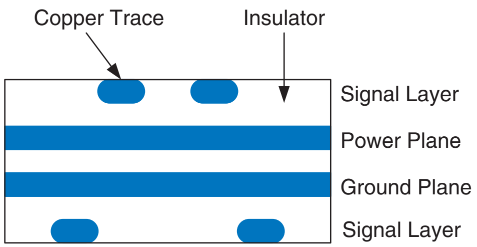


Figure eA.16 Printed circuit board cross-section

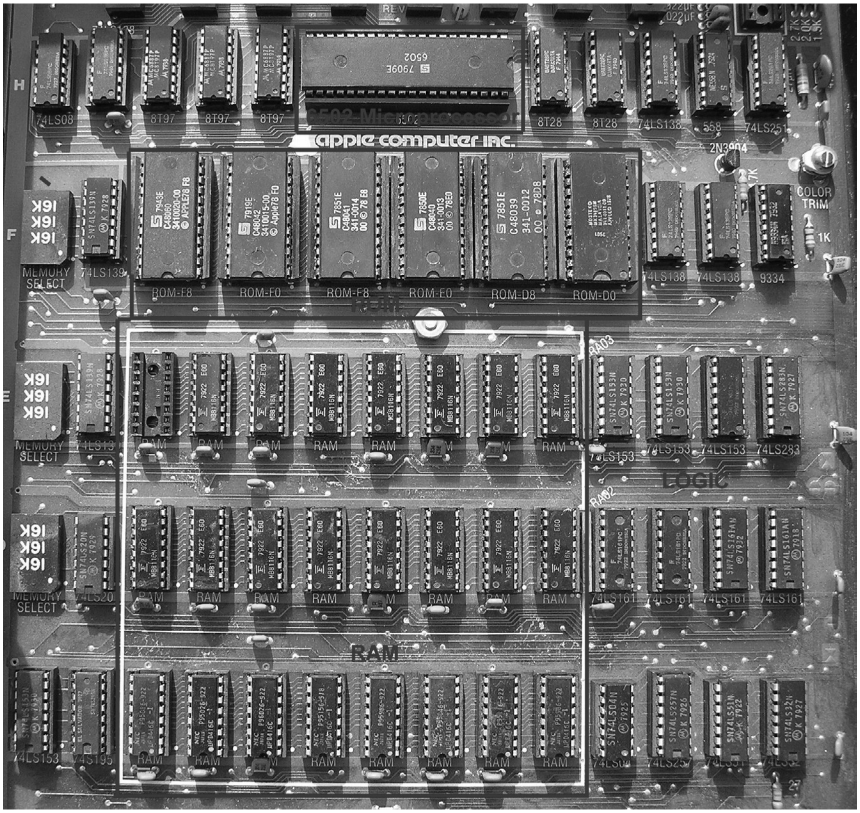


Figure eA.17 Apple II+ circuit board forming 12 KiB of ROM containing the operating system. Three rows of eight 16-kib DRAM chips provide 48 KiB of RAM. On the right are several rows of 74xx-series logic for memory address decoding and other functions. The lines between chips are traces that wire the chips together. The dots at the ends of some of the traces are vias filled with metal.

A.8.4 Putting It All Together

Most modern chips with large numbers of inputs and outputs use SMT packages, especially QFPs and BGAs. These packages require a PCB rather than a breadboard. Working with BGAs is especially challenging because they require specialized assembly equipment. Moreover, the balls cannot be probed with a voltmeter or oscilloscope during debugging in the laboratory because they are hidden under the package.

In summary, the designer needs to consider packaging early on to determine whether a breadboard can be used during prototyping and whether BGA parts will be required. Professional engineers rarely use breadboards when they are confident of connecting chips correctly without experimentation.

A.9 TRANSMISSION LINES

We have assumed so far that wires are *equipotential* connections that have a single voltage along their entire length. Signals actually propagate along wires at the speed of light in the form of electromagnetic waves. If the wires are short enough or the signals change slowly, the equipotential assumption is good enough. When the wire is long or the signal is very fast, the *transmission time* along the wire becomes important to accurately determine the circuit delay. We must model such wires as *transmission lines*, in which a wave of voltage and current propagates at the speed of light. When the wave reaches the end of the line, it may reflect back along the line. The reflection may cause noise and odd behaviors unless steps are taken to limit it. Hence, the digital designer must consider transmission line behavior to accurately account for the delay and noise effects in long wires.

Electromagnetic waves travel at the speed of light in a given medium, which is fast but not instantaneous. The speed of light, ν, depends on the *permittivity*, ε, and *permeability*, μ, of the medium[1]:

$$\nu = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}.$$

[1] The capacitance, C, and inductance, L, of a wire are related to the permittivity and permeability of the physical medium in which the wire is located.

The speed of light in free space is $v = c = 3 \times 10^8$ m/s. Signals in a PCB travel at about half this speed because the FR4 insulator has four times the permittivity of air. Thus, PCB signals travel at about 1.5×10^8 m/s, or 15 cm/ns. The time delay for a signal to travel along a transmission line of length l is

$$t_d = \frac{l}{v}. \tag{eA.4}$$

The *characteristic impedance* of a transmission line, Z_0 (pronounced "Z-naught"), is the ratio of voltage to current in a wave traveling along the line: $Z_0 = V/I$. It is *not* the resistance of the wire (a good transmission line in a digital system typically has negligible resistance). Z_0 depends on the inductance and capacitance of the line (see the derivation in Section A.9.7) and typically has a value of 50 to 75 Ω.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{eA.5}$$

Figure eA.18 shows the symbol for a transmission line. The symbol resembles a *coaxial cable* with an inner signal conductor and an outer grounded conductor like that used in television cable wiring.

The key to understanding the behavior of transmission lines is to visualize the wave of voltage propagating along the line at the speed of light. When the wave reaches the end of the line, it may be absorbed or reflected, depending on the termination or load at the end. Reflections travel back along the line, adding to the voltage already on the line. Terminations are classified as matched, open, short, or mismatched. The following sections explore how a wave propagates along the line and what happens to the wave when it reaches the termination.

A.9.1 Matched Termination

Figure eA.19 shows a transmission line of length l with a *matched termination*, which means that the load impedance, Z_L, is equal to the characteristic impedance, Z_0. The transmission line has a characteristic impedance of 50 Ω. One end of the line is connected to a voltage source through a switch that closes at time $t = 0$. The other end is connected

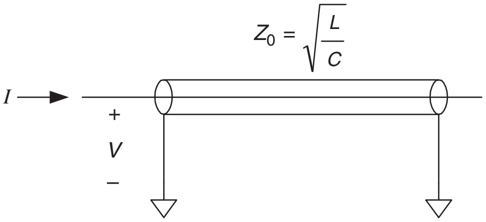


Figure eA.18 Transmission line symbol

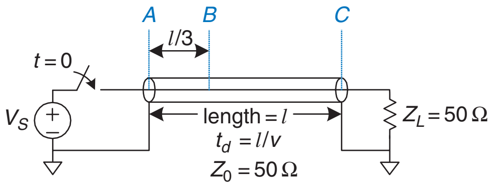


Figure eA.19 Transmission line with matched termination

to the 50 Ω matched load. This section analyzes the voltages and currents at points A, B, and C—at the beginning of the line, one-third of the length along the line, and at the end of the line, respectively.

Figure eA.20 shows the voltages at points A, B, and C over time. Initially, there is no voltage or current flowing in the transmission line because the switch is open. At time t = 0, the switch closes and the voltage source launches a wave with voltage $V = V_S$ along the line. This is called the *incident wave*. Because the characteristic impedance is Z_0, the wave has current $I = V_S/Z_0$. The voltage reaches the beginning of the line (point A) immediately, as shown in Figure eA.20(a). The wave propagates along the line at the speed of light. At time $t_d/3$, the wave reaches point B. The voltage at this point abruptly rises from 0 to V_S, as shown in Figure eA.20(b). At time t_d, the incident wave reaches point C at the end of the line, and the voltage rises there as well. All of the current, I, flows into the resistor, Z_L, producing a voltage across the resistor of $V_R = Z_L I = Z_L (V_S/Z_0) = V_S$ because $Z_L = Z_0$. This voltage is consistent with the wave flowing along the transmission line. Thus, the wave is *absorbed* by the load impedance and the transmission line reaches its *steady state*.

In steady state, the transmission line behaves like an ideal equipotential wire because it is, after all, just a wire. The voltage at all points along the line must be identical. Figure eA.21 shows the steady-state equivalent model of the circuit in Figure eA.19. The voltage is V_S everywhere along the wire.

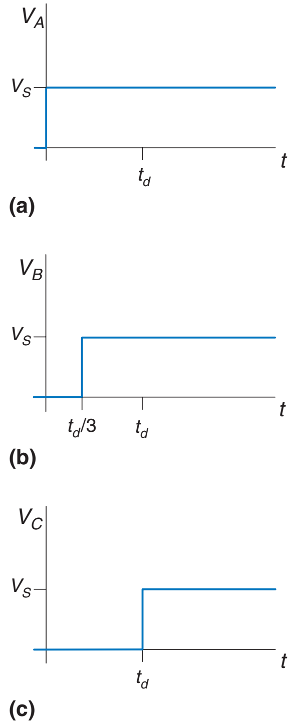


Figure eA.20 Voltage waveforms for Figure eA.19 at points *A*, *B*, and *C*

Example eA.2 TRANSMISSION LINE WITH MATCHED SOURCE AND LOAD TERMINATIONS

Figure eA.22 shows a transmission line with matched source and load impedances Z_S and Z_L. Plot the voltage at nodes A, B, and C versus time. When does the system reach steady state, and what is the equivalent circuit at steady state?

Solution When the voltage source has a source impedance Z_S in series with the transmission line, part of the voltage drops across Z_S, and the remainder propagates down the transmission line. At first, the transmission line behaves as an impedance Z_0, because the load at the end of the line cannot possibly influence the behavior of the line until a speed-of-light delay has elapsed—that is, the signal at the source end cannot even know what is at the load end until it

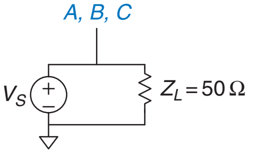


Figure eA.21 Equivalent circuit of Figure eA.19 at steady state

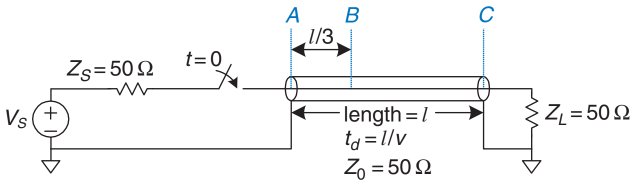


Figure eA.22 Transmission line with matched source and load impedances

reaches it. Hence, by the *voltage divider equation*, the incident voltage flowing down the line is

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{eA.6}$$

Thus, at $t = 0$, a wave of voltage, $V = \frac{V_S}{2}$, is sent down the line from point A. Again, the signal reaches point B at time $t_d/3$ and point C at t_d, as shown in Figure eA.23. All of the current is absorbed by the load impedance Z_L, so the circuit enters steady state at $t = t_d$. In steady state, the entire line is at $V_S/2$, just as the steady-state equivalent circuit in Figure eA.24 would predict.

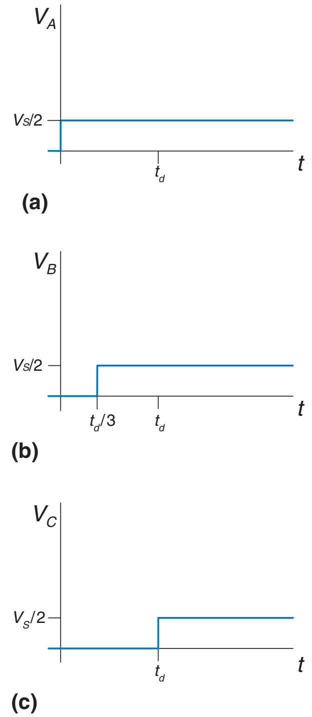


Figure eA.23 Voltage waveforms for Figure eA.22 at points *A*, *B*, and *C*

A.9.2 Open Termination

When the load impedance is not equal to Z_0, the termination cannot absorb all of the current and some of the wave must be reflected. Figure eA.25 shows a transmission line with an open load termination. No current can flow through an open termination, so the current at point C must always be 0.

The voltage on the line is initially zero. At $t = 0$, the switch closes and a wave of voltage, $V = V_S \frac{Z_0}{Z_0 + Z_S} = \frac{V_S}{2}$, begins propagating down the line. Notice that this initial wave is the same as that of Example eA.2 and is independent of the termination because the load at the end of the line cannot influence the behavior at the beginning until at least $2t_d$ has elapsed. This wave reaches point B at $t_d/3$ and point C at t_d, as shown in Figure eA.26.

When the incident wave reaches point C, it cannot continue forward because the wire is open. It must instead reflect back toward the source. The reflected wave also has voltage $V = \frac{V_S}{2}$ because the open termination reflects the entire wave.

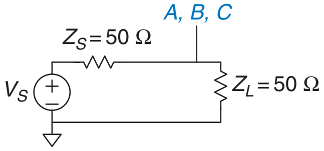


Figure eA.24 Equivalent circuit of Figure eA.22 at steady state

The voltage at any point is the sum of the incident and reflected waves. At time $t = t_d$, the voltage at point C is $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. The reflected wave reaches point B at $5t_d/3$ and point A at $2t_d$. When it reaches point A, the wave is absorbed by the source termination impedance that matches the characteristic impedance of the line. Thus, the system reaches steady state at time $t = 2t_d$, and the transmission line becomes equivalent to an equipotential wire with voltage V_S and current $I = 0$.

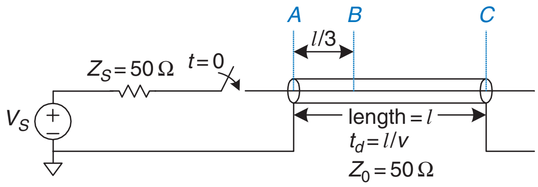


Figure eA.25 **Transmission line with open load termination**

A.9.3 Short Termination

Figure eA.27 shows a transmission line terminated with a short circuit to ground. Thus, the voltage at point C must always be 0.

As in the previous examples, the voltages on the line are initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, begins propagating down the line (Figure eA.28). When it reaches the end of the line, it must reflect with opposite polarity. The reflected wave, with voltage $V = \frac{-V_S}{2}$, adds to the incident wave, ensuring that the voltage at point C remains 0. The reflected wave reaches the source at time $t = 2t_d$ and is absorbed by the source impedance. At this point, the system reaches steady state and the transmission line is equivalent to an equipotential wire with voltage $V = 0$.

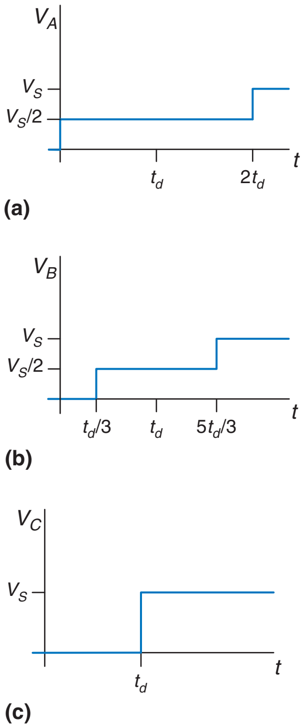


Figure eA.26 **Voltage waveforms for Figure eA.25 at points *A*, *B*, and *C***

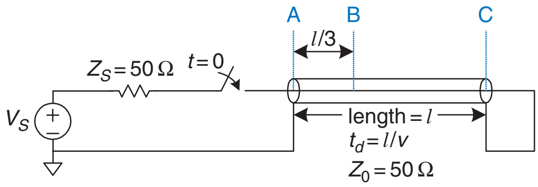


Figure eA.27 **Transmission line with short termination**

A.9.4 Mismatched Termination

The termination impedance is said to be *mismatched* when it does not equal the characteristic impedance of the line. In general, when an incident wave reaches a mismatched termination, part of the wave is absorbed and part is reflected. The reflection coefficient k_r indicates the fraction of the incident wave V_i that is reflected: $V_r = k_r V_i$.

Section A.9.8 derives the reflection coefficient using conservation of current arguments. It shows that, when an incident wave flowing along a transmission line of characteristic impedance Z_0 reaches a termination impedance Z_T at the end of the line, the reflection coefficient is

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0}. \tag{eA.7}$$

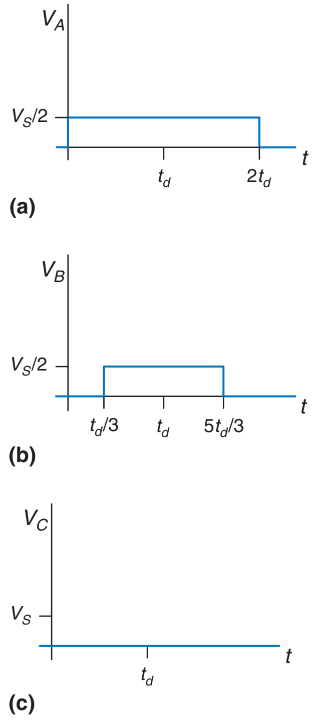


Figure eA.28 Voltage waveforms for Figure eA.27 at points *A*, *B*, and *C*

Note a few special cases. If the termination is an open circuit ($Z_T = \infty$), $k_r = 1$ because the incident wave is entirely reflected (thus, the current out the end of the line remains zero). If the termination is a short circuit ($Z_T = 0$), $k_r = -1$ because the incident wave is reflected with negative polarity (thus, the voltage at the end of the line remains zero). If the termination is a matched load ($Z_T = Z_0$), $k_r = 0$ because the incident wave is completely absorbed.

Figure eA.29 illustrates reflections in a transmission line with a *mismatched load termination* of 75 Ω. $Z_T = Z_L = 75\ \Omega$, and $Z_0 = 50\ \Omega$, so $k_r = 1/5$. As in previous examples, the voltage on the line is initially 0. When the switch closes, a wave of voltage $V = \frac{V_S}{2}$ propagates down the line, reaching the end at $t = t_d$. When the incident wave reaches the termination at the end of the line, one-fifth of the wave is reflected, and the remaining four-fifths flows into the load impedance. Thus, the reflected wave has a voltage $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. The total voltage at point C is the sum of the incoming and reflected voltages, $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. At $t = 2t_d$, the reflected wave reaches point A, where it is absorbed by the matched 50 Ω termination, Z_S. Figure eA.30 plots the voltages and currents along the line. Again, note that in steady state (in this case, at time $t > 2t_d$), the transmission line is equivalent to an equipotential wire, as shown in Figure eA.31. At steady state, the system acts like a voltage divider, so

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

Reflections can occur at both ends of the transmission line. Figure eA.32 shows a transmission line with a source impedance, Z_S, of 450 Ω and an open termination at the load. The reflection coefficients at the load and source, k_{rL} and k_{rS}, are 1 and 4/5, respectively. In this case, waves reflect off both ends of the transmission line until a steady state is reached.

The *bounce diagram* shown in Figure eA.33 helps visualize reflections off both ends of the transmission line. The horizontal axis represents distance along the transmission line, and the vertical axis represents time, increasing downward. The two sides of the bounce

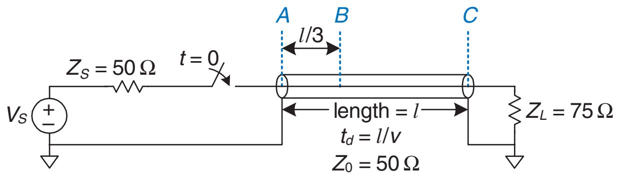


Figure eA.29 Transmission line with mismatched termination

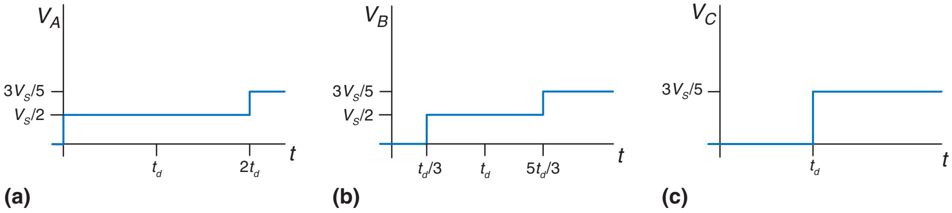


Figure eA.30 Voltage waveforms for Figure eA.29 at points *A*, *B*, and *C*

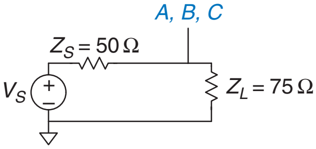


Figure eA.31 Equivalent circuit of Figure eA.29 at steady state

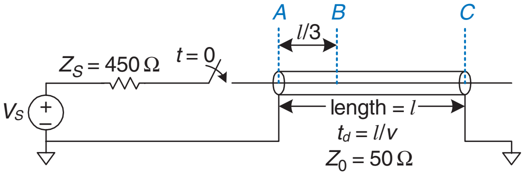


Figure eA.32 Transmission line with mismatched source and load terminations

diagram represent the source and load ends of the transmission line, points *A* and *C*. The incoming and reflected signal waves are drawn as diagonal lines between points *A* and *C*. At time $t = 0$, the source impedance and transmission line behave as a voltage divider, launching a voltage wave of $\frac{V_S}{10}$ from point *A* toward point *C*. At time $t = t_d$, the signal reaches point *C* and is completely reflected ($k_{rL} = 1$). At time $t = 2t_d$, the reflected wave of $\frac{V_S}{10}$ reaches point *A* and is reflected with a reflection coefficient, $k_{rS} = 4/5$, to produce a wave of $\frac{2V_S}{25}$ traveling toward point *C*, and so forth.

The voltage at a given time at any point on the transmission line is the sum of all the incident and reflected waves. Thus, at time $t = 1.1t_d$, the voltage at point *C* is $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. At time $t = 3.1t_d$, the voltage at point *C* is $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, and so forth. Figure eA.34 plots the voltages against time. As *t* approaches infinity, the voltages approach steady state with $V_A = V_B = V_C = V_S$.

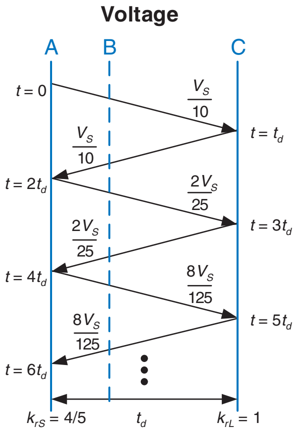


Figure eA.33 Bounce diagram for Figure eA.32

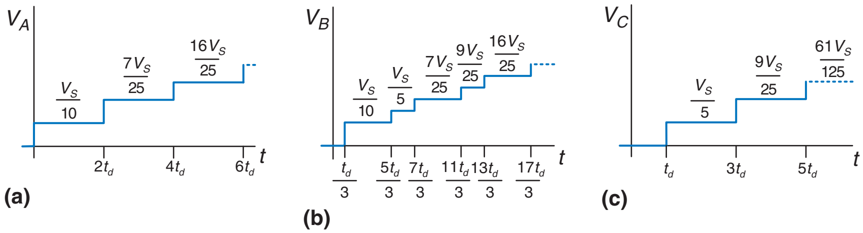


Figure eA.34 **Voltage and current waveforms for Figure eA.32**

A.9.5 When to Use Transmission Line Models

Transmission line models for wires are needed whenever the wire delay, t_d, is longer than a fraction, typically 20%, of the edge rates (rise or fall times) of a signal. If the wire delay is shorter, it has an insignificant effect on the propagation delay of the signal, and the reflections dissipate while the signal is transitioning. If the wire delay is longer, it must be considered in order to accurately predict the propagation delay and waveform of the signal. In particular, reflections may distort the digital characteristic of a waveform, resulting in incorrect logic operations.

Recall that signals travel on a PCB at about 15 cm/ns. For TTL logic, with edge rates of 10 ns, wires must be modeled as transmission lines only if they are longer than 30 cm (10 ns × 15 cm/ns × 20%). PCB traces are usually less than 30 cm, so most traces can be modeled as ideal equipotential wires. In contrast, many modern chips have edge rates of 2 ns or less, so traces longer than about 6 cm (about 2.5 inches) must be modeled as transmission lines. Clearly, use of edge rates that are crisper than necessary just causes difficulties for the designer.

Breadboards lack a ground plane, so the electromagnetic fields of each signal are nonuniform and difficult to model. Moreover, the fields interact with other signals. This can cause strange reflections and crosstalk between signals. Thus, breadboards are unreliable above a few megahertz.

In contrast, PCBs have good transmission lines, with consistent characteristic impedance and velocity along the entire line. As long as they are terminated with a source or load impedance that is matched to the impedance of the line, PCB traces do not suffer from reflections.

A.9.6 Proper Transmission Line Terminations

Two common ways to properly terminate a transmission line exist, as shown in Figure eA.35. In *parallel termination*, the driver has a low impedance ($Z_S \approx 0$). A load resistor Z_L with impedance Z_0 is placed in parallel with the load (between the input of the receiver gate and

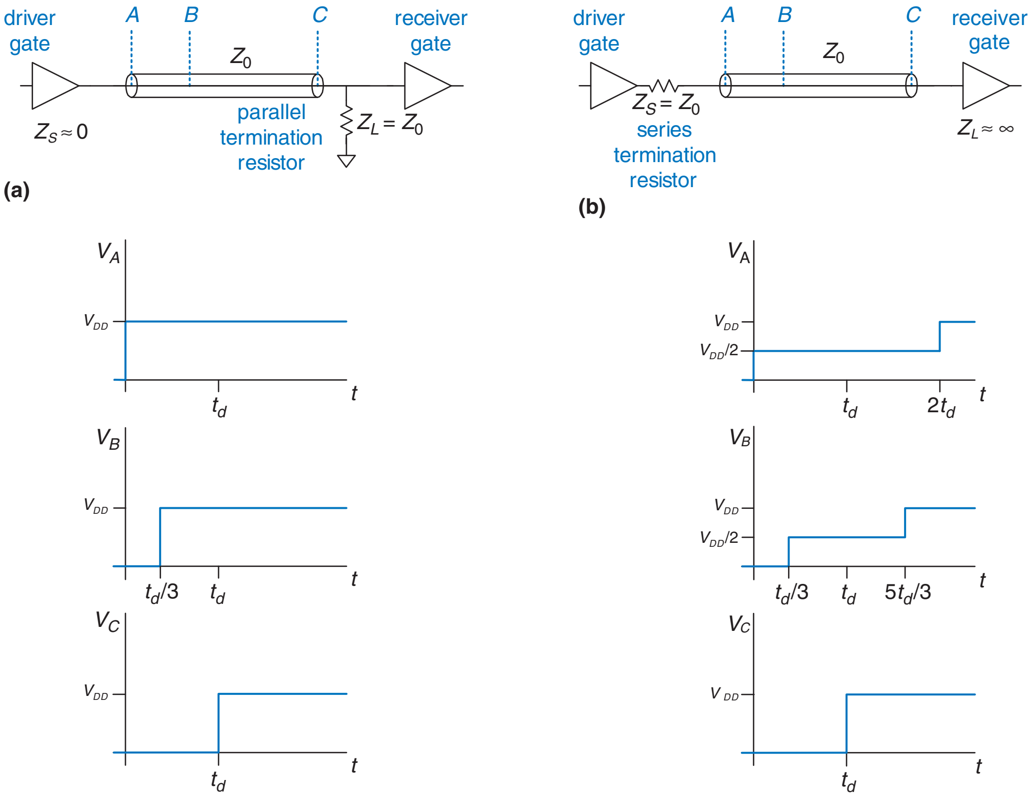


Figure eA.35 Termination schemes: (a) parallel, (b) series

ground). When the driver switches from 0 to V_{DD}, it sends a wave with voltage V_{DD} down the line. The wave is absorbed by the matched load termination, and no reflections take place. In *series termination*, a source resistor Z_S is placed in series with the driver to raise the source impedance to Z_0. The load has a high impedance ($Z_L \approx \infty$). When the driver switches, it sends a wave with voltage $V_{DD}/2$ down the line. The wave reflects at the open circuit load and returns, bringing the voltage on the line up to V_{DD}. The wave is absorbed at the source termination. Both schemes are similar in that the voltage at the receiver transitions from 0 to V_{DD} at $t = t_d$, just as one would desire. They differ in power consumption and in the waveforms that appear elsewhere along the line. Parallel termination dissipates power continuously through the load resistor when the line is at a high voltage. Series termination dissipates no DC power, because the load is an open circuit. However, in series terminated lines, points near the middle of the transmission line initially see a voltage of $V_{DD}/2$, until the reflection returns. If other gates are attached to the middle of the line, they will momentarily see an illegal logic level. Therefore, series termination works best for *point-to-point* communication with a single driver and a single receiver. Parallel termination is better for a *bus* with multiple receivers, because receivers at the middle of the line never see an illegal logic level.

A.9.7 Derivation of Z_0*

Z_0 is the ratio of voltage to current in a wave propagating along a transmission line. This section derives Z_0; it assumes some previous knowledge of resistor-inductor-capacitor (RLC) circuit analysis.

Imagine applying a step voltage to the input of a semi-infinite transmission line (so that there are no reflections). Figure eA.36 shows the semi-infinite line and a model of a segment of the line of length dx. R, L, and C are the values of resistance, inductance, and capacitance per unit length. Figure eA.36(b) shows the transmission line model with a resistive component, R. This is called a *lossy* transmission line model, because energy is dissipated, or lost, in the resistance of the wire. However, this loss is often negligible, and we can simplify the analysis by ignoring the resistive component and treating the transmission line as an *ideal* transmission line, as shown in Figure eA.36(c).

Voltage and current are functions of time and space throughout the transmission line, as given by Equations eA.8 and eA.9.

$$\frac{\partial}{\partial x}V(x,t) = L\frac{\partial}{\partial t}I(x,t) \tag{eA.8}$$

$$\frac{\partial}{\partial x}I(x,t) = C\frac{\partial}{\partial t}V(x,t) \tag{eA.9}$$

Taking the space derivative of Equation eA.8 and the time derivative of Equation eA.9 and substituting gives Equation eA.10, the *wave equation*.

$$\frac{\partial^2}{\partial x^2}V(x,t) = LC\frac{\partial^2}{\partial t^2}V(x,t) \tag{eA.10}$$

Z_0 is the ratio of voltage to current in the transmission line, as illustrated in Figure eA.37(a). Z_0 must be independent of the length of the line because the behavior of the wave cannot depend on things at a distance. Because it is independent of length, the impedance must still

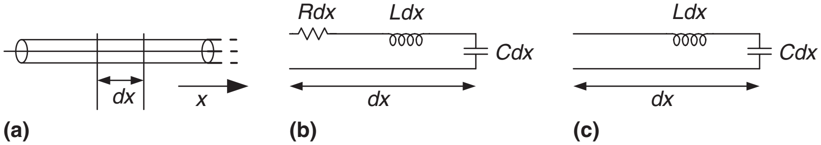


Figure eA.36 Transmission line models: (a) semi-infinite cable, (b) lossy, (c) ideal

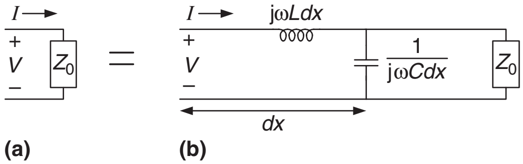


Figure eA.37 Transmission line model: (a) for entire line and (b) with additional length, *dx*

equal Z_0 after the addition of a small amount of transmission line, dx, as shown in Figure eA.37(b).

Using the impedances of an inductor and a capacitor, we rewrite the relationship of Figure eA.37 in equation form:

$$Z_0 = j\omega L dx + [Z_0 \| (1 / (j\omega C dx))] \tag{eA.11}$$

Rearranging, we get

$$Z_0^2 (j\omega C) - j\omega L + \omega^2 Z_0 L C dx = 0 \tag{eA.12}$$

Taking the limit as dx approaches 0, the last term vanishes and we find that

$$Z_0 = \sqrt{\frac{L}{C}} \tag{eA.13}$$

A.9.8 Derivation of the Reflection Coefficient*

The reflection coefficient k_r is derived using conservation of current. Figure eA.38 shows a transmission line with characteristic impedance Z_0 and load impedance Z_L. Imagine an incident wave of voltage V_i and current I_i. When the wave reaches the termination, some current I_L flows through the load impedance, causing a voltage drop V_L. The remainder of the current reflects back down the line in a wave of voltage V_r and current I_r. Z_0 is the ratio of voltage to current in waves propagating along the line, so $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

The voltage on the line is the sum of the voltages of the incident and reflected waves. The current flowing in the positive direction on the line is the difference between the currents of the incident and reflected waves.

$$V_L = V_i + V_r \tag{eA.14}$$

$$I_L = I_i - I_r \tag{eA.15}$$

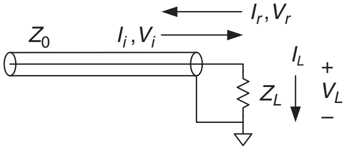


Figure eA.38 Transmission line showing incoming, reflected, and load voltages and currents

Using Ohm's law and substituting for I_L, I_i, and I_r in Equation eA.15, we get

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{eA.16}$$

Rearranging, we solve for the reflection coefficient, k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{eA.17}$$

A.9.9 Putting It All Together

Transmission lines model the fact that signals take time to propagate down long wires because the speed of light is finite. An ideal transmission line has uniform inductance L and capacitance C per unit length and zero resistance. The transmission line is characterized by its characteristic impedance Z_0 and delay t_d, which can be derived from the inductance, capacitance, and wire length. The transmission line has significant delay and noise effects on signals whose rise/fall times are less than about $5t_d$. This means that, for systems with 2 ns rise/fall times, PCB traces longer than about 6 cm must be analyzed as transmission lines to accurately understand their behavior.

A digital system consisting of a gate driving a long wire attached to the input of a second gate can be modeled with a transmission line as shown in Figure eA.39. The voltage source, source impedance Z_S, and switch model the first gate switching from 0 to 1 at time 0. The driver gate cannot supply infinite current; this is modeled by Z_S. Z_S is usually small for a logic gate, but a designer may choose to add a resistor in series with the gate to raise Z_S and match the impedance of the line. The input to the second gate is modeled as Z_L. CMOS circuits usually have little input current, so Z_L may be close to infinity. The designer may also choose to add a resistor in parallel with the second gate, between the gate input and ground, so that Z_L matches the impedance of the line.

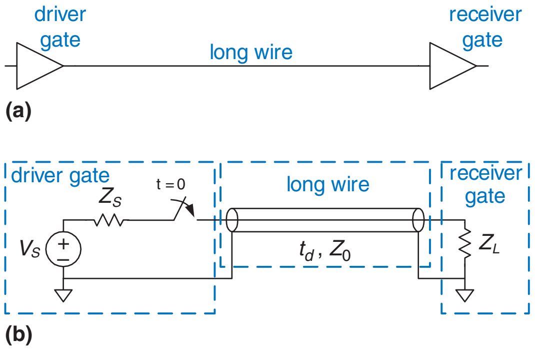


Figure eA.39 Digital system modeled with transmission line

When the first gate switches, a wave of voltage is driven onto the transmission line. The source impedance and transmission line form a voltage divider, so the voltage of the incident wave is

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{eA.18}$$

At time t_d, the wave reaches the end of the line. Part is absorbed by the load impedance and part is reflected. The reflection coefficient k_r indicates the portion that is reflected: $k_r = V_r/V_i$, where V_r is the voltage of the reflected wave and V_i is the voltage of the incident wave.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{eA.19}$$

The reflected wave adds to the voltage already on the line. It reaches the source at time $2t_d$, where part is absorbed and part is again reflected. The reflections continue back and forth, and the voltage on the line eventually approaches the value that would be expected if the line were a simple equipotential wire.

A.10 ECONOMICS

Although digital design is so much fun that some of us would do it for free, most designers and companies intend to make money. Therefore, economic considerations are a major factor in design decisions.

The cost of a digital system can be divided into *nonrecurring engineering* (*NRE*) *costs*, and *recurring costs*. NRE accounts for the cost of designing the system. It includes the salaries of the design team, computer and software costs, and the costs of producing the first working unit. The fully loaded cost of a designer in the United States in 2021 (including salary, health insurance, retirement plan, and a computer with design tools) was roughly $200,000 per year, so design costs can be significant. Recurring costs are the cost of each additional unit; this includes components, manufacturing, marketing, technical support, and shipping.

The sales price must cover not only the cost of the system but also other costs, such as office rental, taxes, and salaries of staff who do not directly contribute to the design (such as the janitor and the CEO). After all of these expenses, the company should still make a profit.

Example eA.3 BEN TRIES TO MAKE SOME MONEY

Ben Bitdiddle has designed a crafty circuit for counting raindrops. He decides to sell the device and try to make some money, but he needs help deciding what implementation to use. He decides to use either an FPGA or an ASIC. The development kit to design and test the FPGA costs $1500. Each FPGA costs $17. The ASIC costs $600,000 for a mask set and $4 per chip.

Regardless of what chip implementation he chooses, Ben needs to mount the packaged chip on a printed circuit board (PCB), which will cost him $1.50 per board. He thinks he can sell 1,000 devices per month. Ben has coerced a team of bright undergraduates into designing the chip for their senior project so that it doesn't cost him anything to design.

If the sales price has to be twice the cost (100% profit margin), and the product life is 2 years, which implementation is the better choice?

Solution Ben figures out the total cost for each implementation over 2 years, as shown in Table eA.4. Over 2 years, Ben plans on selling 24,000 devices; the total cost is given in Table eA.4 for each option. If the product life is only two years, the FPGA option is clearly superior. The per-unit cost is $445,500/24,000 = $18.56, and the sales price is $37.13 per unit to give a 100% profit margin. The ASIC option would have cost $732,000/24,000 = $30.50 and would have sold for $61 per unit.

Table eA.4 ASIC vs FPGA costs

Cost	ASIC	FPGA
NRE	$600,000	$1500
Chip	$4	$17
PCB	$1.50	$1.50
TOTAL	$600,000 + (24,000 × $5.50) = $732,000	$1500 + (24,000 × $18.50) = $445,500
Per unit	$30.50	$18.56

Example eA.4 BEN GETS GREEDY

After seeing the marketing ads for his product, Ben thinks he can sell even more chips per month than originally expected. If he were to choose the ASIC option, how many devices per month would he have to sell to make the ASIC option more profitable than the FPGA option?

Solution Ben solves for the minimum number of units, N, that he would need to sell in 2 years:

$$\$600,000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

Solving the equation gives N = 46,039 units, or 1,919 units per month. He would need to almost double his monthly sales to benefit from the ASIC solution.

Example eA.5 BEN GETS LESS GREEDY

Ben realizes that his eyes have gotten too big for his stomach, and he doesn't think he can sell more than 1,000 devices per month. But he does think the product life can be longer than 2 years. At a sales volume of 1,000 devices per month, how long would the product life have to be to make the ASIC option worthwhile?

Solution If Ben sells more than 46,039 units in total, the ASIC option is the best choice. So, Ben would need to sell at a volume of 1,000 per month for at least 47 months (rounding up), which is almost 4 years. By then, his product is likely to be obsolete.

Chips are usually purchased from a distributor rather than directly from the manufacturer (unless you are ordering tens of thousands of units). Digikey (`www.digikey.com`) and Arrow (`www.arrow.com`) are leading distributors that sell a wide variety of electronics. Jameco (`www.jameco.com`) and All Electronics (`www.allelectronics.com`) have eclectic catalogs that are competitively priced and well suited to hobbyists.

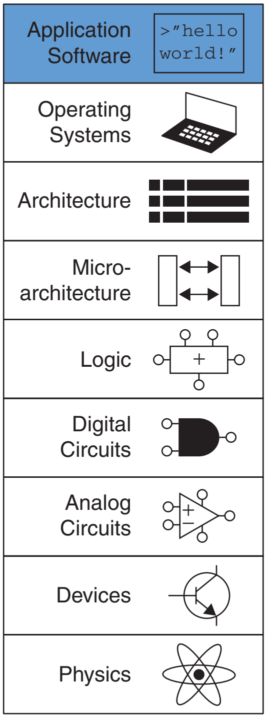
B RISC-V Instruction Set Summary

The inside covers of the textbook have a handy summary of the entire RISC-V instruction set.

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00016-7

C Programming

eC



C.1 INTRODUCTION

The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the appendix so that readers may easily cover or skip it depending on previous experience.

Programmers use many different languages to tell a computer what to do. Fundamentally, computers process instructions in *machine language* consisting of 1's and 0's, as is explored in Chapter 6. But programming in machine language is tedious and slow, leading programmers to use more abstract languages to get their meaning across more efficiently. Table eC.1 lists some examples of languages at various levels of abstraction.

One of the most popular programming languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C (including a family of closely related languages such as C++, C#, and Objective C) is the most widely used language in existence. Its popularity stems from a number of factors, including its:

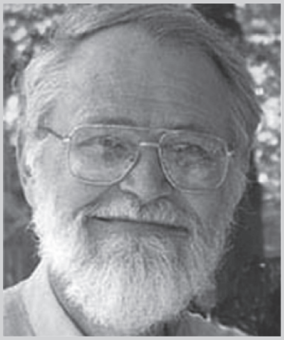
- Availability on a tremendous variety of platforms, from supercomputers down to embedded microcontrollers
- Relative ease of use, with a huge user base

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00017-9

Table eC.1 **Languages at roughly decreasing levels of abstraction**

Language	Description
MATLAB	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

Dennis Ritchie, 1941–2011



Brian Kernighan, 1942–

C was formally introduced in 1978 by Brian Kernighan and Dennis Ritchie's classic book, *The C Programming Language*. In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89. Shortly thereafter, in 1990, this standard was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO/IEC updated the standard in 1999 to what is called C99, which we will be discussing in this text.

- Moderate level of abstraction, providing higher productivity than assembly language, yet giving the programmer a good understanding of how the code will be executed
- Suitability for generating high-performance programs
- Ability to interact directly with the hardware

This chapter is devoted to C programming for a variety of reasons. Most importantly, C allows the programmer to directly access addresses in memory, illustrating the connection between hardware and software emphasized in this book. C is a practical language that all engineers and computer scientists should know. Its uses in many aspects of implementation and design—for example, software development, embedded systems programming, and simulation—make proficiency in C a vital and marketable skill.

The following sections describe the overall syntax of a C program, discussing each part of the program—including the header, function and variable declarations, data types, and commonly used functions provided in libraries. Chapter 9 (available as a web supplement—see Preface) describes a hands-on application by using C to program SparkFun's RED-V RedBoard, which contains a RISC-V microcontroller.

SUMMARY

- **High-level programming:** High-level programming is useful at many levels of design, from writing analysis or simulation software to programming microcontrollers that interact with hardware.
- **Low-level access:** C code is powerful because, in addition to high-level constructs, it provides access to low-level hardware and memory.

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer. C Code Example eC.1 is a simple C program that prints the phrase "Hello world!" to the *console*, the computer screen. C programs are generally contained in one or more text files that end in ".c." Good programming style requires a file name that indicates the contents of the program—for example, this file could be called hello.c.

C is the language used to program such ubiquitous systems as Linux, Windows, and iOS. C is a powerful language because of its direct access to hardware. As compared with other high-level languages—for example, Perl and MATLAB—C does not have as much built-in support for specialized operations such as file manipulation, pattern matching, matrix manipulation, and graphical user interfaces. It also lacks features to protect the programmer from common mistakes, such as writing data past the end of an array. Its power combined with its lack of protection has assisted hackers who exploit poorly written software to break into computer systems.

C Code Example eC.1 SIMPLE C PROGRAM

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void) {
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C.2.1 C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the `main` function, which is where the program starts executing. Most programs use other functions defined elsewhere in the C code and/or in a library. The overall sections of the `hello,c` program are the header, the `main` function, and the body.

While this chapter provides a fundamental understanding of C programming, entire texts are written that describe C in depth. One of our favorites is the classic text *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of C. This text gives a concise description of the nuts and bolts of C. Another good text is *A Book on C* by Al Kelley and Ira Pohl.

Header: `#include <stdio.h>`

The header includes the *library functions* needed by the program. In this case, the program uses the `printf` function, which is part of the standard I/O library, `stdio.h`. See Section C.9 for further details on C's built-in libraries.

Main function: `int main(void)`

All C programs must include exactly one `main` function. Execution of the program occurs by running the code inside `main`, called the *body* of `main`. Function syntax is described in Section C.6. The body of a function contains a sequence of *statements*. Each statement ends with a semicolon. `int` denotes that the `main` function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

Body: `printf("Hello world!\n");`

The body of this `main` function contains one statement, a call to the `printf` function, which prints the phrase "Hello world!" followed by a newline character indicated by the special sequence "`\n`". Further details about I/O (input/output) functions are described in Section C.9.1.

All programs follow the general format of the simple `hello.c` program. Of course, very complex programs may contain millions of lines of code and span hundreds of files.

C.2.2 Running a C Program

C programs can be run on many different machines. This *portability* is another advantage of C. The program is first compiled on the desired machine using the C *compiler*. Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler). Here, we show how to compile and run a C program using gcc, which is freely available for download. It runs directly on Linux machines and is accessible under the Cygwin environment on Windows machines. It is also available for many embedded systems, such as SparkFun's RED-V RedBoard, which includes a RISC-V microcontroller. The general process described below of C file creation, compilation, and execution is the same for any C program.

1. Create the text file, for example, `hello.c`.
2. In a terminal window, change to the directory that contains the file `hello.c` and type `gcc hello.c` at the command prompt.
3. The compiler creates an executable file. By default, the executable is called `a.out` (or `a.exe` on Windows machines).
4. At the command prompt, type `./a.out` (or `./a.exe` on Windows) and press return.
5. "Hello world!" will appear on the screen.

SUMMARY

- `filename.c`: C program files are typically named with a .c extension.
- `main`: Each C program must have exactly one `main` function.
- `#include`: Most C programs use functions provided by built-in libraries. These functions are used by writing `#include <library.h>` at the top of the C file.
- `gcc filename.c`: C files are converted into an executable using a compiler such as the GNU compiler (gcc) or the C compiler (cc).
- **Execution:** After compilation, C programs are executed by typing `./a.out` (or `./a.exe`) at the command line prompt.

C.3 COMPILATION

A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable. Entire textbooks are written on compilers, but we describe them here briefly. The overall operation of the compiler is to (1) preprocess the file by including referenced libraries and expanding macro definitions, (2) ignore all unnecessary information such as comments, (3) translate the high-level code into simple instructions native to the processor that are represented in binary, called machine language, and (4) compile all the instructions into a single binary executable that can be read and executed by the computer. Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run. For example, the RISC-V machine language is covered in Chapter 6 in detail.

C.3.1 Comments

Programmers use comments to describe code at a high level and clarify code function. Anyone who has read uncommented code can attest to its importance. C programs use two types of comments: Single-line comments begin with // and terminate at the end of the line; multiple-line comments begin with /* and end with */. While comments are critical to the organization and clarity of a program, they are ignored by the compiler.

```
// This is an example of a one-line comment.
/* This is an example
  of a multi-line comment. */
```

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose. The comment below could be included at the top of the hello.c file.

```
// hello.c
// 15 Jan 2021 Sarah.Harris@unlv.edu, David_Harris@hmc.edu
//
// This program prints "Hello world!" to the screen
```

C.3.2 #define

Constants are named using the #define directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you write a program that allows at most 5 user guesses. You can use #define to identify that number.

```
#define MAXGUESSES 5
```

The # indicates that this line in the program will be handled by the *preprocessor*. Before compilation, the preprocessor replaces each occurrence of the identifier MAXGUESSES in the program with 5. By convention, #define lines are located at the top of the file and identifiers are written in all capital letters. By defining constants in one location and then using the identifier in the program, the program remains consistent, and the value is easily modified—it need only be changed at the #define line instead of at each line in the code where the value is used.

C Code Example eC.2 shows how to use the #define directive to convert inches to centimeters. The variables inch and cm are declared to be float, which means they represent single-precision floating-point numbers. If the conversion factor (INCH2CM) were used throughout a large program, having it declared using #define obviates errors due to typos (e.g., typing 2.53 instead of 2.54) and makes it easy to find and change (e.g., if more significant digits were required).

Number constants in C default to decimal but can also be hexadecimal (prefix "0x") or octal (prefix "0")[1]. Binary constants are not defined in C99 but are supported by some compilers (prefix "0b"). For example, the following assignments are equivalent:

```
char x = 37;
char x = 0x25;
char x = 045;
char x = 0b100101;
```

Globally defined constants eradicate *magic numbers* from a program. A magic number is a constant that shows up in a program without a name. The presence of magic numbers in a program often introduces tricky bugs—for example, when the number is changed in one location but not another.

C Code Example eC.2 USING #define TO DECLARE CONSTANTS

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

C.3.3 #include

Modularity encourages us to split programs across separate files and functions. Commonly used functions can be grouped together for easy reuse. Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the #include preprocesser directive. *Standard libraries* that provide commonly used functions are accessed in this way. For example, the following line is required to use the functions defined in the standard input/output (I/O) library, such as printf.

```
#include <stdio.h>
```

The ".h" postfix of the include file indicates that it is a header file. While #include directives can be placed anywhere in the file before the included

[1] The prefixes for hexadecimal, octal, and binary (0x, 0, and 0b) start with the number 0, not the letter O.

functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included but must use quotation marks (" ") around the file name instead of brackets (< >). For example, a user-created header file called `myfunctions.h` would be included using the following line.

```
#include "myfunctions.h"
```

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included.

SUMMARY

- **Comments:** C provides single-line comments (`//`) and multi-line comments (`/* */`).
- `#define NAME val`: the `#define` directive allows an identifier (`NAME`) to be used throughout the program. Before compilation, all instances of `NAME` are replaced with `val`.
- `#include`: `#include` allows common functions to be used in a program. For built-in libraries, include the following line at the top of the code: `#include <library.h>`. To include a user-defined header file, the name must be in quotes, listing the path relative to the current directory as needed: that is, `#include "other/myFuncs.h"`.

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example, the following declaration states that the variable is of type `char` (which is a 1-byte type), and that the variable name is `x`. The compiler decides where to place this 1-byte variable in memory.

```
char x;
```

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure eC.1. A variable occupies one or more bytes of memory; the address of multiple-byte variables is indicated by the lowest numbered byte. The type of a variable indicates whether to interpret the byte(s) as an integer, floating-point number, or other type. The rest of this section describes C's primitive data types, the declaration of global and local variables, and the initialization of variables.

Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., `int`, `while`, etc.), may not start with a number (i.e., `int 1x;` is not a valid declaration), and may not include special characters such as `\`, `*`, `?`, or `-`. Underscores (_) are allowed.

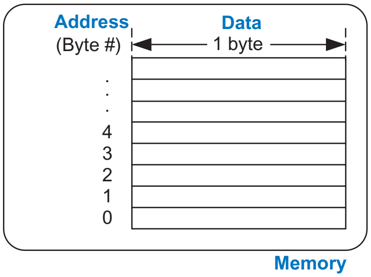


Figure eC.1 C's view of memory

C.4.1 Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized as integers, floating-point variables, and characters. An integer represents a two's complement or unsigned number within a finite range. A floating-point variable uses IEEE floating-point representation to describe real numbers with a finite range and precision. A character can be viewed as either an ASCII value or an 8-bit integer.[2] Table eC.2 lists the size and range of each primitive type. Integers may be 16, 32, or 64 bits. They use two's complement unless qualified as `unsigned`.

Table eC.2 Primitive data types and sizes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`int16_t`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`uint16_t`	16	0	$2^{16} - 1 = 65{,}535$
`int32_t`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`uint32_t`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`int64_t`	64	-2^{63}	$2^{63} - 1$
`uint64_t`	64	0	$2^{64} - 1$
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

[2] Technically, the C99 standard defines a character as "a bit representation that fits in a byte," without requiring a byte to be 8 bits. However, current systems define a byte as 8 bits.

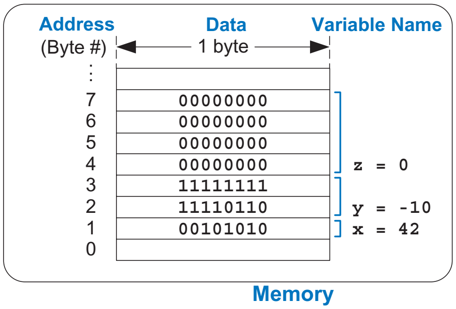


Figure eC.2 Variable storage in memory for C Code Example eC.3

The size of the `int` type is machine dependent and is often the native word size of the machine. For example, on a 32-bit processor, the size of an `int` or `unsigned int` is 32 bits. On a 16-bit processor, an `int` is usually 16 bits. However, compilers for 64-bit processors typically use 32 bits for `int`s to reduce subtle bugs porting old code that assumed this size. If you care about the size of a data type, use `int16_t`, `int32_t`, or `int64_t` to explicitly define the size. (These are signed datatypes; their unsigned counterparts are `uint16_t`, etc.) Floating-point numbers use 32 or 64 bits for single- or double-precision, respectively. Characters are 8 bits.

C Code Example eC.3 shows the declaration of variables of different types. As shown in Figure eC.2, `x` requires one byte of data, `y` requires two, and `z` requires four. The program decides where these bytes are stored in memory, but each type always requires the same amount of data. For illustration, the addresses of `x`, `y`, and `z` in this example are 1, 2, and 4. Variable names are case-sensitive, so, for example, the variable `x` and the variable `X` are two different variables. (But it would be very confusing to use both in the same program!)

C Code Example eC.3 EXAMPLE DATA TYPES

```
// Examples of several data types and their binary representations
unsigned char x = 42;        // x = 00101010
int16_t y = −10;             // y = 11111111 11110110
unit32_t z = 2;              // z = 00000000 00000000 00000000 00000010
```

C.4.2 Global and Local Variables

Global and local variables differ in where they are declared and where they are visible. A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions. Global variables should be used sparingly because they violate the principle of modularity, making large programs more difficult to read. However, a variable accessed by many functions can be made global.

A local variable is declared inside a function and can only be used by that function. Therefore, two functions can have local variables with

The *scope* of a variable is the context in which it can be used. For example, for a local variable, its scope is the function in which it is declared. It is out of scope everywhere else.

the same names without interfering with each other. Local variables are declared at the beginning of a function. They cease to exist when the function ends and are recreated when the function is called again. They do not retain their value from one invocation of a function to the next.

C Code Examples eC.4 and eC.5 compare programs using global versus local variables. In C Code Example eC.4, the global variable `max` can be accessed by any function. Using a local variable, as shown in C Code Example eC.5, is the preferred style because it preserves the well-defined interface of modularity.

C Code Example eC.4 GLOBAL VARIABLES

```
// Use a global variable to find and print the maximum of 3 numbers

int max;            // global variable holding the maximum value

void findMax(int a, int b, int c) {
  max = a;
  if (b > max) {
    if (c > b) max = c;
    else       max = b;
  } else if (c > max) max = c;
}

void printMax(void) {
  printf("The maximum number is: %d\n", max);
}

int main(void) {
  findMax(4, 3, 7);
  printMax();
}
```

C Code Example eC.5 LOCAL VARIABLES

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
  int result = a; // local variable holding the maximum value

  if (b > result) {
    if (c > b) result = c;
    else       result = b;
  } else if (c > result) result = c;

  return result;
}

void printMax(int m) {
  printf("The maximum number is: %d\n", m);
}

int main(void) {
  int max;

  max = getMax(4, 3, 7);
  printMax(max);
}
```

C.4.3 Initializing Variables

A variable needs to be *initialized*—assigned a value—before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had the last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program. C Code Example eC.3 shows variables initialized at the same time they are declared. C Code Example eC.4 shows how variables are initialized before their use, but after declaration; the global variable `max` is initialized by the `getMax` function before it is read by the `printMax` function. Reading from uninitialized variables is a common programming error and can be tricky to debug.

SUMMARY

- **Variables:** Each variable is defined by its data type, name, and memory location. A variable is declared as `data type name`.
- **Data types:** A data type describes the size (number of bytes) and representation (interpretation of the bytes) of a variable. Table eC.2 lists C's built-in data types.
- **Memory:** C views memory as a list of bytes. Memory stores variables and associates each variable with an address (byte number).
- **Global variables:** Global variables are declared outside of all functions and can be accessed anywhere in the program.
- **Local variables:** Local variables are declared within a function and can be accessed only within that function.
- **Variable initialization:** Each variable must be initialized before it is read. Initialization can happen either at declaration or afterward.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

```
y = a + 3;
```

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants. C supports the operators shown in Table eC.3, listed by category and in order of decreasing precedence. For example, multiplicative operators take precedence over additive operators. Within the same category, operators are evaluated in the order that they appear in the program.

Table eC.3 Operators listed by decreasing precedence

Category	Operator	Description	Example
Unary	++	post-increment	a++; // a = a+1
	--	post-decrement	x--; // x = x-1

	&	memory address of a variable	x = &y; // x = the memory
			// address of y

	~	bitwise NOT	z = ~a;
	!	Boolean NOT	!x
	-	negation	y = -a;
	++	pre-increment	++a; // a = a + 1
	--	pre-decrement	--x; // x = x - 1

	(type)	casts a variable to (type)	x = (int)c; // cast c to an int and
			// assign it to x

	sizeof()	size of a variable or type in bytes	int32_t y;
			x = sizeof(y); // x = 4

Multiplicative	*	multiplication	y = x * 12;
	/	division	z = 9 / 3; // z = 3
	%	modulo	z = 5 % 2; // z = 1
Additive	+	addition	y = a + 2;
	-	subtraction	y = a - 2;
Bitwise Shift	<<	bitshift left	z = 5 << 2; // z = 0b00010100
	>>	bitshift right	x = 9 >> 3; // x = 0b00000001
Relational	==	equals	y == 2
	!=	not equals	x != 7
	<	less than	y < 12
	>	greater than	val > max
	<=	less than or equal	z <= 2
	>=	greater than or equal	y >= 10

Table eC.3 Operators listed by decreasing precedence—cont'd

Category	Operator	Description	Example
Bitwise	&	bitwise AND	`y = a & 15;`
	^	bitwise XOR	`y = 2 ^ 3;`
	|	bitwise OR	`y = a | b;`
Logical	&&	Boolean AND	`x && y`
	||	Boolean OR	`x || y`

Ternary	? :	ternary operator	`y = x ? a : b; // if x is TRUE,`
			`// y=a, else y=b`

Assignment	=	assignment	`x = 22;`
	+=	addition and assignment	`y += 3;      // y = y + 3`
	−=	subtraction and assignment	`z −= 10;     // z = z − 10`
	*=	multiplication and assignment	`x *= 4;      // x = x * 4`
	/=	division and assignment	`y /= 10;     // y = y / 10`
	%=	modulo and assignment	`x %= 4;      // x = x % 4`
	>>=	bitwise right-shift and assignment	`x >>= 5;     // x = x >> 5`
	<<=	bitwise left-shift and assignment	`x <<= 2;     // x = x << 2`
	&=	bitwise AND and assignment	`y &= 15;     // y = y & 15`
	|=	bitwise OR and assignment	`x |= y;      // x = x | y`
	^=	bitwise XOR and assignment	`x ^= y;      // x = x ^ y`

Unary operators, also called monadic operators, have a single operand. Ternary operators have three operands, and all others have two. The ternary operator (from the Latin *ternarius*, meaning consisting of three) chooses the second or third operand depending on whether the first value is TRUE (nonzero) or FALSE (zero), respectively. C Code Example eC.6 shows how to compute `y = max(a,b)` using the ternary operator, along with an equivalent but more verbose `if/else` statement.

The Truth, the Whole Truth, and Nothing But the Truth

C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as `if` and `while`, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

C Code Example eC.6 (a) TERNARY OPERATOR, AND (b) EQUIVALENT if/else STATEMENT

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer
(b) if (a > b) y = a;
    else       y = b;
```

Simple assignment uses the = operator. C code also allows for compound assignment, that is, assignment after a simple operation such as addition (+=) or multiplication (*=). In compound assignments, the variable on the left side is both operated on and assigned the result. C Code Example eC.7 shows these and other C operations. Binary values in the comments are indicated with the prefix "0b."

C Code Example eC.7 OPERATOR EXAMPLES

<table>
<tr><th>Expression</th><th>Result</th><th>Notes</th></tr>
<tr><td>53 / 14</td><td>3</td><td>Integer division truncates</td></tr>
<tr><td>53 % 14</td><td>11</td><td>53 mod 14</td></tr>
<tr><td>0x2C && 0xE   // 0b101100 && 0b1110</td><td>1</td><td>Logical AND</td></tr>
<tr><td>0x2C || 0xE   // 0b101100 || 0b1110</td><td>1</td><td>Logical OR</td></tr>
<tr><td>0x2C & 0xE   // 0b101100 & 0b1110</td><td>0xC (0b001100)</td><td>Bitwise AND</td></tr>
<tr><td>0x2C | 0xE   // 0b101100 | 0b1110</td><td>0x2E (0b101110)</td><td>Bitwise OR</td></tr>
<tr><td>0x2C ^ 0xE   // 0b101100 ^ 0b1110</td><td>0x22 (0b100010)</td><td>Bitwise XOR</td></tr>
<tr><td>0xE << 2   // 0b1110 << 2</td><td>0x38 (0b111000)</td><td>Left shift by 2</td></tr>
<tr><td>0x2C >> 3   // 0b101100 >> 3</td><td>0x5 (0b101)</td><td>Right shift by 3</td></tr>
<tr><td>x = 14;
x += 2;</td><td>x = 16</td><td></td></tr>
<tr><td>y = 0x2C;   // y = 0b101100
y &= 0xF;   // y = y & 0b1111</td><td>y = 0xC (0b001100)</td><td></td></tr>
<tr><td>x = 14; y = 83;
y = y + x++;</td><td>x = 15, y = 97</td><td>Increment x after using it</td></tr>
<tr><td>x = 14; y = 83;
y = y + ++x;</td><td>x = 15, y = 98</td><td>Increment x before using it</td></tr>
</table>

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example eC.8 shows the `sum3` function. The function declaration begins with the return type, `int`, followed by the name, `sum3`, and the inputs enclosed within parentheses (`int a, int b, int c`). Curly braces `{}` enclose the body of the function, which may contain zero or more statements. The return statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can return only a single value.

Functions are also referred to as procedures.

C Code Example eC.8 `sum3` **FUNCTION**

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

After the following call to `sum3`, `y` holds the value 42.

```
int y = sum3(10, 15, 17);
```

Although a function may have inputs and outputs, neither is required. C Code Example eC.9 shows a function with no inputs or outputs. The keyword `void` before the function name indicates that nothing is returned. `void` between the parentheses indicates that the function has no input arguments.

C Code Example eC.9 FUNCTION `printPrompt` **WITH NO INPUTS OR OUTPUTS**

```
// Print a prompt to the console
void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Nothing between the parentheses also indicates no input arguments. So, in this case, we could have written:

```
void printPrompt()
```

A function must be declared in the code before it is called. This may be done by placing the called function earlier in the file. For this reason, `main` is often placed at the end of the C file after all of the functions it calls. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line of the function; it declares the return type, function name, and function inputs. For example, the function prototypes for the functions in C Code Examples eC.8 and eC.9 are:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

C Code Example eC.10 shows how function prototypes are used. Even though the functions themselves are after `main`, the function prototypes at the top of the file allow them to be used in `main`.

With careful ordering of functions, prototypes may be unnecessary. However, they are unavoidable in certain cases, such as when function `f1` calls `f2` and `f2` calls `f1`. It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

C Code Example eC.10 FUNCTION PROTOTYPES

```
#include <stdio.h>
// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);
int main(void) {
  int y = sum3(10, 15, 20);
  printf("sum3 result: %d\n", y);
  printPrompt();
}
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically, function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names so you don't have to constantly look up the correct capitalization. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., printPrompt), or to use underscores between words (e.g., print_prompt). We have unscientifically observed that reaching for the underscore key exacerbates carpal tunnel syndrome (my pinky finger twinges just thinking about the underscore!) and hence prefer camelCase. But the most important thing is to be consistent in style within your organization.

The `main` function is always declared to return an `int`; this return value tells the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If `main` reaches the end without encountering a return statement, it will automatically return 0. Most operating systems do not automatically inform the user of the value returned by the program.

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

C.7.1 Conditional Statements

If, if/else, and switch/case statements are conditional statements commonly used in high-level languages, including C.

If Statements

An if statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
  statement
```

C Code Example eC.11 shows how to use an if statement in C. When the variable `aintBroke` is equal to 1, the variable `dontFix` is set to 1. A block of multiple statements can be executed by placing curly braces `{}` around the statements, as shown in C Code Example eC.12.

Curly braces, `{}`, are used to group zero or more statements into a *compound statement* or *block*.

C Code Example eC.11 IF STATEMENT

```
int dontFix = 0;
if (aintBroke == 1)
  dontFix = 1;
```

C Code Example eC.12 IF STATEMENT WITH A BLOCK OF STATEMENTS

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

If/else Statements

If/else statements execute one of two statements depending on a condition, as shown below. When the expression in the if statement is TRUE, `statement1` is executed. Otherwise, `statement2` is executed.

```
if (expression)
  statement1
else
  statement2
```

C Code Example eC.6(b) gives an example if/else statement in C. The code sets `y` equal to `a` if `a` is greater than `b`; otherwise `y` = `b`.

Switch/case Statements

Switch/case statements execute one of several statements depending on the conditions, as shown in the general format below.

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
  default: statement4
}
```

For example, if `variable` is equal to `expression2`, execution continues at `statement2` until the keyword `break` is reached, at which point it exits the switch/case statement. If no conditions are met, the `default` executes.

If the keyword `break` is omitted, execution begins at the point where the condition is TRUE and then falls through to execute the remaining cases below it. This is usually not what you want and is a common error among beginning C programmers.

C Code Example eC.13 shows a switch/case statement that, depending on the variable `option`, determines the amount of money `amt` to be disbursed. A switch/case statement is equivalent to a series of nested if/else statements, as shown by the equivalent code in C Code Example eC.14.

C Code Example eC.13 SWITCH/CASE STATEMENT

```
// Assign amt depending on the value of option
switch (option) {
  case 1:   amt = 100; break;
  case 2:   amt = 50;  break;
  case 3:   amt = 20;  break;
  case 4:   amt = 10;  break;
  default: printf("Error: unknown option.\n");
}
```

C Code Example eC.14 NESTED IF/ELSE STATEMENT

```
// Assign amt depending on the value of option
if      (option == 1) amt = 100;
else if (option == 2) amt = 50;
else if (option == 3) amt = 20;
else if (option == 4) amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2 Loops

While, do/while, and for loops are common loop constructs used in many high-level languages, including C. These loops repeatedly execute a statement as long as a condition is satisfied.

While Loops

While loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

```
while (condition)
  statement
```

The while loop in C Code Example eC.15 computes the factorial of 9 = 9 × 8 × 7 × … × 1. Note that the condition is checked before executing the `statement`. In this example, the statement is a compound statement or block, so curly braces are required.

C Code Example eC.15 WHILE LOOP

```
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;
// Multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

Do/while Loops

Do/while loops are like while loops but the condition is checked only after the `statement` is executed once. The general format is shown below. The `condition` is followed by a semi-colon.

```
do
  statement
while (condition);
```

The do/while loop in C Code Example eC.16 queries a user to guess a number. The program checks the condition (if the user's number is equal to the correct number) only after the body of the do/while loop executes once. This construct is useful when, as in this case, something must be done (e.g., the guess retrieved from the user) before the condition is checked. The `scanf` function, which is discussed in Section C.9.1, puts the value of the key pressed by a user into the variable `guess`.

C Code Example eC.16 DO/WHILE LOOP

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7

int guess, numGuesses = 0;
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);     // Read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

For Loops

For loops, like while and do/while loops, repeatedly execute a `statement` until a condition is not satisfied. However, for loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the for loop is

```
for (initialization; condition; loop operation)
  statement
```

The `initialization` code executes only once, before the for loop begins. The `condition` is tested at the beginning of each iteration of the loop. If the condition is not TRUE, the loop exits. The loop operation executes at the end of each iteration. C Code Example eC.17 shows the factorial of 9 computed using a for loop.

C Code Example eC.17 FOR LOOP

```
// Compute 9!
int i; // loop variable
int fact = 1;

for (i = 1; i < 10; i++)
  fact *= i;
```

Whereas the while and do/while loops in C Code Examples eC.15 and eC.16 include code for incrementing and checking the loop variable `i` and `numGuesses`, respectively, the for loop incorporates those statements into its format. A for loop could be expressed equivalently, but less conveniently, as

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

SUMMARY

- **Control-flow statements:** C provides control-flow statements for conditional statements and loops.
- **Conditional statements:** Conditional statements execute a statement when a condition is TRUE. C includes the following conditional statements: if, if/else, and switch/case.
- **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides while, do/while, and for loops.

C.8 MORE DATA TYPES

Beyond various sizes of integers and floating-point numbers, C includes other special data types, including pointers, arrays, strings, and structures. These data types are introduced in this section along with dynamic memory allocation.

C.8.1 Pointers

A pointer is the address of a variable. C Code Example eC.18 shows how to use pointers. `salary1` and `salary2` are variables that can contain integers, and `ptr` is a variable that can hold the address of an integer. The compiler will assign arbitrary locations in RAM for these variables, depending on the runtime environment. For the sake of concreteness, suppose this program is compiled on a 32-bit system with `salary1` at addresses 0x70–73, `salary2` at addresses 0x74–77, and `ptr` at 0x78–7B. Figure eC.3 shows memory and its contents after the program is executed.

In a variable declaration, a star (*) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the * operator *dereferences* a pointer, returning the value stored at

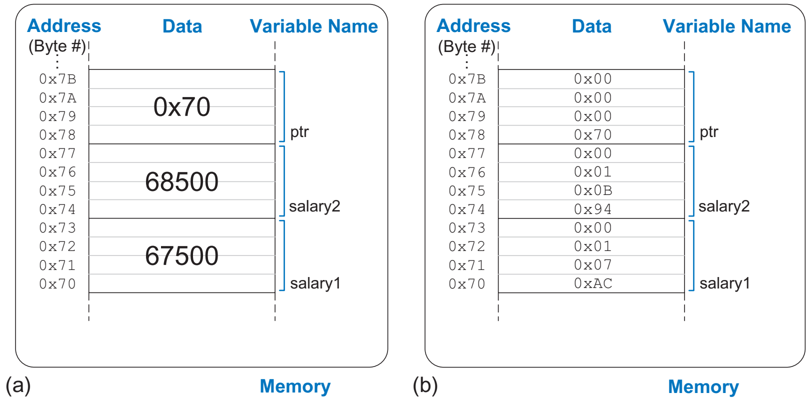


Figure eC.3 Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

the indicated memory address contained in the pointer. The & operator is pronounced "address of," and it produces the memory address of the variable being referenced.

C Code Example eC.18 POINTERS

```
// Example pointer manipulations
int salary1, salary2; // 32-bit numbers
int *ptr;             // a pointer specifying the address of an int variable

salary1 = 67500;      // salary1 = $67,500 = 0x000107AC
ptr = &salary1;       // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

Dereferencing a pointer to a nonexistent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

Pointers are particularly useful when a function needs to modify a variable instead of just returning a value. Because functions can't modify their inputs directly, a function can make the input a pointer to the variable. This is called passing an input variable *by reference* instead of *by value*, as shown in prior examples. C Code Example eC.19 gives an example of passing x by reference so that quadruple can modify the variable directly.

C Code Example eC.19 PASSING AN INPUT VARIABLE BY REFERENCE

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a) {

  *a = *a * 4;
}

int main(void) {
  int x = 5;
  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as NULL in a program.

C.8.2 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to $N - 1$, where N is the length of the array. C Code Example eC.20 declares an array variable called `scores` that holds the final exam scores for three students. Memory space is reserved for three 32-bit integers, that is, $3 \times 4 = 12$ bytes. Suppose the `scores` array starts at address 0x40. The address of the 0th element (i.e., `scores[0]`) is 0x40, the first element is 0x44, and the second element is 0x48, as shown in Figure eC.4. In C, the array variable, in this case `scores`, is a pointer to the 0th element; in other words, `scores` contains the address of the 0th array element. It is the programmer's responsibility not to access elements beyond the end of the array. C has no internal bounds checking, so a program that writes beyond the end of an array will compile fine but may stomp on other parts of memory when it runs.

Colloquially, the "0th" array element is also referred to as the "first" array element.

C Code Example eC.20 ARRAY DECLARATION

```
int32_t scores[3];    // array of three 4-byte numbers
```

The elements of an array can be initialized either at declaration using curly braces `{}`, as shown in C Code Example eC.21, or individually in the body of the code, as shown in C Code Example eC.22. Each element of an array is accessed using brackets `[]`. The contents of memory containing the array are shown in Figure eC.4. Array initialization using curly braces `{}` can be performed only at declaration and not afterward. for loops are commonly used to assign and read array data, as shown in C Code Example eC.23.

C Code Example eC.21 ARRAY INITIALIZATION AT DECLARATION USING `{}`

```
int32_t scores[3] = {93, 81, 97}; // scores[0] = 93; scores[1] = 81; scores[2] = 97;
```

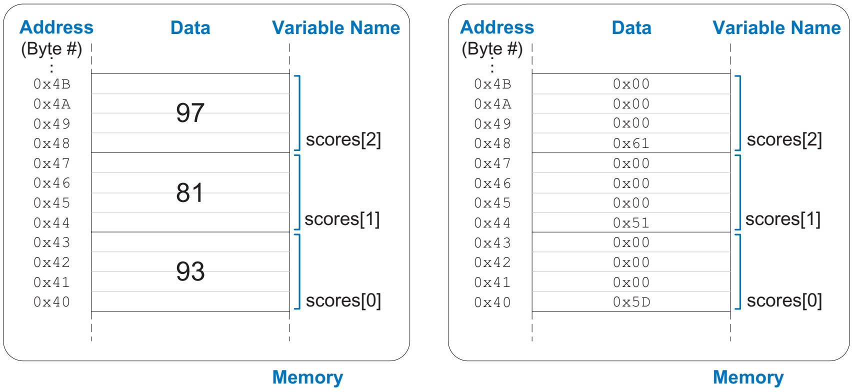


Figure eC.4 `scores` **array stored in memory**

C Code Example eC.22 ARRAY INITIALIZATION USING ASSIGNMENT

```
int32_t scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example eC.23 ARRAY INITIALIZATION USING A `for` LOOP

```
// User enters 3 student scores into an array
int32_t scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i = 0; i < 3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example eC.24 shows how an array is passed to a function. The input argument `arr` is simply the address of the 0th element of an array. Often, the number of elements in an array is also passed as an input argument. In a function, an input argument of type `int[]` indicates that it is an array of integers. Arrays of any type may be passed to a function.

C Code Example eC.24 PASSING AN ARRAY AS AN INPUT ARGUMENT

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;
  avg = getMean(data, 4);
  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, `getMean` could also have been declared as

```
float getMean(int *arr, int len);
```

Although functionally equivalent, `datatype[]` is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

A function is limited to a single output, that is, return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself. C Code Example eC.25 sorts an array from lowest to highest and leaves the result in the same array. The three function prototypes below are equivalent. The length of an array in a function declaration (i.e., `int vals[100]`) is ignored.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

C Code Example eC.25 PASSING AN ARRAY AND ITS LENGTH AS INPUTS

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len) {
  int i, j, temp;
  for (i = 0; i < len; i++) {
    for (j = i + 1; j < len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Arrays may have multiple dimensions. C Code Example eC.26 uses a two-dimensional array to store the grades across eight problem sets for ten students. Recall that initialization of array values using `{}` is allowed only at declaration.

C Code Example eC.26 TWO-DIMENSIONAL ARRAY INITIALIZATION

```
// Initialize 2D array at declaration
 int grades[10][8] = { {100, 107, 99,  101, 100, 104, 109, 117},
                       {103, 101, 94,  101, 102, 106, 105, 110},
                       {101, 102, 92,  101, 100, 107, 109, 110},
                       {114, 106, 95,  101, 100, 102, 102, 100},
                       {98,  105, 97,  101, 103, 104, 109, 109},
                       {105, 103, 99,  101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,  107, 98,  101, 109, 104, 110, 108} };
```

C Code Example eC.27 shows some functions that operate on the 2D grades array from C Code Example eC.26. Multidimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

C Code Example eC.27 OPERATING ON MULTIDIMENSIONAL ARRAYS

```
#include <stdio.h>

// Print the contents of a 10 × 8 array
void print2dArray(int arr[10][8]) {
  int i, j;

  for (i = 0; i < 10; i++) {        // for each of the 10 students
    printf("Row %d\n", i);
    for (j = 0; j < 8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}

// Calculate the mean score of a 10 × 8 array
float getMean(int arr[10][8]) {
  int i, j;
  float mean, total = 0;

  // get the mean value across a 2D array
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 8; j++) {
      total += arr[i][j];       // sum array values
    }
  }
  mean = total / (10 * 8);
  printf("Mean is: %f\n", mean);

  return mean;
}
```

Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators. Instead, you must use a loop to copy or compare each element one at a time.

C.8.3 Characters

A character (`char`) is an 8-bit variable. It can be viewed either as a two's complement number between −128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. For example, the letter A has the ASCII code 0x41, B = 0x42, etc. Thus "A" + 3 is 0x44, or "D." Table 6.2 lists the ASCII character encodings, and Table eC.4 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (`\r`), newline (`\n`), horizontal tab (`\t`), and the end of a string (`\0`). `\r` is shown for completeness but is rarely used in C programs. `\r` returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. `\n`, instead, moves the location of typing to the beginning of a new line.[3] The *NULL* character (`'\0'`) indicates the end of a text string and is discussed next in Section C.8.4.

C.8.4 Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code

Table eC.4 Special characters

Special Character	Hexadecimal Encoding	Description
`\r`	0x0D	carriage return
`\n`	0x0A	new line
`\t`	0x09	tab
`\0`	0x00	terminates a string
`\\`	0x5C	backslash
`\"`	0x22	double quote
`\'`	0x27	single quote
`\a`	0x07	bell

The term "carriage return" originates from typewriters that required the carriage, the contraption that holds the paper, to move to the right in order to allow typing to begin at the left side of the page. A carriage return lever, shown on the left in the figure below, is pressed so that the carriage would both move to the right and advance the paper by one line, called a line feed.

A Remington electric typewriter used by Winston Churchill. http://cwr.iwm.org.uk/server/show/conMediaFile.71979

[3] Windows text files use `\r\n` to represent end-of-line while UNIX-based systems use `\n`, which can cause nasty bugs when moving text files between systems.

for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the NULL terminator (ASCII value 0x00) at the end of the string.

C strings are called *null terminated* or *zero terminated* because the length is determined by looking for a zero at the end. In contrast, languages such as Pascal use the first byte to specify the string length, up to a maximum of 255 characters. This byte is called the *prefix byte* and such strings are called *P-strings*. An advantage of null-terminated strings is that the length can be arbitrarily great. An advantage of P-strings is that the length can be determined immediately without having to inspect all of the characters of the string.

C Code Example eC.28 shows the declaration of a 10-element character array called greeting that holds the string "Hello!" For concreteness, suppose greeting starts at memory address 0x50. Figure eC.5 shows the contents of memory from 0x50 to 0x59 holding the string "Hello!" Note that the string uses only the first seven elements of the array even though ten elements are allocated in memory.

C Code Example eC.28 STRING DECLARATION

```
char greeting[10] = "Hello!";
```

C Code Example eC.29 shows an alternate declaration of the string greeting. The pointer greeting holds the address of the first element of a 7-element array composed of each of the characters in "Hello!" followed by the null terminator. The code also demonstrates how to print strings by using the %s (string) format code.

C Code Example eC.29 ALTERNATE STRING DECLARATION

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =. Each element of the character array must be individually copied from the source string to the target string. This is true for any array. C Code Example eC.30 copies one string, src, to another, dst. The sizes of the arrays are not needed because the end of the src string is indicated by the null terminator. However, dst must be large enough so that you don't stomp on other data. strcpy and other string manipulation functions are available in C's built-in libraries (see Section C.9.4).

C Code Example eC.30 COPYING STRINGS

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src) {
  int i = 0;

  do {
    dst[i] = src[i];        // copy characters one byte at a time
  } while (src[i++]);       // until the null terminator is found
{
```

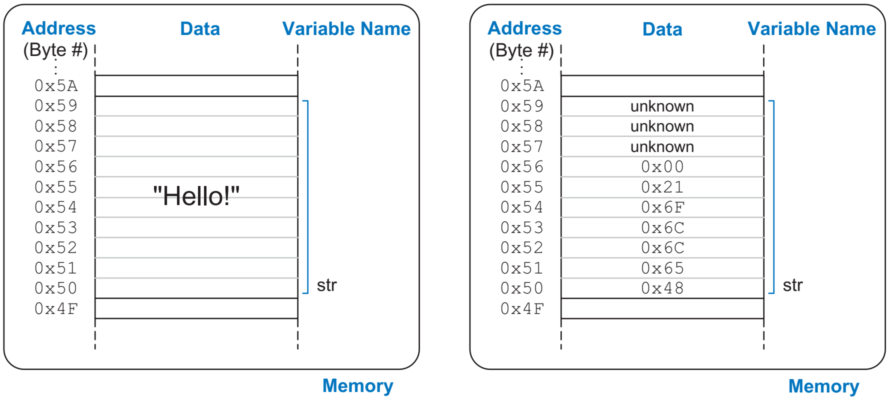


Figure eC.5 The string "Hello!" stored in memory

C.8.5 Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

struct is a keyword indicating that it is a structure; name is the structure tag name; and element1 and element2 are members of the structure. A structure may have any number of members. C Code Example eC.31 shows how to use a structure to store contact information. The program then declares a variable c1 of type struct contact.

C Code Example eC.31 STRUCTURE DECLARATION

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures. C Code Example eC.32 creates an array of contacts.

C Code Example eC.32 ARRAY OF STRUCTURES

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure. C Code Example eC.33 shows an example of declaring a pointer to a `struct contact`, assigning it to point to the 42nd element of `classlist` from C Code Example eC.32, and using the member access operator to set a value in that element.

C Code Example eC.33 ACCESSING STRUCTURE MEMBERS USING POINTERS AND ->

```
struct contact *cptr;
cptr = &classlist[42];
cptr -> height = 1.9; // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference. Passing by value requires the compiler to copy the entire structure into memory for the function to access. This can require a large amount of memory and time for a big structure. Passing by reference involves passing a pointer to the structure, which is more efficient. The function can also modify the structure being pointed to rather than having to return another structure. C Code Example eC.34 shows two versions of the `stretch` function that makes a contact 2 cm taller. `stretchByReference` avoids copying the large structure twice.

C Code Example eC.34 PASSING STRUCTURES BY VALUE OR BY NAME

```
struct contact stretchByValue(struct contact c) {
  c.height += 0.02;
  return c;
}
void stretchByReference(struct contact *cptr) {
  cptr -> height += 0.02;
}

int main(void) {

  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);    // and stretch some more
}
```

C.8.6 typedef

C also allows you to define your own names for data types using the typedef statement. For example, writing struct contact becomes tedious when it is used often, so we can define a new type named contact and use it as shown in C Code Example eC.35.

C Code Example eC.35 CREATING A CUSTOM TYPE USING typedef

```
typedef struct contact {
  char name[30];
  int phone;
  float height;  // in meters
} contact;       // defines contact as shorthand for "struct contact"

contact c1;      // now we can declare the variable as type contact
```

typedef can be used to create a new type occupying the same amount of memory as a primitive type. C Code Example eC.36 defines byte and bool as 8-bit types. The byte type may make it clearer that the purpose of pos is to be an 8-bit number rather than an ASCII character. The bool type indicates that the 8-bit number is representing TRUE or FALSE. These types make a program easier to read than if one simply used char everywhere.

C Code Example eC.36 typedef byte AND bool

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

C Code Example eC.37 illustrates defining a 3-element vector and a 3×3 matrix type using arrays.

C Code Example eC.37 typedef vector AND matrix

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 Dynamic Memory Allocation*

In all examples thus far, variables have been declared *statically*; that is, their size is known at compile time. This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The `malloc` function from `stdlib.h` allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example, the following code allocates ten 16-bit integers (10 × 2 = 20 bytes). The `sizeof` operator returns the size of a type or variable in bytes.

```
// Dynamically allocate 20 bytes of memory
int16_t *data = malloc(10 * sizeof(int16_t));
```

C Code Example eC.38 illustrates dynamic allocation and deallocation. The program accepts a variable number of inputs, stores them in a dynamically allocated array, and computes their average. The amount of memory necessary depends on the number of elements in the array and the size of each element. For example, if an `int` is a 4-byte variable and ten elements are needed, 40 bytes are dynamically allocated. The `free` function deallocates the memory so that it could later be used for other purposes. Failing to deallocate dynamically allocated memory is called a *memory leak* and should be avoided.

C Code Example eC.38 DYNAMIC MEMORY ALLOCATION AND DEALLOCATION

```
// Dynamically allocate and deallocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example eC.24.

int main(void) {
  int len, i;
  int *nums;

  printf("How many numbers would you like to enter? ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i = 0; i < 100; i++) {
      printf("Enter number: ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8 Linked Lists*

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*. Linked lists illustrate many of the concepts of structures, pointers, and dynamic memory allocation.

C Code Example eC.39 describes a linked list for storing computer user accounts to accommodate a variable number of users. Each user has a user name, password, unique user identification number (UID), and a field indicating whether they have administrator privileges. Each element of the list is of type `userL`, containing all of this user information along with a link to the next element in the list. A pointer to the head of the list is stored in a global variable called `users`, and is initially set to NULL to indicate that there are no users.

The program defines functions to insert, delete, and find a user and to count the number of users. The `insertUser` function allocates space for a new list element and adds it to the head of the list. The `deleteUser` function scans through the list until the specified UID is found and then removes that element, adjusting the link from the previous element to skip the deleted element and freeing the memory occupied by the deleted element. The `findUser` function scans through the list until the specified UID is found and returns a pointer to that element, or NULL if the UID is not found. The `numUsers` function counts the number of elements in the list.

C Code Example eC.39 LINKED LIST

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];       // user name
  char passwd[80];      // password
  int uid;              // user identification number
  int admin;            // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));   // create space for new user
  strcpy(newUser->uname, uname);     // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;             // insert at start of linked list
  users = newUser;
}
```

```
void deleteUser(int uid) {   // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;      // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

SUMMARY

- **Pointers:** A pointer holds the address of a variable.
- **Arrays:** An array is a list of similar elements declared using square brackets [].
- **Characters:** char types can hold small integers or special codes for representing text or symbols.
- **Strings:** A string is an array of characters ending with the null terminator 0x00.
- **Structures:** A structure stores a collection of related variables.
- **Dynamic memory allocation:** malloc is a built-in function in the standard library (stdlib.h) for allocating memory as the program runs. free deallocates the memory after use.
- **Linked Lists:** A linked list is a common data structure for storing a variable number of elements.

C.9 STANDARD LIBRARIES

Programmers commonly use a variety of standard functions, such as printing and trigonometric operations. To save each programmer from having to write these functions from scratch, C provides *libraries* of frequently used functions. Each library has a header file and an associated object file, which is a partially compiled C file. The header file holds variable declarations, defined types, and function prototypes. The object file contains the functions themselves and is linked at compile time to create the executable. Because the library function calls are already compiled into an object file, compile time is reduced. Table eC.5 lists some of the most frequently used C libraries, and each is described briefly below.

C.9.1 stdio

The standard input/output library `stdio.h` contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

```
#include <stdio.h>
```

Table eC.5 Frequently used C libraries

C Library Header File	Description
`stdio.h`	**Standard input/output library**. Includes functions for printing or reading to/from the screen or a file (`printf`, `fprintf` and `scanf`, `fscanf`) and to open and close files (`fopen` and `fclose`).
`stdlib.h`	**Standard library**. Includes functions for random number generation (`rand` and `srand`), for dynamically allocating or freeing memory (`malloc` and `free`), terminating the program early (`exit`), and for conversion between strings and numbers (`atoi`, `atol`, and `atof`).
`math.h`	**Math library**. Includes standard math functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor`, and `ceil`.
`string.h`	**String library**. Includes functions to compare, copy, concatenate, and determine the length of strings.

printf

The *print formatted* function printf displays text to the console. Its required input argument is a string enclosed in quotes "". The string contains text and optional commands to print variables. Variables to be printed are listed after the string and are printed using format codes shown in Table eC.6. C Code Example eC.40 gives a simple example of printf.

Table eC.6 printf format codes for printing variables

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating-point number (float or double)
%e	Floating-point number (float or double) in scientific notation (e.g., 1.56e7)
%c	Character (char)
%s	String (null-terminated array of characters)

C Code Example eC.40 PRINTING TO THE CONSOLE USING printf

```
// Simple print function
#include <stdio.h>

int num = 42;
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Console Output:

```
The answer is 42.
```

Floating-point formats (floats and doubles) default to printing six digits after the decimal point. To change the precision, replace %f with %w.df, where w is the minimum width of the number and d is the number of decimal places to print. Note that the decimal point is included in the width count. In C Code Example eC.41, pi is printed with a total of four characters, two of which are after the decimal point: 3.14. `e` is printed with a total of eight characters, three of which are after the decimal point. Because it has only one digit before the decimal point, it is padded with three leading spaces to reach the requested width. `c` should be printed with five characters, three of which are after the decimal point. But it is too wide to fit, so the requested width is overridden while retaining the three digits after the decimal point.

C Code Example eC.41 FLOATING POINT NUMBER FORMATS FOR PRINTING

```
// Print floating-point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne = %8.3f\nc = %5.3f\n", pi, e, c);
```

Console Output:

```
pi = 3.14
e  = 2.718
c  = 299800000.000
```

Because `%` and `\` are used in print formatting, to print these characters themselves, you must use the special character sequences shown in C Code Example eC.42.

C Code Example eC.42 PRINTING % AND \ USING `printf`

```
// How to print % and \ to the console
printf("Here are some special characters: %% \\ \n");
```

Console Output:

```
Here are some special characters: % \
```

`scanf`

The scanf function reads text typed on the keyboard. It uses format codes in the same way as `printf`. C Code Example eC.43 shows how to use `scanf`. When the `scanf` function is encountered, the program waits until the user types a value before continuing execution. The arguments to `scanf` are a string indicating one or more format codes and pointers to the variables where the results should be stored.

C Code Example eC.43 READING USER INPUT FROM THE KEYBOARD WITH `scanf`

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
int a;
  char str[80];
  float f;

  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating-point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);   // note no & needed: str is a pointer
}
```

File Manipulation

Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information. In C, the file must first be opened with the `fopen` function. It can then be read or written with `fscanf` or `fprintf` in a way analogous to reading and writing to the console. Finally, it should be closed with the `fclose` command.

The `fopen` function takes as arguments the file name and a *print mode*. It returns a *file pointer* of type `FILE*`. If `fopen` is unable to open the file, it returns NULL. This might happen when one tries to read a nonexistent file or write a file that is already opened by another program. The modes are:

`"w"`: Write to a file. If the file exists, it is overwritten.

`"r"`: Read from a file.

`"a"`: Append to the end of an existing file. If the file doesn't exist, it is created.

C Code Example eC.44 shows how to open, print to, and close a file. It is good practice to always check whether the file was opened successfully and to provide an error message if it was not. The `exit` function will be discussed in Section C.9.2. The `fprintf` function is like `printf` but it also takes the file pointer as an input argument to know which file to write. `fclose` closes the file, ensuring that all of the information is actually written to disk and freeing up file system resources.

C Code Example eC.44 PRINTING TO A FILE USING `fprintf`

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1); // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

It is idiomatic to open a file and check whether the file pointer is NULL in a single line of code, as shown in C Code Example eC.44. However, you could just as easily separate the functionality into two lines:

```
fptr=fopen("result.txt","w");
if (fptr == NULL)
 ...
```

C Code Example eC.45 illustrates reading numbers from a file named data.txt using `fscanf`. The file must first be opened for reading. The program then uses the `feof` function to check whether it has reached the end of the file. As long as the program is not at the end, it reads the next number and prints it to the screen. Again, the program closes the file at the end to free up resources.

C Code Example eC.45 READING INPUT FROM A FILE USING `fscanf`

```
#include <stdio.h>
int main(void) {
  FILE *fptr;
  int data;

  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) { // check that the end of the file hasn't been reached
    fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);

  }

  fclose(fptr);

}
```

data.txt

```
25 32 14 89
```

Console Output:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Other Handy stdio Functions

The sprintf function prints characters into a string, and sscanf reads variables from a string. The fgetc function reads a single character from a file, while fgets reads a complete line into a string.

fscanf is rather limited in its ability to read and parse complex files, so it is often easier to fgets one line at a time and then digest that line using sscanf or with a loop that inspects characters one at a time using fgetc.

C.9.2 stdlib

The standard library stdlib.h provides general-purpose functions, including random-number generation (rand and srand), dynamic memory allocation (malloc and free, already discussed in Section C.8.7), exiting the program early (exit), and number format conversions. To use these functions, add the following line at the top of the C file.

```
#include <stdlib.h>
```

rand and srand

rand returns a pseudo-random integer. Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the *seed*. To convert the number to a particular range, use the modulo operator (%) as shown in C Code Example eC.46 for a range of 0 to 9. The values x and y will be random but they will be the same each time this program runs. Sample console output is given below the code.

C Code Example eC.46 RANDOM NUMBER GENERATION USING rand

```
#include <stdlib.h>
int x, y;

x = rand();        // x = a random integer
y = rand() % 10;   // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

A programmer creates a different sequence of random numbers each time a program runs by changing the seed. This is done by calling the srand function, which takes the seed as its input argument. As shown in

C Code Example eC.47, the seed itself must be random, so a typical C program assigns it by calling the `time` function, which returns the current time in seconds.

For historical reasons, the `time` function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT). This date is just after the UNIX operating system was created by a group at Bell Labs, including Dennis Ritchie and Brian Kernighan, in 1969. Similar to New Year's Eve parties, some UNIX enthusiasts hold parties to celebrate significant values returned by `time`. For example, on February 1, 2009 at 23:31:30 UTC, `time` returned 1,234,567,890. In the year 2038, 32-bit UNIX clocks will overflow into the year 1901.

C Code Example eC.47 SEEDING THE RANDOM NUMBER GENERATOR USING `srand`

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>       // needed to call time()

int main(void) {

  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

`exit`

The `exit` function terminates a program early. It takes a single argument that is returned to the operating system to indicate the reason for termination. 0 indicates normal completion, while nonzero conveys an error condition.

Format Conversion: `atoi, atol, atof`

The standard library provides functions for converting ASCII strings to integers or doubles using `atoi` and `atof`, as shown in C Code Example eC.48. This is particularly useful when reading in mixed data (a mix of strings and numbers) from a file or when processing numeric command line arguments, as described in Section C.10.3.

C Code Example eC.48 FORMAT CONVERSION

```
// Convert ASCII strings to ints and floats
#include <stdlib.h>

int main(void) {
  int x;
  double z;

  x = atoi("42");
  z = atof("3.822");

  printf("x = %d\tz = %f\n", x, z);
}
```

Console Output:

```
x = 42     z = 3.822000
```

C.9.3 math

The math library `math.h` provides commonly used math functions such as trigonometry functions, square root, and logs. C Code Example eC.49 shows how to use some of these functions. To use math functions, place the following line in the C file:

```
#include <math.h>
```

C Code Example eC.49 MATH FUNCTIONS

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);         // a = 1, note: the input argument is in radians
  b = 2 * acos(0);    // b = pi (acos means arc cosine)
  c = sqrt(144);      // c = 12
  d = exp(2);         // d = e^2 = 7.389056
  e = log(7.389056);  // e = 2 (natural logarithm, base e)
  f = log10(1000);    // f = 3 (log base 10)
  g = floor(178.567); // g = 178, rounds to next lowest whole number
  h = pow(2, 10);     // h = 2^10 (i.e., 2 raised to the 10th power)

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
         a, b, c, d, e, f, g, h);
}
```

Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 string

The string library `string.h` provides commonly used string manipulation functions. Key functions include:

```
// Copy src into dst and return dst
char *strcpy(char *dst, char *src);

// Concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// Compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// Return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Although we have introduced relatively simple C programs, real-world programs can consist of tens or even thousands of C files to enable modularity, readability, and multiple programmers. This section describes how to compile a program spread across multiple C files and shows how to use compiler options and command line arguments.

C.10.1 Compiling Multiple C Source Files

Multiple C files are compiled into a single executable by listing all file names on the compile line as shown below. Remember that the group of C files still must contain only one `main` function, conventionally placed in a file named `main.c`.

```
gcc  main.c  file2.c  file3.c
```

C.10.2 Compiler Options

Compiler options allow the programmer to specify such things as output file names and formats, optimizations, etc. Compiler options are not standardized, but Table eC.7 lists ones that are commonly used. Each option is typically preceded by a dash (`-`) on the command line, as shown. For example, the "`-o`" option allows the programmer to specify an output file name other than the `a.out` default. A plethora of options exist; they can be viewed by typing `gcc --help` at the command line.

Table eC.7 Compiler options

Compiler Option	Description	Example
`-o outfile`	Specifies output file name	`gcc -o hello hello.c`
`-S`	Create assembly language output file (not executable)	`gcc -S hello.c` this produces `hello.s`
`-v`	Verbose mode—prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-Olevel`	Specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	`gcc -O3 hello.c`
`--version`	List the version of the compiler	`gcc -version`
`--help`	List all command line options	`gcc --help`
`-Wall`	Print all warnings	`gcc -Wall hello.c`

C.10.3 Command Line Arguments

Like other functions, `main` can also take input variables. However, unlike other functions, these arguments are specified at the command line. As shown in C Code Example eC.50, `argc` stands for *argument count*, and it denotes the number of arguments on the command line. `argv` stands for *argument vector*, and it is an array of the strings found on the command line. For example, suppose the program in C Code Example eC.50 is compiled into an executable called `testargs`. When the lines below are typed at the command line, `argc` has the value 4, and the array `argv` has the values `{"./testargs", "arg1", "25", "lastarg!"}`. Note that the executable name is counted as the 0th (i.e., left-most) argument. The console output after typing this command is shown below C Code Example eC.50.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Programs that need numeric arguments may convert the string arguments to numbers using the functions in `stdlib.h`.

C Code Example eC.50 COMMAND LINE ARGUMENTS

```
// Print command line arguments
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```

Console Output

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

C.11 COMMON MISTAKES

As with any programming language, you are almost certain to make errors while you write nontrivial C programs. Below are descriptions of some common mistakes made when programming in C. Some of these errors are particularly troubling because they compile but do not function as the programmer intended.

C Code Mistake eC.1 MISSING & IN `scanf`

Erroneous Code

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // missing & before a
```

Corrected Code

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

C Code Mistake eC.2 USING = INSTEAD OF == FOR COMPARISON

Erroneous Code

```
if (x = 1) // always evaluates as TRUE
  printf("Found!\n");
```

Corrected Code

```
if (x == 1)
  printf("Found!\n");
```

C Code Mistake eC.3 INDEXING PAST LAST ELEMENT OF ARRAY

Erroneous Code

```
int array[10];
array[10] = 42;     // index is 0-9
```

Corrected Code

```
int array[10];
array[9] = 42;
```

C Code Mistake eC.4 USING = IN `#define` STATEMENT

Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

Corrected Code

```
#define NUM 4
```

C Code Mistake eC.5 USING AN UNINITIALIZED VARIABLE

Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
  ...
```

Corrected Code

```
int i = 10;
if (i == 10)
  ...
```

C Code Mistake eC.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

Erroneous Code

```
#include "myfile.h"
```

Corrected Code

```
#include "othercode\myfile.h"
```

Debugging skills are acquired with practice, but here are a few hints.

- Fix bugs starting with the first error indicated by the compiler. Later errors may be downstream effects of this error. After fixing that bug, recompile and repeat until all bugs (at least those caught by the compiler!) are fixed.
- When the compiler says a valid line of code is in error, check the code above it (i.e., for missing semicolons or braces).
- When needed, split up complicated statements into multiple lines.
- Use `printf` to output intermediate results.
- When a result doesn't match expectations, start debugging the code at the *first* place it deviates from expectations.
- Look at all compiler warnings. While some warnings can be ignored, others may alert you to more subtle code errors that will compile but not run as intended.

C Code Mistake eC.7 USING LOGICAL OPERATORS (`!`, `||`, `&&`) INSTEAD OF BITWISE (`~`, `|`, `&`)

Erroneous Code

```
char x = !5;     // logical NOT:  x = 0
char y = 5||2;   // logical OR:   y = 1
char z = 5&&2;   // logical AND:  z = 1
```

Corrected Code

```
char x = ~5;  // bitwise NOT:  x = 0b11111010
char y = 5|2; // bitwise OR:   y = 0b00000111
char z = 5&2; // logical AND:  z = 0b00000000
```

C Code Mistake eC.8 FORGETTING `break` IN A `switch/case` STATEMENT

Erroneous Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default:  direction = 0;
}
// direction = 0
```

Corrected Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default:  direction = 0;
}
// direction = 2
```

C Code Mistake eC.9 MISSING CURLY BRACES `{}`

Erroneous Code

```
if (ptr == NULL) // missing curly braces
  printf("Unable to open file.\n");
  exit(1);       // always executes
```

Corrected Code

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

C Code Mistake eC.10 USING A FUNCTION BEFORE IT IS DECLARED

Erroneous Code

```
int main(void) {
  test();
}

void test(void) {
 ...
}
```

Corrected Code

```
void test(void) {
  ...
}
int main(void) {
  test();
}
```

C Code Mistake eC.11 DECLARING A LOCAL AND GLOBAL VARIABLE WITH THE SAME NAME

Erroneous Code

```
int x = 5;    // global declaration of x
int test(void) {
  int x = 3;  // local declaration of x
  ...
}
```

Corrected Code

```
int x = 5;    // global declaration of x
int test(void) {
  int y = 3;  // local variable is y
  ...
}
```

C Code Mistake eC.12 TRYING TO INITIALIZE AN ARRAY WITH {} AFTER DECLARATION

Erroneous Code

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

Corrected Code

```
int scores[3] = {93, 81, 97};
```

C Code Mistake eC.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

Erroneous Code

```
int scores[3] = {88, 79, 93};
int scores2[3];
scores2 = scores;
```

Corrected Code

```
int scores[3] = {88, 79, 93};
int scores2[3];
for (i = 0; i < 3; i++)
  scores2[i] = scores[i];
```

C Code Mistake eC.14 FORGETTING THE SEMI-COLON AFTER A `do/while` LOOP

Erroneous Code

```
int num;
do {
  num = getNum();
} while (num < 100) // missing ;
```

Corrected Code

```
int num;
do {
  num = getNum();
} while (num < 100);
```

C Code Mistake eC.15 USING COMMAS INSTEAD OF SEMICOLONS IN `for` LOOP

Erroneous Code

```
for (i=0, i < 200, i++)
  ...
```

Corrected Code

```
for (i = 0; i < 200; i++)
  ...
```

C Code Mistake eC.16 INTEGER DIVISION INSTEAD OF FLOATING-POINT DIVISION

Erroneous Code

```
// integer (truncated) division occurs when
// both arguments of division are integers
float x = 9 / 4; // x = 2.0
```

Corrected Code

```
// at least one of the arguments of
// division must be a float to
// perform floating-point division
float x = 9.0 / 4; // x = 2.25
```

C Code Mistake eC.17 WRITING TO AN UNINITIALIZED POINTER

Erroneous Code

```
int *y = 77;
```

Corrected Code

```
int x, *y = &x;
*y = 77;
```

C Code Mistake eC.18 GREAT EXPECTATIONS (OR LACK THEREOF)

A common beginner error is to write an entire program (usually with little modularity) and expect it to work perfectly the first time. For nontrivial programs, writing modular code and testing the individual functions along the way are essential. Debugging becomes exponentially harder and more time-consuming with complexity.

Another common error is lacking expectations. When this happens, the programmer can only verify that the code produces a result, not that the result is correct. Testing a program with known inputs and expected results is critical in verifying functionality.

This appendix has focused on using C on a system such as a personal computer. Chapter 9 (available as a web supplement) describes how C is used to program SparkFun's RED-V RedBoard—based on a RISC-V microcontroller—that can be used in embedded systems. Microcontrollers are usually programmed in C because the language provides nearly as much low-level control of the hardware as assembly language, yet is much more succinct and faster to write.

Further Reading

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

The fascinating biography of Robert Noyce, an inventor of the microchip and founder of Fairchild and Intel. For anyone thinking of working in Silicon Valley, this book gives insights into the culture of the region, a culture influenced more heavily by Noyce than any other individual.

Ciletti M., *Advanced Digital Design with the Verilog HDL, 2nd ed.*, Prentice Hall, 2010.

A good reference for Verilog 2005 (but not SystemVerilog).

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips,* Wiley, 2005.

An insider's tale of the development of several generations of Intel's Pentium chips, told by one of the leaders of the project. For those considering a career in the field, this book offers views into the management of huge design projects and a behind-the-scenes look at one of the most significant commercial microprocessor lines.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

The most complete text on computer arithmetic systems. An excellent resource for building high-quality arithmetic units for computers.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach*, *6th ed.*, Morgan Kaufmann, 2017.

The authoritative text on advanced computer architecture. If you are intrigued about the inner workings of cutting-edge microprocessors, this is the book for you.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

A classic story of the design of a computer system. Three decades later, the story is still a page-turner and the insights on project management and technology still ring true.

Patterson D., and Waterman A., *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon, 2017.

A concise introduction to the RISC-V architecture by two of the RISC-V architects.

Pedroni V., *Circuit Design and Simulation with VHDL, 2nd ed.*, MIT Press, 2010.

A reference showing how to design circuits with VHDL.

SystemVerilog IEEE Standard (IEEE STD 1800).

The IEEE standard for the SystemVerilog Hardware Description Language; last updated in 2019. Available at *ieeexplore.ieee.org*.

VHDL IEEE Standard (IEEE STD 1076).

The IEEE standard for VHDL; last updated in 2017. Available at *ieeexplore.ieee.org*.

Wakerly J., *Digital Design: Principles and Practices*, *5th ed.*, Pearson, 2018.

A comprehensive and readable text on digital design, and an excellent reference book.

Weste N., and Harris D., *CMOS VLSI Design*, *4th ed.*, Addison-Wesley, 2010.

Very Large Scale Integration (VLSI) Design is the art and science of building chips containing oodles of transistors. This book, coauthored by one of our favorite writers, spans the field from the beginning through the most advanced techniques used in commercial products.

Index

Note: Page numbers followed by "*f*," "*t*," and "*b*" refer to figures, tables, and boxes, respectively; page numbers preceded by "e" refer to online material.

RISC-V Instruction Set Summary

31:25		24:20	19:15	14:12	11:7	6:0	
funct7		rs2	rs1	funct3	rd	op	R-Type
imm$_{11:0}$			rs1	funct3	rd	op	I-Type
imm$_{11:5}$		rs2	rs1	funct3	imm$_{4:0}$	op	S-Type
imm$_{12,10:5}$		rs2	rs1	funct3	imm$_{4:1,11}$	op	B-Type
imm$_{31:12}$					rd	op	U-Type
imm$_{20,10:1,11,19:12}$					rd	op	J-Type
fs3	funct2	fs2	fs1	funct3	fd	op	R4-Type
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure B.1 RISC-V 32-bit instruction formats

- `imm`: signed immediate in **imm$_{11:0}$**
- `uimm`: 5-bit unsigned immediate in **imm$_{4:0}$**
- `upimm`: 20 upper bits of a 32-bit immediate, in **imm$_{31:12}$**
- `Address`: memory address: **rs1** + SignExt(**imm$_{11:0}$**)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({**imm$_{12:1}$**, 1'b0})
- `JTA`: jump target address: PC + SignExt({**imm$_{20:1}$**, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$= rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$= rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 — rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

*Encoded in instr$_{31:25}$, the upper seven bits of the immediate field

Table B.2 RV64I: Extra integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	011	–	I	ld rd, imm(rs1)	load double word	rd = [Address]$_{63:0}$
0000011 (3)	110	–	I	lwu rd, imm(rs1)	load word unsigned	rd = ZeroExt([Address]$_{31:0}$)
0011011 (27)	000	–	I	addiw rd, rs1, imm	add immediate word	rd = SignExt((rs1 + SignExt(imm))$_{31:0}$)
0011011 (27)	001	0000000	I	slliw rd, rs1, uimm	shift left logical immediate word	rd = SignExt((rs1$_{31:0}$ << uimm)$_{31:0}$)
0011011 (27)	101	0000000	I	srliw rd, rs1, uimm	shift right logical immediate word	rd = SignExt((rs1$_{31:0}$ >> uimm)$_{31:0}$)
0011011 (27)	101	0100000	I	sraiw rd, rs1, uimm	shift right arith. immediate word	rd = SignExt((rs1$_{31:0}$ >>> uimm)$_{31:0}$)
0100011 (35)	011	–	S	sd rs2, imm(rs1)	store double word	[Address]$_{63:0}$ = rs2
0111011 (59)	000	0000000	R	addw rd, rs1, rs2	add word	rd = SignExt((rs1 + rs2)$_{31:0}$)
0111011 (59)	000	0100000	R	subw rd, rs1, rs2	subtract word	rd = SignExt((rs1 − rs2)$_{31:0}$)
0111011 (59)	001	0000000	R	sllw rd, rs1, rs2	shift left logical word	rd = SignExt((rs1$_{31:0}$ << rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0000000	R	srlw rd, rs1, rs2	shift right logical word	rd = SignExt((rs1$_{31:0}$ >> rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0100000	R	sraw rd, rs1, rs2	shift right arithmetic word	rd = SignExt((rs1$_{31:0}$ >>> rs2$_{4:0}$)$_{31:0}$)

In RV64I, registers are 64 bits, but instructions are still 32 bits. The term “word” generally refers to a 32-bit value. In RV64I, immediate shift instructions use 6-bit immediates: uimm$_{5:0}$; but for word shifts, the most significant bit of the shift amount (uimm$_5$) must be 0. Instructions ending in “w” (for “word”) operate on the lower half of the 64-bit registers. Sign- or zero-extension produces a 64-bit result.

Table B.3 RVF/D: RISC-V single- and double-precision floating-point instructions

op	funct3	funct7	rs2	Type	Instruction	Description	Operation
1000011 (67)	rm	fs3, fmt	–	R4	fmadd fd,fs1,fs2,fs3	multiply-add	fd = fs1 * fs2 + fs3
1000111 (71)	rm	fs3, fmt	–	R4	fmsub fd,fs1,fs2,fs3	multiply-subtract	fd = fs1 * fs2 − fs3
1001011 (75)	rm	fs3, fmt	–	R4	fnmsub fd,fs1,fs2,fs3	negate multiply-add	fd = −(fs1 * fs2 + fs3)
1001111 (79)	rm	fs3, fmt	–	R4	fnmadd fd,fs1,fs2,fs3	negate multiply-subtract	fd = −(fs1 * fs2 - fs3)
1010011 (83)	rm	00000, fmt	–	R	fadd fd,fs1,fs2	add	fd = fs1 + fs2
1010011 (83)	rm	00001, fmt	–	R	fsub fd,fs1,fs2	subtract	fd = fs1 − fs2
1010011 (83)	rm	00010, fmt	–	R	fmul fd,fs1,fs2	multiply	fd = fs1 * fs2
1010011 (83)	rm	00011, fmt	–	R	fdiv fd,fs1,fs2	divide	fd = fs1 / fs2
1010011 (83)	rm	01011, fmt	00000	R	fsqrt fd,fs1	square root	fd = sqrt(fs1)
1010011 (83)	000	00100, fmt	–	R	fsgnj fd,fs1,fs2	sign injection	fd = fs1, sign = sign(fs2)
1010011 (83)	001	00100, fmt	–	R	fsgnjn fd,fs1,fs2	negate sign injection	fd = fs1, sign = −sign(fs2)
1010011 (83)	010	00100, fmt	–	R	fsgnjx fd,fs1,fs2	xor sign injection	fd = fs1, sign = sign(fs2) ^ sign(fs1)
1010011 (83)	000	00101, fmt	–	R	fmin fd,fs1,fs2	min	fd = min(fs1, fs2)
1010011 (83)	001	00101, fmt	–	R	fmax fd,fs1,fs2	max	fd = max(fs1, fs2)
1010011 (83)	010	10100, fmt	–	R	feq rd,fs1,fs2	compare =	rd = (fs1 == fs2)
1010011 (83)	001	10100, fmt	–	R	flt rd,fs1,fs2	compare <	rd = (fs1 < fs2)
1010011 (83)	000	10100, fmt	–	R	fle rd,fs1,fs2	compare ≤	rd = (fs1 ≤ fs2)
1010011 (83)	001	11100, fmt	00000	R	fclass rd,fs1	classify	rd = classification of fs1
RVF only							
0000111 (7)	010	–	–	I	flw fd, imm(rs1)	load float	fd = [Address]$_{31:0}$
0100111 (39)	010	–	–	S	fsw fs2,imm(rs1)	store float	[Address]$_{31:0}$ = fd
1010011 (83)	rm	1100000	00000	R	fcvt.w.s rd, fs1	convert to integer	rd = integer(fs1)
1010011 (83)	rm	1100000	00001	R	fcvt.wu.s rd, fs1	convert to unsigned integer	rd = unsigned(fs1)
1010011 (83)	rm	1101000	00000	R	fcvt.s.w fd, rs1	convert int to float	fd = float(rs1)
1010011 (83)	rm	1101000	00001	R	fcvt.s.wu fd, rs1	convert unsigned to float	fd = float(rs1)
1010011 (83)	000	1110000	00000	R	fmv.x.w rd, fs1	move to integer register	rd = fs1
1010011 (83)	000	1111000	00000	R	fmv.w.x fd, rs1	move to f.p. register	fd = rs1
RVD only							
0000111 (7)	011	-	–	I	fld fd, imm(rs1)	load double	fd = [Address]$_{63:0}$
0100111 (39)	011	–	–	S	fsd fs2,imm(rs1)	store double	[Address]$_{63:0}$ = fd
1010011 (83)	rm	1100001	00000	R	fcvt.w.d rd, fs1	convert to integer	rd = integer(fs1)
1010011 (83)	rm	1100001	00001	R	fcvt.wu.d rd, fs1	convert to unsigned integer	rd = unsigned(fs1)
1010011 (83)	rm	1101001	00000	R	fcvt.d.w fd, rs1	convert int to double	fd = double(rs1)
1010011 (83)	rm	1101001	00001	R	fcvt.d.wu fd, rs1	convert unsigned to double	fd = double(rs1)
1010011 (83)	rm	0100000	00001	R	fcvt.s.d fd, fs1	convert double to float	fd = float(fs1)
1010011 (83)	rm	0100001	00000	R	fcvt.d.s fd, fs1	convert float to double	fd = double(fs1)

fs1, fs2, fs3, fd: floating-point registers. **fs1**, **fs2**, and **fd** are encoded in fields **rs1**, **rs2**, and **rd**; only R4-type also encodes **fs3**. **fmt**: precision of computational instruction (single=00_2, double=01_2, quad=11_2). **rm**: rounding mode (0=to nearest, 1=toward zero, 2=down, 3=up, 4=to nearest (max magnitude), 7=dynamic). sign(**fs1**): the sign of **fs1**.