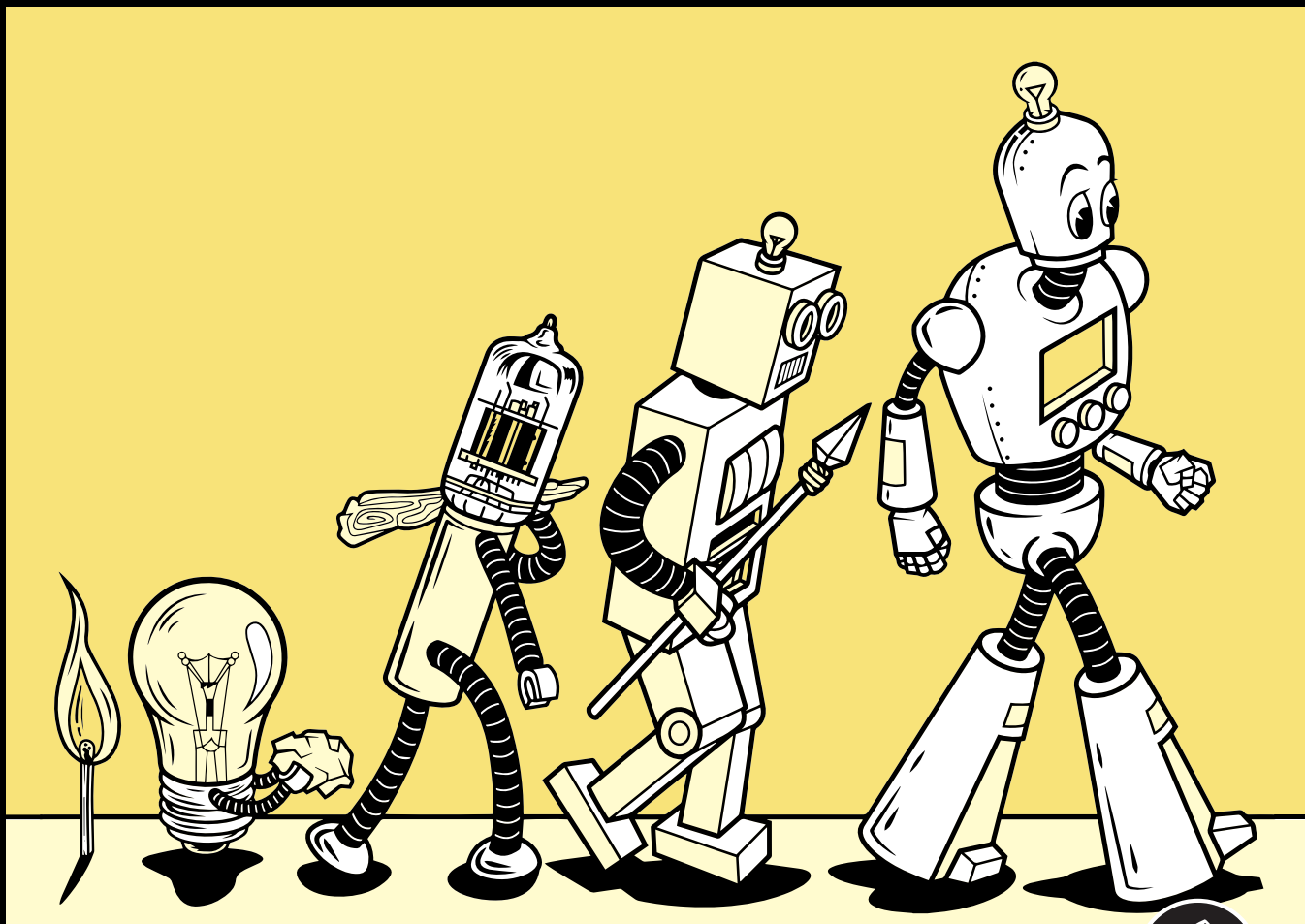
COMPUTER ARCHITECTURE

FROM THE STONE AGE TO THE QUANTUM AGE

CHARLES FOX



no starch press®

COMPUTER ARCHITECTURE

COMPUTER ARCHITECTURE

From the Stone Age to the Quantum Age

by Charles Fox



no starch press®

San Francisco

COMPUTER ARCHITECTURE. Copyright © 2024 by Charles Fox.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

28 27 26 25 24 1 2 3 4 5

ISBN-13: 978-1-7185-0286-4 (print)
ISBN-13: 978-1-7185-0287-1 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Jennifer Kepler
Developmental Editors: Nathan Heidelberger and Alex Freed
Cover Illustrator: Joshua Kemble
Interior Design: Octopod Studios
Technical Reviewer: Andrew Bower
Copyeditor: Gary Smith
Proofreader: Scout Festa

Library of Congress Control Number: 2023042109

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Can you build an emerald city with these grains of sand?

—Jim Steinman

About the Author

Dr. Charles Fox is a senior lecturer teaching computer architecture at the University of Lincoln, which holds a TEF Gold rating for teaching quality, the highest available in the UK. He has received teaching awards including the university's Teaching Excellence Award, a Programme Leader award, and shortlistings from Student Unions for Inspirational Teacher of the Year and PhD Supervisor of the Year. He's a Fellow of the Higher Education Academy (FHEA) and author of the textbook *Data Science for Transport* (Springer, 2018). He learned to program on the BBC Micro at the age of eight, then studied computer science at Cambridge with several of its designers. He has an MSc in informatics from the University of Edinburgh and a DPhil in information engineering from the University of Oxford. He has worked as a hedge fund quant, data science consultant, and venture capital investment advisor, and he has published over 100 papers, which often apply neural, embedded, parallel, and smart architectures in AI and robotics. He hated architecture as a student but later found it to be quite useful. His teaching style assumes students may feel similarly, so he introduces students to new concepts by using history, games, music, retro computing, hacking, and making.

About the Technical Reviewer

Andrew Bower works at Advanced Micro Devices Inc. Following a misspent youth programming a succession of 1980s and 1990s computers at a low level, he earned a degree in computer science from the University of Cambridge and went to work in the semiconductor industry. He has mostly developed embedded software and development or debugging tools for systems on chip software for Broadcom Inc., Xilinx Inc., and semiconductor startups. Andrew is a Chartered Engineer and holds an MSc in engineering management from the University of Leeds, but you can't keep him away from the C compiler. While reviewing this book, Andrew got a bit carried away creating an open source developer toolchain for and software emulation of the Manchester Baby to verify the LogiSim model, which are available at *https://github.com/andy-bower*.

BRIEF CONTENTS

Introduction xxi

PART I: FUNDAMENTAL CONCEPTS

Chapter 1: Historical Architectures 3
Chapter 2: Data Representation 45
Chapter 3: Basic CPU-Based Architecture 73

PART II: THE ELECTRONIC HIERARCHY

Chapter 4: Switches 93
Chapter 5: Digital Logic 113
Chapter 6: Simple Machines 135
Chapter 7: Digital CPU Design 155
Chapter 8: Advanced CPU Design 181
Chapter 9: Input/Output 201
Chapter 10: Memory 215

PART III: EXAMPLE ARCHITECTURES

Chapter 11: Retro Architectures 245
Chapter 12: Embedded Architectures 281
Chapter 13: Desktop Architectures 301
Chapter 14: Smart Architectures 339
Chapter 15: Parallel Architectures 355
Chapter 16: Future Architectures 397

Appendix: Operating System Support 425
Acknowledgments 437
Figure Credits 439
Index 447

CONTENTS IN DETAIL

INTRODUCTION **xxi**

Who Is This Book For? xxii
Why Computer Architecture? xxii
Changes in the Field xxiii
How to Use This Book xxiv
Inside Common Devices xxvi
Desktop PC xxvi
Laptop xxviii
Smartphone xxix
Washing Machine xxxi
Book Overview xxxii
Exercises xxxiv
Further Reading xxxv

PART I
FUNDAMENTAL CONCEPTS

1
HISTORICAL ARCHITECTURES **3**

What Is a Computer? 4
Before the Industrial Revolution 6
The Stone Age 6
The Bronze Age 7
The Iron Age 9
The Islamic Golden Age 12
The Renaissance and Enlightenment 13
The Steam Age 14
The Jacquard Loom 15
Victorian Barrel Organs and Music Boxes 15
Babbage's Difference Engine 16
Babbage's Analytical Engine 17
Mechanical Differential Analyzers 20
The Diesel Age 22
The IBM Hollerith Tabulating Machine 22
Electromechanical Differential Analyzers 23
Electromechanical Machines of World War II 24
The Zuse Z3 26

The Electrical Age 27
Pure Electronic Cryptology of World War II 27
ENIAC 28
Virtual Machine ENIAC 29
The Manchester Baby 30
The 1950s and Commercial Computing 31
The Transistor Age 32
The 1960s and Big Transistors 32
The 1970s and Integrated Circuits 34
The 1980s Golden Age 35
The Bland 1990s 36
The 2000s and Reconnecting the Community 37
The 2010s and the End of Moore's Law 38
The 2020s, the Cloud, and the Internet of Things 40
So Who Invented the Computer? 41
Summary 42
Exercises 42
Further Reading 43

2
DATA REPRESENTATION **45**

A Brief History of Data Representations 46
Tally Sticks and Trading Tokens 46
Roman Numerals 47
Split Tallies 48
Arabic and Other Numerals 48
Modern Number Systems 50
Bases and Exponents 50
Base 10: Decimal 51
Base 2: Binary 51
Base 1,000 52
Base 60: Sexagesimal 52
Base 16: Hexadecimal 53
Base 256: Bytes 55
How to Convert Between Bases 55
Representing Data 56
Natural Numbers 56
Integers 59
Rationals 60
Fixed Point 60
Floating Point 61
Arrays 62
Text 63

Multimedia Data Representation 67
Data Structures 69
Measuring Data 70
Summary 71
Exercises 72
Further Reading 72

3
BASIC CPU-BASED ARCHITECTURE **73**

A Musical Processing Unit 74
From Music to Calculation 76
From Calculation to Computation 76
Babbage's Central Processing Unit 77
High-Level Architecture 78
Programmer Interface 79
Internal Subcomponents 82
Internal Operation 85
Summary 88
Exercises 88
Further Reading 89

PART II
THE ELECTRONIC HIERARCHY

4
SWITCHES **93**

Directional Systems 94
Water Valve 94
Heat Diode 95
p-n Junction Diode 96
Switching 101
Water Current Switch 101
Electrical Tube Switch 102
p-n-p Transistor 103
Water Pressure Effect Switch 105
Field-Effect Transistors 106
Clocks 107
Fabricating Transistors 108
Moore's Law 109
Summary 110
Exercises 111
Further Reading 111

5
DIGITAL LOGIC **113**

Claude Shannon and Logic Gates 114
Logic Gates 115
Identifying Universal Gates 117
Making Logic Gates from Transistors 117
Putting Logic Gates on Chips 120
Boolean Logic 122
Logic as Arithmetic 123
Model Checking vs. Proof 124
Simplifying Logic Circuits Using Boolean Logic 127
Laying Out Digital Logic 128
7400 Chips 128
Photolithography 129
Programmable Logic Arrays 130
Field Programmable Gate Arrays 131
Summary 132
Exercises 133
Further Reading 134

6
SIMPLE MACHINES **135**

Combinatorial Logic 136
Bitwise Logical Operations 136
Multi-input Logical Operations 136
Shifters 137
Decoders and Encoders 138
Multiplexers and Demultiplexers 139
Adders 140
Negators and Subtractors 143
From Combinatorial to Sequential Logic 144
Clocked Logic 147
Clocked Flip-Flops 147
Counters 148
Sequencers 149
Random-Access Memory 150
Summary 152
Exercises 153
Further Reading 154

7
DIGITAL CPU DESIGN **155**

The Baby's Programmer Interface ... 156
Halting ... 156
Constants ... 157
Load and Store ... 157
Arithmetic ... 158
Jumps ... 158
Branches ... 159
Assemblers ... 159
The Baby's Internal Structures ... 163
Registers ... 164
Arithmetic Logic Unit ... 166
Control Unit ... 167
Putting It All Together ... 170
Fetch ... 170
Decode ... 171
Execute ... 172
Complete Baby Implementation ... 176
Summary ... 178
Exercises ... 179
Further Reading ... 180

8
ADVANCED CPU DESIGN **181**

Number of User Registers ... 181
Number of Instructions ... 183
Duration of Instructions ... 183
Different Addressing Modes ... 184
Subroutines ... 186
Stackless Architectures ... 187
Stack Architectures ... 187
Floating-Point Units ... 189
Pipelining ... 190
Hazards ... 192
Hazard Correction ... 193
Out-of-Order Execution ... 196
Hyperthreading ... 198
Summary ... 199
Exercises ... 199
Further Reading ... 200

9
INPUT/OUTPUT **201**

Basic I/O Concepts ... 201
Buses ... 203
    Bus Lines ... 204
    The CPU-Bus Interface ... 204
I/O Modules ... 206
    Control and Timing ... 207
    Error Detection ... 209
I/O Module Techniques ... 209
    Polling ... 209
    Interrupts ... 210
    Direct Memory Access ... 211
I/O Without Modules ... 212
    CPU I/O Pins ... 212
    Memory Mapping ... 212
    Bus Hierarchies ... 212
Summary ... 213
Exercises ... 213
Further Reading ... 214

10
MEMORY **215**

The Memory Hierarchy ... 215
Primary Memory ... 217
    Bytes and Endianness ... 217
    Memory Modules ... 219
    Random-Access Memory ... 219
    Read-Only Memory ... 225
Caches ... 226
    Cache Concepts ... 227
    Cache Read Policies ... 229
    Cache Write Policies ... 231
    Advanced Cache Architectures ... 232
Secondary and Offline Memory ... 233
    Tapes ... 234
    Disks ... 236
    Solid-State Drives ... 240
Tertiary Memory ... 240
Data Centers ... 241
Summary ... 241
Exercises ... 241
Further Reading ... 242

PART III
EXAMPLE ARCHITECTURES

11
RETRO ARCHITECTURES 245

Programming in the 1980s Golden Age 246
8-Bit Era 248
16-Bit Era 249
Working with the MOS 6502 8-Bit CPU 250
Internal Subcomponents 250
Programmer Interface 255
8-Bit Computer Design with the Commodore 64 260
Understanding the Architecture 260
Programming the C64 263
Working with the Motorola 68000 16-Bit CPU 266
Internal Subcomponents 266
Programmer Interface 267
16-Bit Computer Design with the Commodore Amiga 269
Retro Peripherals 275
Cathode Ray Tube Displays 275
User Input 276
Serial Port 276
MIDI Interfaces 277
Summary 278
Exercises 278
Further Reading 280

12
EMBEDDED ARCHITECTURES 281

Design Principles 282
Single Purpose 282
Reliability 282
Mobility and Power 282
Encapsulation 283
Careful Debugging 283
Microcontrollers 283
CPU 283
Memory 284
Timers and Counters 284
Embedded I/O 284
Analog-Digital Conversion 285
Embedded Serial Ports 286
Inter-Integrated Circuit Bus 286
Controller Area Network Bus 287

Arduino 288
The ATmega328 Microcontroller 289
The Rest of the Arduino Board 291
Programming Arduino 291
Other CPU-Based Embedded Systems 293
Atmel AVR Without the Arduino 293
PIC Microcontrollers 293
Digital Signal Processors 294
Embedded Systems with No CPU 295
Programmable Logic Controllers 295
Embedded FPGAs 297
Summary 298
Exercises 299
Further Reading 299

13
DESKTOP ARCHITECTURES 301

CISC Design Philosophy 302
Microprogramming 303
x86 History 304
Prehistory 304
16-Bit Classical Era 305
32-Bit Clone Wars Era 306
64-Bit Branding Era 307
Programming x86 308
Registers 309
Netwide Assembler Syntax 311
Segmentation 317
Backward-Compatible Modes 318
PC Computer Design 318
The Bus Hierarchy 319
Common Buses 322
Standard Devices 324
Summary 330
Exercises 331
Further Reading 336

14
SMART ARCHITECTURES 339

Smart Devices 340
RISC Philosophy 341
RISC-V 343
Understanding the Architecture 343
Programming Core RISC-V 345
Extending RISC-V 347

Different RISC-V Implementations . . . 348
RISC-V Toolchain and Community . . . 349
Smart Computer Design . . . 349
Low-Power DRAM . . . 350
Cameras . . . 351
Touchscreens . . . 351
Summary . . . 352
Exercises . . . 352
Further Reading . . . 354

15
PARALLEL ARCHITECTURES **355**

Serial vs. Parallel Thinking . . . 356
Single Instruction, Multiple Data on CPU . . . 358
Introduction to SIMD . . . 358
SIMD on x86 . . . 359
SIMD on GPU . . . 364
GPU Architecture . . . 365
Nvidia GPU Assembly Programming . . . 366
SASS Dialects . . . 372
Higher-Level GPU Programming . . . 375
Multiple Instruction, Multiple Data . . . 378
MIMD on a Single Processor . . . 378
Shared-Memory MIMD . . . 378
MIMD Distributed Computing . . . 383
Instructionless Parallelism . . . 387
Dataflow Architectures . . . 387
Dataflow Compilers . . . 388
Hardware Neural Networks . . . 389
Summary . . . 390
Exercises . . . 392
Further Reading . . . 395

16
FUTURE ARCHITECTURES **397**

The New Golden Age . . . 398
Open Source Architectures . . . 398
Atomic-Scale Transistors . . . 399
3D Silicon Architectures . . . 400
10,000-Year Memory . . . 401
Optical Architectures . . . 402
Optical Transistors . . . 402
Optical Correlators . . . 403
Optical Neural Networks . . . 405

DNA Architectures 405
Synthetic Biology 406
DNA Computing 407
Neural Architectures 407
Transistors vs. Ion Channels 408
Logic Gates vs. Neurons 409
Copper Wires vs. Chemical Signals 410
Simple Machines vs. Cortical Columns 410
Chips vs. Cortex 412
Parallel vs. Serial Computation 413
Quantum Architectures 414
A Cartoon Version of Quantum Mechanics 414
The Math Version of Quantum Mechanics 415
Quantum Registers of Qubits 416
Computation Across Worlds 418
Practical Quantum Architectures 419
Future Physics Architectures 420
Summary 422
Exercises 422
Further Reading 423

APPENDIX: OPERATING SYSTEM SUPPORT 425

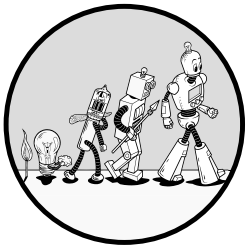
Concurrency 426
Kernel Mode and User Mode 426
Virtual Memory 427
Device Drivers 427
Loaders 428
Linkers 429
Extra Boot Sequence Stages 430
Hypervisor Mode, Virtualization, and Containers 432
Real-Time Operating Systems 432
Speculative Execution Vulnerabilities 433
Exercises 436
Further Reading 436

ACKNOWLEDGMENTS 437

FIGURE CREDITS 439

INDEX 447

INTRODUCTION



This book explores the field of *computer architecture*, examining the underlying principles and design of computer hardware. The field spans a wide range of hardware components and technologies at many levels, from basic silicon and transistors, through logic gates, simple calculating machines, and assembly languages, up to complex processors and memory.

The book also traces the history of computing, from ancient Greek mechanisms to World War II code-breaking machines, retro 8-bit game consoles, highly optimized contemporary CPUs and deep-learning GPUs, embedded Internet of Things devices and cloud servers, and even future architectures, such as quantum computers. Computer architecture identifies the trends connecting these various machines and components. As you'll see, some computing principles are much older than you think.

Who Is This Book For?

Computer architecture is one of a few subjects that separate full-blown computer scientists from mere programmers. If you're an undergraduate computer science student, it's probably a requirement for your degree. If you're a self-taught programmer or hacker, it may be a subject you wish to learn more about, both to make your programs run more harmoniously with your hardware and as a badge of professionalism that many employers look for. This book assumes you know some basic high school programming, math, and physics, but otherwise is self-contained. It can serve as a textbook for the hardware requirements of an undergraduate computer architecture degree or as a first resource for independent learners.

Why Computer Architecture?

When I was a young programmer in the 1980s, programming and using a computer was deeply intertwined with an understanding of the computer's design. For example, the art of writing games on 8-bit micros in the 1980s was very much about becoming one with the particular CPU and chipset in your home microcomputer; we were fiercely loyal to our chosen architectures. Computers had very limited resources, so games were written to exploit specific features of the architecture and to squeeze out as much power as possible. Many game concepts from this time emerged as results of specific architectural structures and tricks.

Programming today is quite different. In most application-level programming, there are many levels of software hierarchy insulating the programmer from the hardware. You might program in a language whose types bear little relation to those of the processor and memory. Those types spare you from thinking in terms of memory addresses—or at least they sit on an operating system that replaces physical with virtual memory addresses and prohibits access to programs stored in hardware other than via some abstracted interface of system calls. As a result, when programmers from the 8-bit era see today's reconstructed Android and JavaScript versions of their games, they can find them inauthentic. The games have lost the intimate connection to the hardware that inspired and constrained them.

Some people, *systems programmers*, design and maintain the stack of tools mediating between hardware and software, but everyone else sits on top of that stack. Nevertheless, those tools still connect you to the underlying hardware, albeit indirectly, and if you understand the hardware's structure, you can often make more effective use of the tools at all levels. You can also take better measurements of how the tools are performing and use that information to make smarter choices in your programs. You might use a more efficient algorithm or change how some process is implemented.

Programmers who really care about high performance, such as writers of game engines and science and financial simulations, can benefit from cutting through some of the layers of the stack and talking directly to the "bare metal" hardware. This kind of programming, known as *assembly programming*, is rare today, since optimizing compilers can generally beat most

handwritten assembly attempts; still, some programmers like to move closer to the metal. They might do this by switching from a memory-managed language to a pointer-based one, or from a language using symbolic types to languages using the machine's own types. For many decades, the lower-level language of choice has been C, though new contenders, such as Rust, are always emerging.

Computer architecture is also directly relevant to computer security. Attacks often involve working at some lower level than the one assumed to be secure. While a computer may be proved safe at some level, such as the userland of an operating system, lower-level details like the precise timings of CPU components and the speed of access to different memory locations open new possibilities for exploitation. For example, the Spectre and Meltdown vulnerabilities exist at the CPU level but can be measured and exploited by userland code when the programmer understands what to look for.

Lastly, by studying the history of computer architecture and seeing how the field has evolved not just over decades but also over centuries, we can learn from past mistakes and discover new uses for old ideas. It's quite common for concepts from historical architectures to come back into use after long periods. For example, to look at a number, Charles Babbage's mechanical computers had to physically move it out of RAM and into the processor; this meant the number could exist only in one place at a time, rather than being copied. We now see exactly this structure in today's research into quantum computing, where some of Babbage's ideas to work around the problem may find new life. The history of architecture acts as a reservoir of ideas that we can draw upon as needed.

Changes in the Field

Until recently, computer architecture was a boring, mature subject. It was developed in the 1950s and 1960s, and its basic principles held for a long time. Books were updated periodically to feature the latest product examples, such as CPUs that were faster and used smaller transistors, but the architectural principles remained the same. Since 2010, however, this has all changed: the subject has entered a new "golden age," in part due to the shifting requirements of other branches of computing. There's been a recent trend away from traditional desktops, in two opposite directions.

First, computers are becoming *less* powerful, both in the sense of computational ability and energy usage. We now want to have larger numbers of smaller, cheaper, lower-energy computers supporting all aspects of our lives. These sorts of devices enable *smart cities*, *smart agriculture*, *smart transport*, and the *Internet of Things*. At the same time, these devices collect vast quantities of data—what we now call *big data*—and processing that data requires a second, new type of computer: extremely large supercomputers or compute clusters located in purpose-built sites the size of factories. Inside these buildings there are no people, only rows of blinking server lights.

You may have heard of *deep learning*, a rebranding of the 60-year-old *neural network* algorithm. It could be argued that deep learning isn't a branch of machine learning or AI theory at all, but rather a branch of computer architecture. After all, it's new architectures, based on massive hardware parallelization through clusters of GPUs and custom silicon, that have brought the old algorithm up to speed, enabling it to run at scales many orders of magnitude larger than before. Thanks to these advances in computer architecture, we can finally use neural networks to solve real-world problems such as recognizing objects in videos and conversing with humans in natural language chatbots.

Another architectural change has shaken a long-held belief. For many decades, programmers swore by *Moore's law*, which said, depending on who you believe, that either the number of transistors or the speed of processors would double every 18 months. This made programmers complacent, thinking they would get ever-increasing speed on conventional architectures. Recently, however, considerations of energy usage have brought the speed form of Moore's law to an end. We're still able to build more and more transistors, but for the first time since the Victorian era, we now need to reconceptualize computing as inherently parallel in order to make use of them.

It remains an open question whether parallel architectures will be visible to programmers in the future and thus require inherently parallel thinking for everyday programming, or whether people will write new compilers that translate between conventional serial programs and novel parallel architectures. Either way, there will be exciting new careers figuring it out. We're looking for new ideas that might come from very old sources, such as clockwork and water computers, or from very new ones, such as neural, optical, and quantum computing.

Finally, the recent widespread availability of online collaboration tools has enabled a new wave of open source architectural systems and tools. RISC-V, BOOM, and Chisel, as well as emulators of past, present, and future machines, have all made the study of computer architecture easier, quicker, and more accessible. You'll be introduced to many of these tools in this book. For the first time in a long time, it's very exciting to study and teach architecture!

How to Use This Book

Architecture is often a compulsory course or professional requirement, and many people who don't enjoy the subject still have to learn it—I should know, I used to be one of them! For such students, I've added some spoonfuls of sugar to help the medicine go down: I'll link the subject to other topics you might be more passionate about. If you hate hardware but like music, robotics, AI, history, or even advanced LEGO building, then this might be the book for you. You might even begin to love architecture through one of these connections; or, if you do just need to pass an exam, maybe this book will help you get through it less painfully than some of the others.

Though the future of computer architecture is novel and exciting, it's important to know the past, so this book takes a broadly historical approach. Computers have generally grown more complex over time; by tracing their history, we can progressively build up this complexity. For example, you'll learn the basic structures of a CPU—as still in use today—by learning to program on a steampunk, Victorian Analytical Engine. I'll show you how to convert its moving mechanical parts into logic gate–based equivalents and how to build up and program a version of the Manchester Baby, one of the first electronic computers. You'll then extend electronic machines to 8-bit and 16-bit retro gaming computers, learning to program a Commodore 64 and an Amiga. Next, I'll introduce modern desktop and smart computers, including x86 and RISC-V architectures, before moving on to cloud and supercomputers. Finally, we'll look at ideas for future technologies.

We'll study many example systems in this book, but they're intended primarily to illustrate the general concepts rather than be a guide to the specifics of modern products. When you've finished reading the book, you should have enough understanding, for example, to be able to build an 8-bit micro on a breadboard, write retro 8-bit games in assembly, write basic embedded and parallel programs, understand the arc of history, and predict some of the future of architecture. You should also then be ready to read the canonical reference books for future study and work.

You'll get the most out of this book if you try to geek out over each chapter—don't just take them at face value. For example, large-scale CPUs can be designed in LogiSim, burned onto cheap field programmable gate arrays (FPGAs), and run for real. As another example, you can use all of the architectures and assemblers presented in this book to write your own video games. The LogiSim files and assembly code snippets discussed in the book are all available for download; see the book's web page, *https://nostarch.com/computerarchitecture*, for a link. I also encourage you to learn more about the book's topics by using the library, Wikipedia, and the wider web, and finding the further readings listed at the end of each chapter; then find the interesting resources that *they* reference, too. Likewise, try to use the tools presented in the book's end-of-chapter exercises in new ways, and look out for other interesting project ideas online. For instance, many YouTubers have built simple 8-bit computers by ordering a 6502, RAM chips, and wires from eBay. Architecture is a particularly visual, bloggable, and YouTube-friendly subject, so if you create something interesting, be sure to share the results.

A good way to begin your study of architecture is to buy a set of small screwdrivers and void your products' warranties by opening up your PC, laptop, and smartphone, as well as some less obvious devices such as your router, TV, and washing machine. In the following section, we'll see some examples of what you might find inside these devices and how to navigate around them.

Inside Common Devices

Computer architecture ranges from the atomic scale of transistors to the planetary scale of internetworked grid computing. To get an early feel for the subject, we'll begin here at the most human level: what we see when we take the cover off a domestic computer and look inside. Typically, the main components visible to the naked eye are silicon chips arranged on a printed circuit board. Later in the book, we'll dig down through chips, logic gates, and transistors and build upward to clusters and grids.

Desktop PC

For the last couple of decades, most desktop PCs have used components and casings of standardized sizes, so you can assemble a PC from components made by many competitors without worrying about whether they'll fit together. IBM started this trend in the 1980s. Thanks to this standardization, if you remove the screws and cover and open up a desktop PC, you'll usually see something like the structure shown in Figure 1.



Figure 1: The inside of a desktop PC

The key feature is a large printed circuit board called a *mainboard* (aka a *motherboard* or *systemboard*), with other smaller boards plugged into it at right angles. The mainboard, as shown in Figure 2, contains the essential parts of the computer, including the *central processing unit (CPU)*, sometimes just called the *processor*, and main memory; the other boards are optional extensions.

You can usually locate the CPU by eye: it's the center of the system, looking like the capital city of a country on a map, with all the roads leading to it. It's typically under a very large fan to disperse the heat created by all the CPU's transistors. The memory is the next most important component. Main memory is usually clearly visible as some physically large but homogeneous region; this is because main memory is computationally large and homogeneous. In a desktop, main memory appears in several boards of several identical RAM chips, clearly lined up.

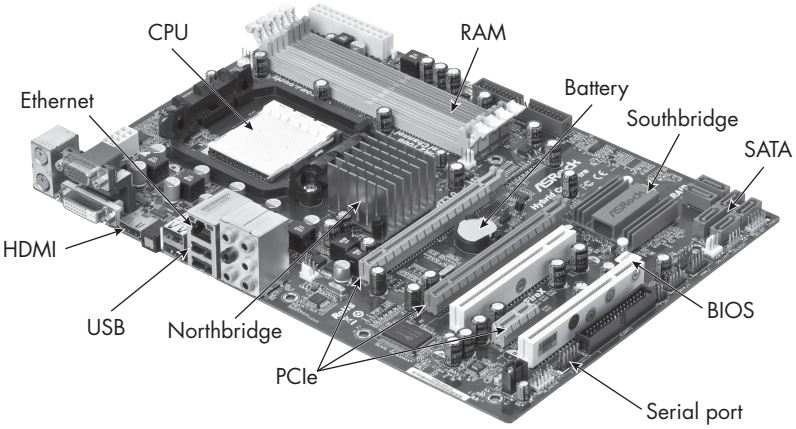


Figure 2: A mainboard from a desktop PC

Printed circuit boards (PCBs), such as mainboards, are made using a similar process to silk-screening posters or T-shirts, as shown in Figure 3. In silk-screen printing, you choose a number of colors for your design and buy a can of paint in each. You then use a CAD program to make your design from block regions of these colors. You print out binary images for each color onto separate transparencies, to show where that color paint will go, and then you make a silk screen *mask* for each color. A mask begins as a piece of silk fabric, which you cover in a light-sensitive gel. You place the transparency that you printed out over it, then shine a bright light onto it.



Figure 3: To print this poster, a silk screen mask is placed on a piece of paper, and paint is pushed through it using a squeegee.

The gel reacts where the printout is transparent and doesn't react where the printout is black. You then put the mask in water, which washes away the reacted parts of gel and leaves in place the non-reacted parts. The remaining gel doesn't allow paint to pass through it, but the exposed areas of the silk fabric do allow paint to pass through. You can now lay your mask on top of a blank piece of paper or T-shirt, pour paint all over it, and the paint will pass

through only in the desired areas of your design. Once you allow this color layer to dry, you repeat the whole process for each remaining color to build up the design.

PCBs can be made similarly. You start with an acid-proof fiberglass insulator board, completely covered with a layer of copper. You design the circuit layout in a CAD program, print it onto a transparency, and shine a light through the transparency to selectively mask a light-sensitive chemical on the board. You then dip the board in acid to remove the unmasked parts of the copper layer. The remaining copper forms the wiring for the PCB. Next, you solder electronic components onto the appropriate locations on the board. The soldering used to be done by hand, but now it's done by robots, which are much more accurate and can handle smaller components.

Beyond the mainboard, the rest of a PC case contains power transformers that convert domestic electricity into the various voltages used in the computer, as well as bulk storage devices such as hard disks and optical media disc drives—that is, CD, DVD, or Blu-ray.

In previous decades, PCs had lots of extension cards for interfacing with the monitor, sound equipment, and networks, but more recently these standard interfaces have moved to chips on the mainboard. The standard-sized PC case (called the ATX form factor) often contains mostly empty space in modern desktops as the parts have been miniaturized and integrated onto the mainboard. The noticeable exception to this trend is the graphics card (graphics processing unit, or GPU), which in high-end machines may have grown as large as or larger than the mainboard to enable fast 3D video games and scientific computing. Gamers like to show off these cards by illuminating them with LED lights and using transparent PC cases.

Laptop

Laptop PCs have the same logical structure as desktops, but they use smaller, less power-consuming components, albeit with less computing power and higher manufacturing costs. Figure 4 shows an example laptop mainboard.

The laptop mainboard isn't perfectly rectangular; rather, it's shaped to fit the available space. Since there's no room for large connectors, many components are soldered directly together. Rather than having swappable extension cards sticking out at right angles to the mainboard, the form factors here are chosen to make everything fit neatly under the keyboard. There's also less standardization of form factors and components than for desktops, with each manufacturer choosing their own. Together, these features tend to make laptops more expensive and harder to upgrade or interchange.

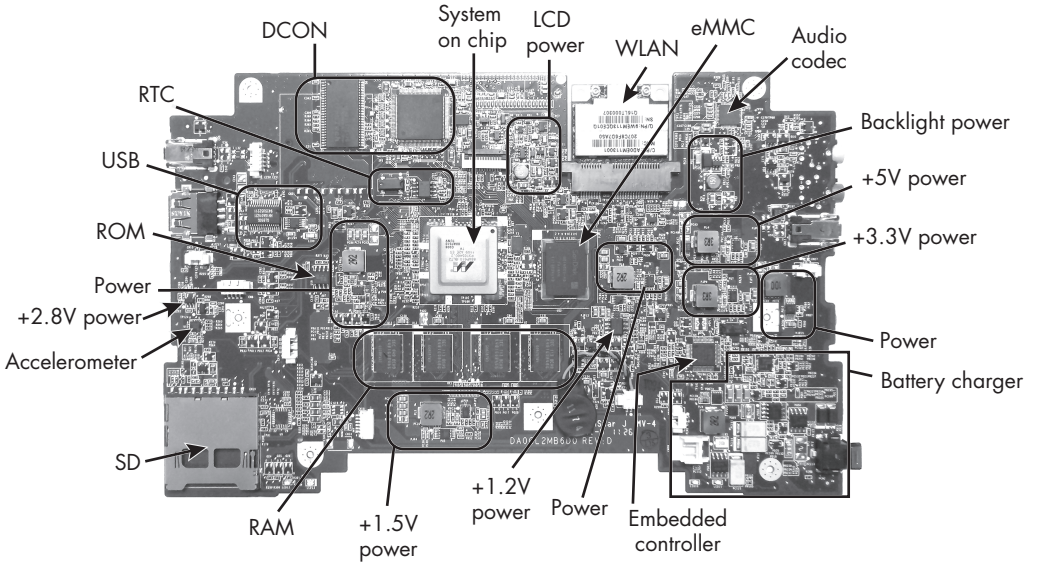


Figure 4: A laptop mainboard

In recent years, the secure boot systems in laptops have opened up a security application of computer architecture. It used to be quite easy to take your employer's laptop, which was encumbered by some proprietary operating system, remove the operating system, and replace it with an open source one such as Linux. The makers of the proprietary operating systems tried to remove your freedom to do this, in some cases claiming it was what your employer wanted, by paying hardware makers to implement secure boot systems. These systems lock the user out of access to the hard disk's boot sector even before the operating system or a bootloader has a chance to load. You now need to circumvent the secure boot at the hardware level, such as by hot-wiring two pins on a dedicated chip together, thus factory-resetting the computer. The pins are quite small nowadays, so it sometimes requires a microscope and precision soldering to do the hot-wiring. (This is purely hypothetical, as it may be illegal to tamper with your employer's device or with the hardware maker's agreements with the operating system vendor.)

Smartphone

In a computing context, the word *smart* nowadays means "is a computer." Historically, consumer devices like phones and televisions were designed for single purposes, but a recent trend has been to include full computation power in them. For a while, this was a novelty, but now a large portion of the world's population carry a full computer in their pocket. We therefore need to take smartphones and other smart devices seriously as computers, just as much as traditional desktops and laptops. Figure 5 shows a mainboard for a smartphone.

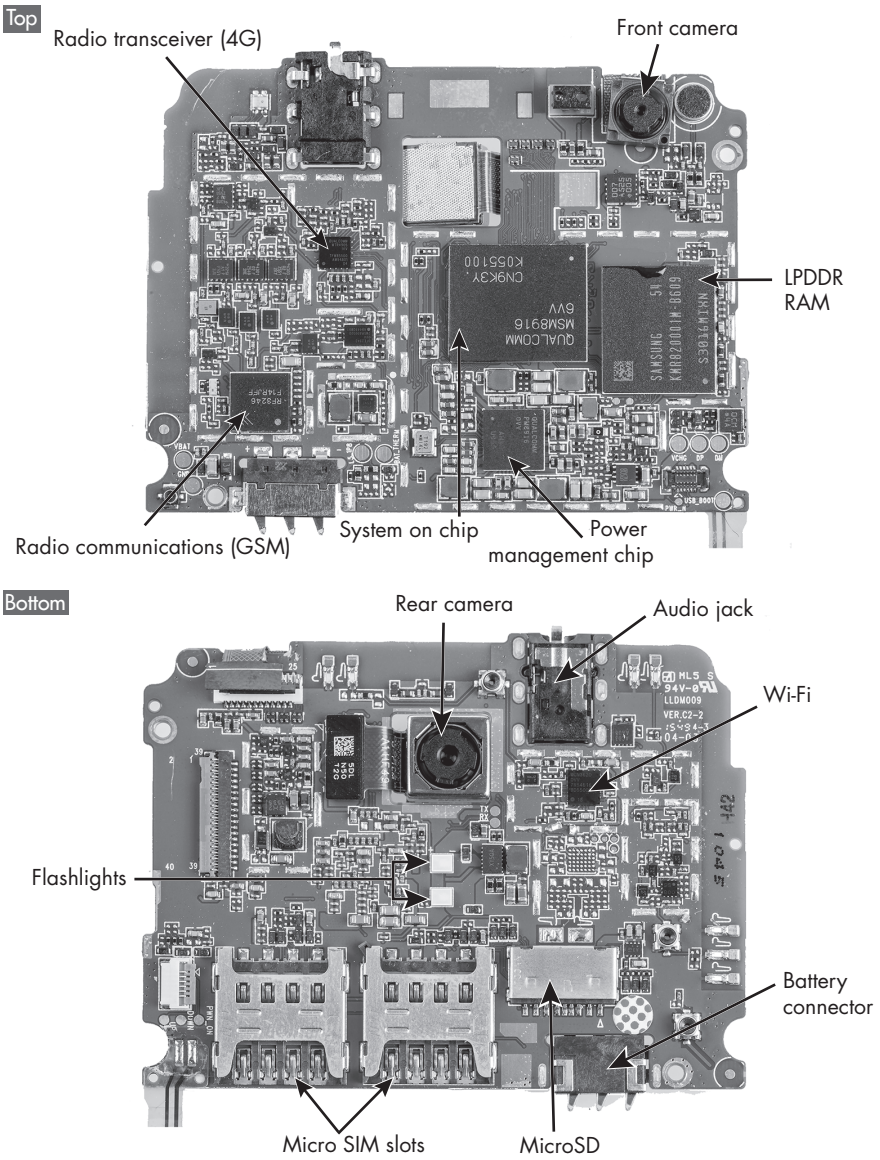


Figure 5: The inside of a Wileyfox Swift smartphone, showing the top and bottom sides of the mainboard

The design is based around an ARM Cortex CPU. Some of the other chips are specialized for phone-specific roles, including Wi-Fi and cell network (GSM) radio communications, battery management, and position and environment sensing (such as the inertial measurement unit and temperature and pressure sensors). The memory is different from desktops and laptops—here we see the use of low-power RAM (LPDDR). This reduces battery usage by clearing and turning off parts of the memory when they aren't needed.

Computers are now so miniaturized that their connectors may be the bottlenecks taking up the most space, rather than the actual computer. For example, replacing a phone's 3.5 mm headphone jack connector with a smaller port is an ongoing debate. No longer having a standard headphone connector can be a nuisance, but having one is a limiting factor for the phone's size.

Washing Machine

If our phones and TVs are computers, might we even consider our washing machine to be a computer nowadays? Figure 6 shows the mainboard of a typical modern washing machine.

There's a small processor on the board, which probably contains *firmware*, a single program "burned" into the chip that performs only one task. This is an example of the embedded systems we'll discuss in Chapter 12.

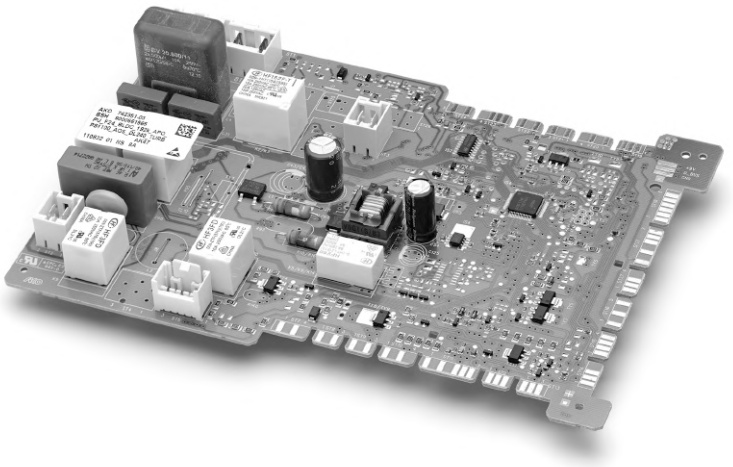


Figure 6: The mainboard of a washing machine

Consumer devices such as washing machines and refrigerators are currently of interest because, like phones, they might be next to become "smart"—that is, capable of running arbitrary programs. When "smart homes" are complete, users will expect to be able to dial into their washing machine to check its status and give it commands from far away. A smart washing machine might even come with an app store that enables you to download and run extra functions, such as machine learning tools. These could detect and appropriately wash different individual items of clothing, saving money and Earth's energy and water resources.

That concludes our tour of a few devices. As we dive into computer architecture in the coming chapters, our understanding of how these devices work and are organized will grow. Before we get started, here's a quick overview of the book and a few exercises for you to try.

Book Overview

Part I introduces the fundamental concepts underlying all architectures.

Chapter 1: Historical Architectures Describes the historical evolution of computing, not just to teach the history itself but because many concepts recur and increase in complexity over time; this will enable you to learn about complex modern systems by understanding their simpler ancestors first.

Chapter 2: Data Representation Discusses how to represent data using binary coding schemes, which will later be implemented using digital logic.

Chapter 3: Basic CPU-Based Architecture Explores what a CPU is, its basic subcomponents, and its machine code user interface.

Once you understand the concepts from Part I, the core structures of computer architecture are fundamentally hierarchical; Part II works its way up this hierarchy.

Chapter 4: Switches Introduces switches, the basic building blocks of modern computers.

Chapter 5: Digital Logic Constructs logic gates from these switches.

Chapter 6: Simple Machines Combines these logic gates into simple machines.

Chapter 7: Digital CPU Design Uses these simple machines to make components of a CPU and, finally, a complete but small-scale CPU.

Chapter 8: Advanced CPU Design Introduces more advanced, modern CPU features such as pipelining and out-of-order execution.

Chapter 9: Input/Output Adds input/output (I/O), making one more step from a CPU to a complete computer.

Chapter 10: Memory Introduces memory, the last requirement for a full computer.

Part III consists of progressively complex examples and applications, coinciding roughly with their historical order; these examples are intended to reinforce your knowledge of the structures studied in Part II.

Chapter 11: Retro Architectures Begins with relatively simple, complete retro computers from the 8-bit and 16-bit era, including showing you how to write retro video games in their assembly languages.

Chapter 12: Embedded Architectures Shows how modern, low-power Internet of Things devices have similar structures, capabilities, and programming styles to retro devices.

Chapter 13: Desktop Architectures Studies the complex instruction set and history of the x86 architecture, which is probably the basis for your main desktop computer. This will enable you to program your computer in assembly, on “bare metal” (that is, without an operating

system getting in the way). You'll also explore the common PC I/O standards and peripherals that your desktop likely contains.

Chapter 14: Smart Architectures Turns to the multitude of smaller smart devices replacing desktop computers. These are characterized by RISC architectures such as RISC-V, along with assembly programming and digital logic design tooling.

Chapter 15: Parallel Architectures Discusses parallel architectures, which have become more common as CPUs have struggled to run faster.

Chapter 16: Future Architectures Concludes by speculating about possible future architectures, including neural, DNA, and quantum computing.

ARCHITECTURE, ORGANIZATION, OR DESIGN?

Computer architecture is traditionally distinguished from *computer organization*, with the former referring to the design of hardware-software interfaces visible to the programmer and the latter referring to the hardware implementations of these interfaces not visible to the programmer. In this context, the programmer was considered to be working at the level of assembly language, which performed the role of the programmer interface. In the modern world, however, it's rare for programmers to see the assembly language level, as they almost always work in compiled languages. The compiler and now the operating system—and even higher-level structures like libraries and game engines—abstract the user many further levels above the old assembly interface. As such, the old architecture-versus-organization distinction has become less meaningful.

In this book, we'll instead use *architecture* to refer to the study of all the above, and we'll use the term *instruction set architecture (ISA)* to denote the more specific study of the hardware-programmer interface. Our definition of architecture also includes the study of the parts of computer hardware outside the CPU, such as memory and I/O systems, which is sometimes called *computer design*. Modern computers are increasingly interconnected as clusters and clouds, so it can now be hard or meaningless to distinguish a group of tightly connected computers from a single large computer. Our conception of architecture thus also extends to these kinds of systems.

The words *architecture* and *hierarchy* both contain the morpheme *arch*. The connection isn't trivial: architecture is all about hierarchies. Hierarchies are the ways that complete structures are organized into components and subcomponents. No human could comprehend the structure of a billion transistors on a chip, but as in software design, we survive by mentally chunking them into many layers of abstraction. We chunk transistors into groups of about four or five, called logic gates; then we chunk logic gates into simple machines like adders; then we chunk those machine into components of CPUs and then the CPUs themselves. This way, each level can be designed from tens or hundreds of comprehensible components, with the designer having to think only at the single level at which they're working. As mentioned earlier, the structure of Part II of this book follows this hierarchy, beginning with transistors and building upward, introducing progressively larger and higher structures.

Exercises

Each chapter ends with some exercises to help you apply what you've learned to real-world systems. Some of the tasks, indicated by a "Challenging" heading, present an extra challenge and therefore are more difficult. Tasks under a "More Challenging" heading are extremely hard or time-consuming and are intended more as suggestions for large personal projects.

Inside Your Own Devices

1. If you're happy to void the warranty of your devices, buy a set of small screwdrivers and open up a desktop PC to see what's inside. Take care to only open the box and not disturb any of the circuit boards themselves. Try to identify the main components, including the power supply, mainboard, CPU, RAM, GPU, and communications devices, as in the examples we discussed earlier. If you're unsure about your screwdriver skills, you might wish to practice on an older, sacrificial device before or instead of your main one, or to search for internet videos of other people opening a similar device.
2. Most of the components you find inside will have a brand name and model number stamped on them. Search the internet for these to obtain the components' formal product datasheets. Use the datasheets to identify some of the parts' key properties, such as the number and speed of the CPU cores, the size of the RAM and its caches, the size of the GPU memory, what input and output devices are present, and what their capabilities and speeds are. (If your CPU is hard to access due to a heatsink, you can usually find its make and model on the mainboard datasheet.)

Software Device Inspection

1. You can also dig for similar information without voiding your warranty on many machines by using software tools that inspect the hardware for you. For example, if you're running Linux, try these commands:

   ```
   lscpu
   cat /proc/cpuinfo
   lshw
   free
   hwinfo
   lspci
   lsusb
   nvidia-smi
   clinfo
   ```

   On Windows, run the Settings program from the Start menu and look around System Settings for similar information.
2. Do some internet research to interpret the results.

3. If you physically opened your device, check that the brands and model numbers inside match those reported by the software—it's quite common and interesting for them to not match, so research why this happens if you see an example!

Challenging

If you're used to opening up desktop computers and looking inside, buy some smaller screwdrivers and do the same for an old laptop.

More Challenging

If you're used to opening up laptops, buy some even smaller screwdrivers and try to do the same for your phone or game console. Some phones can be opened using Torx screwdrivers, although others may require a phone repair kit that you can buy online for a few dollars. Some Japanese consoles use Japanese rather than Western-standard screws. You can order a repair kit for these as well, again for a few dollars. (Some devices are not intended to be accessible or repairable and so are glued together, making it hard to do this.)

Further Reading

This book is intended primarily for readers who want to learn about architecture in order to be *users* of it. It should also be useful, however, for those who want to *work* in architecture, for example, as chip designers. If you're such a reader, you might want to take at least a quick glance at the larger, harder standard text for working architects, to get more of a flavor of what they do:

> John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. (Cambridge, MA: Morgan Kaufmann, 2017).

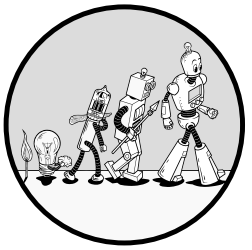
This is the classic and authoritative reference book by the Turing Award–winning inventors of RISC and RISC-V. Just a glance at it is suggested for now. You will likely come back to it after finishing the present book as preparation.

PART I

FUNDAMENTAL CONCEPTS

1

HISTORICAL ARCHITECTURES



Computer science is a much older subject than many people think. This chapter will begin 40,000 years ago and progress to the present day. A modern microchip may seem impenetrable and alien at first sight, but if you know the history, you can understand its smaller components in terms of structures that have developed gradually over thousands of years.

There are a number of other reasons to study the history of the field. Seeing the deep history of computer science gives us more credibility and authority as a field distinct from, say, mathematics or engineering. Seeing how ideas have evolved gradually by building on one another can also protect us from myths of the "lone genius" and reveal how such people were perhaps just like ourselves. Finally, following the general trends through "the arc of history" not only explains how we got to where we are but can also suggest where we're headed next, to help us predict, or create, the future.

What Is a Computer?

When we imagine "a computer" today, we probably think of a device such as a desktop PC, game console, or smartphone. But those aren't the only machines humans have used for calculating and computing. To trace the history of computers, we first need to decide what counts as a computer and how computers are different from mere calculators or calculating machines. This is a surprisingly difficult question, one that is still argued over. My own rule of thumb for deciding if something is a computer is, Can you program *Space Invaders* on it? A simple calculator can't do this, so it isn't a computer; a programmable calculator usually can, so it is a computer.

Let's look at some further concepts that are often suggested for defining computers. Some sources—including the *Oxford English Dictionary*—require computers to be electronic. But similar machines can be made out of other substrates, such as water. Consider *MONIAC*, which stands for Monetary National Income Analogue Computer, a pun on the earlier ENIAC computer that we'll examine later in the chapter. Built in 1949, and shown in Figure 1-1, MONIAC was an analog water computer used to simulate the flow of money through the economy and to illustrate the effects of economic interventions on an economic model.

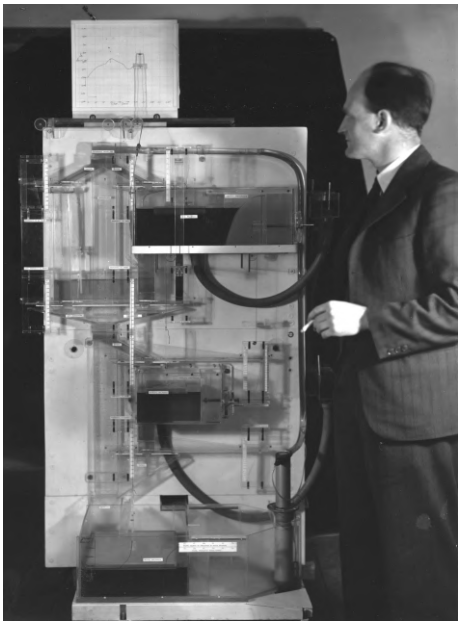


Figure 1-1: The MONIAC water computer and its creator, Bill Phillips

MONIAC allowed you to increase the interest rate and observe the effects on unemployment. Tanks of water showed the positions of money in sectors of the economy such as the central bank, savings, and investment, according to the theory of economics built into the machine.

Some people argue computers must be *digital*, as opposed to *analog*. A digital machine is one that represents data using *digits*, discrete sets of symbols such as the binary digits 0 and 1. In contrast, an analog machine has an infinite, continuous set of possible states, such as the amounts of water in MONIAC's tanks, making MONIAC an analog machine.

Where does MONIAC stand regarding my original *Space Invaders* test? It only computes results for a single economic model, although it might be able to run other economic models if we were able to reconfigure some of the tubes and reservoirs to have different sizes and connections. By extension, perhaps MONIAC could implement *any* computation, such as running *Space Invaders*, through more severe reconfigurations of this nature. But would we then have the same computer in a new configuration, or would we have a new, different machine that still only computes one other, different, thing? In other words, is MONIAC *reprogrammable*?

I've been using *Space Invaders* as a test program, but it's tempting to say that for something to be a computer, you must be able to reprogram it to do *anything*. However, computation theory shows that this can't be used as a definition. Given any candidate computer, it's always possible to find problems it can't solve. These are usually problems about predicting the candidate computer's own future behavior, which can lead it into an infinite loop.

Diving a little deeper into computation theory, we get *Church's thesis*, a more rigorous definition of a computer that most modern computer scientists agree with. It can be paraphrased as:

> A computer is a machine that can simulate any other machine, given as much memory as it asks for.

We'll call machines that satisfy Church's thesis *Church computers*. In particular, machines clearly exist that can do the following, so a Church computer must also be able to perform these tasks:

- Read, write, and process data
- Read, write, and execute programs
- Add (and hence do arithmetic)
- Jump (`goto` statements)
- Branch (`if` statements)

We can now see that the *Space Invaders* definition is a reasonable approximation of Church's thesis in many cases: while *Space Invaders* is a simplistic video game, it happens to require all of the above tasks, which are also the basic ingredients of many other computational tasks and machines. Hence, a machine that can be *reprogrammed* (rather than hardwired) to play *Space Invaders* is usually powerful enough to simulate any other machine, too (as long as we provide as much extra memory as it asks for).

The rest of this chapter traces the history of computers and computer-like devices in chronological order, starting in the Stone Age. As you read, ask yourself who invented the first computer, and note the point where you think the computer was invented. People often argue for drawing this line

in different places, based on their own definitions of what counts as a computer. Where will *you* draw the line, and why?

Before the Industrial Revolution

In this section, we'll take a look at the various preindustrial machines we may or may not consider to be computers. In doing so, we'll see that humans have been using mechanisms resembling computers for longer than we might have thought.

The Stone Age

Our anatomical species, *Homo sapiens*, is around 200,000 years old, but it's widely believed that we lacked modern intelligence until the cognitive revolution of around 40,000 BCE. We don't know exactly how this happened. One current theory is that a single genetic mutation in the FOXP2 gene occurred and was selected by the extreme evolutionary pressures of the Ice Age. This suddenly enabled the brain to form arbitrary new hierarchical concepts, in turn giving rise to language and technology. According to this theory, from then on humans were as intelligent as we are now. They would have been capable of learning, say, quantum computing, had they been given access to modern facilities and information.

One marker of this shift may be the *Lebombo bone*, shown in Figure 1-2—a bone with carved notches that may have been used as a tally stick around 40,000 BCE. In a tally, one mark represents one physical thing. Perhaps these notches signified animals, items of food, favors owed by one person to another, or days to time some hunting or social project.



Figure 1-2: The Lebombo bone

The *Ishango bone*, shown in Figure 1-3, is another bone containing human-made tally-like marks, dating to later in the Ice Age, around 20,000 BCE. Unlike the Lebombo bone, the Ishango bone marks appear to be grouped into tally-like clusters of mostly prime numbers between 3 and 19, and these clusters are grouped into three lines that sum to 60 or 48.

As with the Lebombo bone, it's possible that the marks in the Ishango bone are at purely random locations and were made for some physical purpose, such as to improve hand grip. But several authors have studied the Ishango bone's patterns and argued that they functioned as a tally, an aid for calculation, a lunar agricultural calendar or menstrual cycle calendar, or most speculatively, a table of prime numbers. The totals of 60 and 48 are multiples of 12, and 12 is known to have been the original base for arithmetic in later civilizations, before we shifted to base 10.



Figure 1-3: The Ishango bone, argued by some to extend from tallying to calculation

The Lebombo bone appears to be an example of data representation. Arguably, it may have been used for a simple form of calculation such as adding one to its total each time a new mark was made. Some interpretations of the Ishango bone suggest its use in more advanced calculations, perhaps interactively, like using a pen and paper to perform and keep track of multiple steps of a math problem.

Could you program a bone to play *Space Invaders*? You could devise a set of rules for a human to follow, telling them to make scratches to update representations of the game characters. Gameplay would be quite slow, and the human would have to be there to perform the updates. There's no evidence that humans ever used bones in this programmable way—though maybe one day another bone could be found and its scratches decoded as instructions for a human operator to follow.

The Bronze Age

The ice melted around 4000 BCE, enabling the first cities to grow. Cities required new and larger forms of organization, such as keeping track of trading and taxes. To enable this, by 3000 BCE the Sumerian city culture in Mesopotamia (modern-day Iraq) developed the first writing system, and by 2500 BCE it possessed the first indisputable calculating machine, the abacus (Figure 1-4). The word *abacus* means "sand box," which suggests that before this date the same machinery was implemented using simple rocks in the

sand. The oldest abaci we find in archaeology are the more advanced ones made from wood and beads.

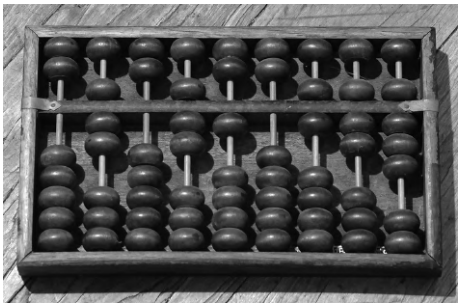


Figure 1-4: An abacus

In its usual usage, the state of the abacus in Figure 1-4 represents the (decimal, natural) number 070710678. There are nine columns, each representing one of the digits in this number. Each column is split into a lower box containing five beads and an upper box containing two beads. The default position for the beads in the lower box is down, and the default position for beads in the upper box is up. In this state, a column represents the digit 0. Each bead pushed up from the bottom to the top of the lower box is worth 1. Each bead pushed down from the top to the bottom of the upper box is worth 5.

To add 1 to a number on the abacus (that is, *increment* it), you raise one bead from the lower box of the rightmost column. If all five beads in a column's lower box are raised, you push them all back down and replace them by lowering one of the beads in the upper box in the same column. If both upper beads are lowered, you push them back up and replace them by raising one bead from the lower box in the column on its left. Moving data from a column to the one on its left is known as a *carry* operation.

To add two numbers, $a + b$, you first set up the abacus to represent the digits of a. You then perform b increments as above. The state of the abacus then represents the result.

This style of calculation—where the first number is "loaded onto" the device and the second is "added into" it, leaving only the final result as the state of the system—is known as an *accumulator architecture*, and it's still in common use today. It "accumulates" the result of a series of calculations; for example, we can add a list of many numbers together by adding each of them in turn into the state and seeing the latest accumulated total after each addition.

NOTE *The abacus in this example uses decimal digits for familiarity. The original Sumerian version used base 12.*

The concept of the algorithm dates from this time. Calculations written on clay tablets, such as those in Figure 1-5, show that number-literate people at this time thought in terms of computation rather than mathematics,

being taught to perform algorithms for arithmetic operations and carrying them out for practice, as opposed to doing proofs.

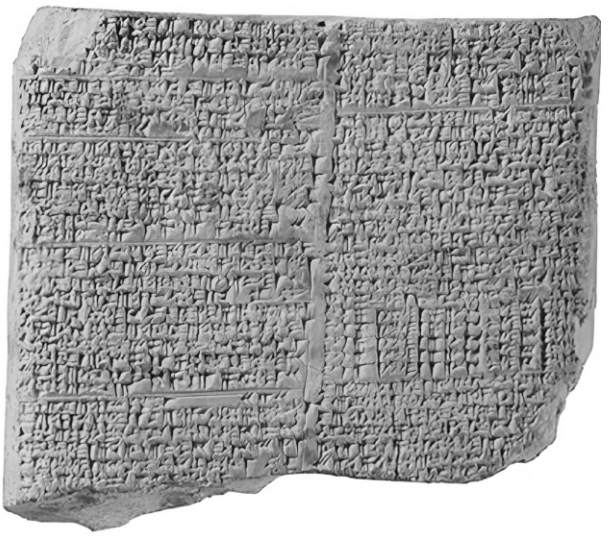


Figure 1-5: A tablet showing the steps of a long division algorithm

The clay tablets show lines of step-by-step arithmetic that may have been performed using the tablets themselves as data storage. Or the tablets may have been used to notate the states of an abacus for teaching purposes.

The abacus was—and in a few places still is—most often used for adding numbers, such as summing the prices of items in a shopping basket, but other ancient abacus arithmetic algorithms are also known, including for subtraction, multiplication, and long division. These were performed similarly to their modern pen-and-paper equivalents. Modern enthusiasts (you can search for them on YouTube) have also shown how to use the abacus for more advanced algorithms such as finding square roots and computing the digits of π. As these algorithms get more complex, the memory of the abacus often needs to be extended with extra columns. Like the Stone Age bones, the abacus could be used as the data store for *any* algorithm if a human is instructed what actions to perform on it. If you want to argue that it's a computer, you may again need to consider the role of the human.

The Iron Age

The Bronze Age city civilizations of Mesopotamia and its neighbors collapsed, mysteriously, around 1200 BCE. They were followed by a "dark age" period, until classical ancient Greece arose around 500 BCE to 300 BCE: the time of Pythagoras, Plato, and Aristotle. Greek power was gradually replaced by the Roman Republic and Roman Empire from around 300 BCE to 400 CE.

The Antikythera mechanism (Figure 1-6) dates from this period, around 100 BCE. It was found in 1901 in a Mediterranean shipwreck; the sunken ship appeared to be on its way from Greece to Rome, with the mechanism for sale or as tribute. The mechanism was only understood and reverse engineered in 2008. We now know that it was a mechanical, clockwork analog machine used to predict astronomical (and likely astrological) events, including five planet positions, moon phases, and the timings of eclipses and the Olympic Games. It consisted of 37 bronze gears, and the user turned a handle to simulate the future course of their states. The results were displayed on clock faces, computed by the ratios of mechanical gears. Enthusiasts recently rebuilt a functioning version using LEGO (Figure 1-6).



Figure 1-6: The Antikythera mechanism remains, as found in a Mediterranean shipwreck (left), and a reconstructed Antikythera mechanism using LEGO (right)

Odometers were long-range distance-measuring machines that the Greeks and Romans used to survey and map their empires. There is indirect evidence of their use from around 300 BCE due to the existence of very accurate distance measurements that would have been hard to obtain any other way. The reconstruction in Figure 1-7 is based on direct archaeological remains from around 50 CE.

This type of odometer worked similarly to the measuring wheels you might have used in elementary school that clicked each time they were pushed a certain distance, typically 1 yard or 1 meter. It is also related to modern odometers used in cars and robotics.

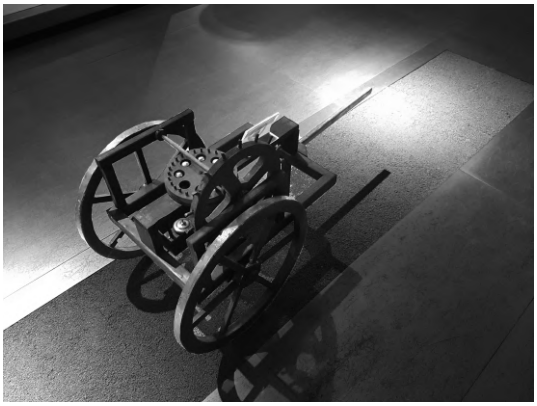


Figure 1-7: A Roman odometry cart

The odometer is pulled by a horse, like a cart. There are a number of metal balls stored in cavities in a circular wooden gear. One of the wheels has a peg attached so that once per rotation it taps and rotates the gear by a small fixed angle. A ball-sized hole under one position of the gear allows a ball above it to fall out of its cavity and into a collecting box below. The total distance traveled is thus logged by the number of balls in the counting box at the end of the trip.

Are these machines computers? There are clearly notions of data being used to represent objects in the world, as well as forms of automation and calculation. But like MONIAC, each machine does only one thing: predict eclipses or measure distance. You couldn't easily reprogram either to play *Space Invaders*.

Like MONIAC, the Antikythera mechanism is an analog machine: its gears rotate continuously and can be in any position. The odometer, in contrast, is digital, like the abacus. Its gear advances only by a discrete amount with each "click" as the peg passes it, and the collecting box always holds a discrete number of balls. Unlike the abacus, however, the odometer is automatic; it doesn't require a human operator, only a horse as a source of power.

You might be able to reprogram the Antikythera mechanism—and with some creativity, the odometer—if you were allowed to completely reconfigure all the gears, including adding and removing gears of arbitrary sizes in arbitrary locations. Then you could try to represent and simulate other physical systems or perform other calculations. As with MONIAC, some consider physically reconfiguring the hardware in this way to be cheating. They would argue that this creates a new, different machine, rather than a different program running on the original machine.

The Islamic Golden Age

After the fall of Rome in 476 CE, western Europe entered the so-called Dark Ages for a thousand years, and the history of computing in western Europe records basically no progress during this time.

However, the Roman Empire continued to operate from its new eastern capital, Byzantium (now Istanbul, Turkey). There was a flow of ideas between Byzantium, Greece, and the Islamic world, the latter becoming the new intellectual center of the time. A particular musical idea from this culture introduces the important concept of programming.

The ancient Greeks previously had a portable *hydraulis* instrument, related to modern church organs. It was composed of a set of pipes, played by a keyboard and powered from an air reservoir pumped by a servant. The Greeks clearly possessed the technology needed to make self-playing versions of the hydraulis, but there's no evidence of them doing so.

It was Islamic scholars, the Banu Musa brothers, who built the first known automated musical instrument: the automated flute player of Baghdad, around 900 CE, shown in Figure 1-8.

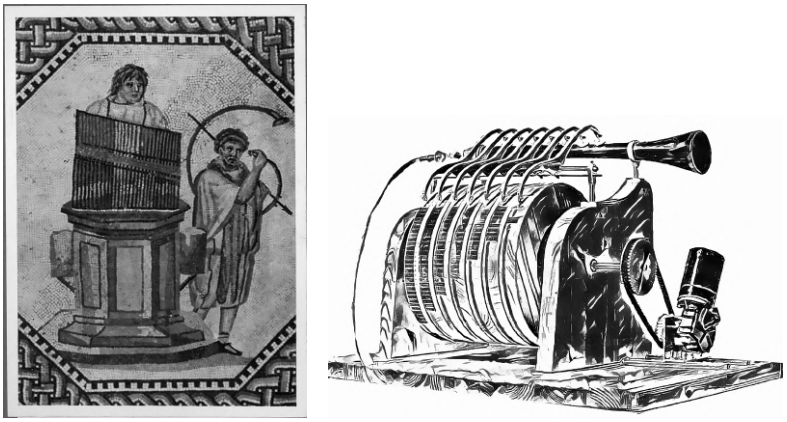


Figure 1-8: A Greek hydraulis (left) and a sketch of the Baghdad automated flute player (right)

The innovation was to use a slowly rotating barrel with movable pins around its edge to indicate the positions of musical notes. As the barrel rotates, the pins make contact with levers that allow air to flow into the instrument to sound a note. The movable nature of the pins allows different compositions to be programmed into the device, making it the first known *programmable* automatic machine. The pins may be viewed today as a binary code: at each time and pitch, there is either a note (1) or no note (0).

Is this a computer? Unlike the Iron Age machines, it can clearly run multiple programs. However, there's no notion of calculation or of decision-making: once a program begins, it will play through and can't change its behavior in response to any input or even to its own state.

The Renaissance and Enlightenment

Byzantium fell in 1453, sending many scholars and their books back to western Europe and helping it wake from the Dark Ages. Leonardo da Vinci was the definitive "renaissance man" of this time: a prolific scientist, artist, and engineer. He possessed many of these old books and looked to them for inspiration. He was probably familiar with Antikythera-like systems thanks to these books. One of his manuscripts from around 1502, the *Codex Madrid*, contains an unbuilt design (Figure 1-9) for a mechanical analog calculator based on Antikythera-like principles.

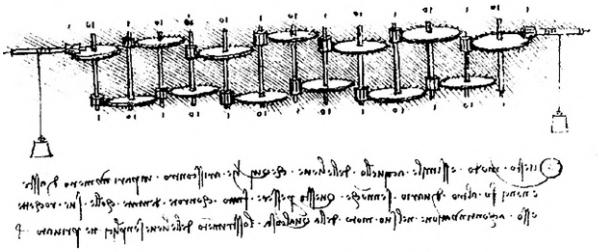


Figure 1-9: The da Vinci calculator's original manuscript

The design was rediscovered and successfully constructed in 1968. There are 13 wheels, each representing the columns of a decimal number. Their possible positions are *continuous*: rather than switching abruptly from one decimal digit to another, they move smoothly by means of gearing. The gear ratio is 1:10 between each pair of columns, so each column's wheel rotates at one-tenth the speed of the column on its right.

Like the abacus, the calculator is an accumulator whose state at any point in time represents a single number, again as digits in columns. One number a can be added to another b. The first number a could be loaded onto the machine by advancing the mechanism to represent its digits. Then it would be turned an additional amount b to advance the total to $a + b$.

For example, to calculate 2,130 + 1,234, we first load 2,130 onto the device, then advance by 1,234 to get 3,364. The numbers wouldn't be precisely aligned at the end of the computation due to the continuous rotation of the wheels. For example, the 6 in the tens place would be almost halfway between showing 6 and 7 because the digit after it is a 4, which is almost halfway to the next carry. In a sense it is a "weaker" machine than the Roman odometer, because the odometer has a notion of converting from continuous wheel positions to discrete symbols using its pin-and-ball mechanism.

Da Vinci's concept was extended by Blaise Pascal in 1642. Figure 1-10 shows Pascal's calculator design and a modern build of it. (It has recently been argued that Pascal's calculator was invented earlier, in 1623, by Wilhelm Schickard.)

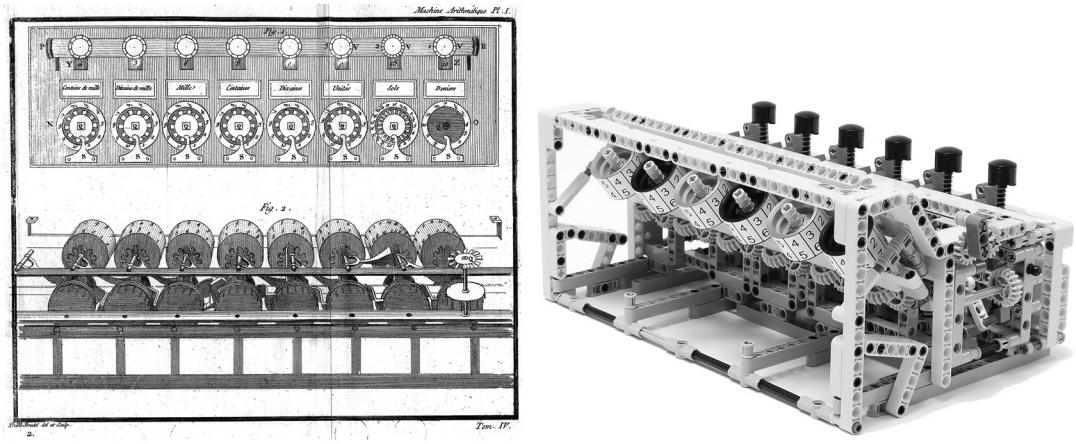


Figure 1-10: Pascal's calculator: the original design and a 2016 LEGO rebuild

Pascal's calculator includes a digital mechanism similar to the odometer (rather than da Vinci's analog gearing) to implement its carry mechanism. When a column reaches the number 9 and another unit is added to it, it triggers a unit turn of the next column as it returns itself to the number 0.

Unlike the Antikythera mechanism, which represented the states of physical (astronomical) objects, da Vinci's and Pascal's machines operate on pure numbers. You could argue this gives them more general-purpose roles than the Antikythera mechanism. That said, the range of their calculations is limited to addition, which in a sense makes them less powerful than the abacus, which had algorithms for other arithmetic operations. On the other hand, like the Antikythera mechanism, these calculators require less human work than an abacus.

Some see the move from da Vinci's analog to Pascal's digital operation as very important. Digital operation appears to involve a simple concept of the machine making a "decision": a carry is either made or not made at each step. Decision-making is certainly important for some tasks, but clearly not so much for addition because both calculators can do it equally well.

The Steam Age

Steam power had been known to the Greeks and Romans as a curiosity, and anyone who has ever boiled water with a lid will have noticed that steam can move the lid around. But it was only from around 1700 in Britain that steam was harnessed in earnest, to power the industrial revolution. Seeded by Enlightenment ideas, especially Newton's physics, this was a positive feedback cycle in which machines and coal were used to produce more machines and extract more coal. Coal was burned to heat water into steam, and steam was first used to pump water from coal mines. In time, steam came to power many other machines, some with computer-like characteristics.

The Jacquard Loom

The production of textiles was a major application of new machines during the Steam Age. But unlike plain cotton clothes, traditional weaving patterns were highly complex. Thus they were considered to be more valuable because they were rarer and more expensive.

In 1804, Joseph Jacquard created a variant of the weaving machines of the time that employed replaceable punched cards to guide the positions of the hooks and needles used in the weave (Figure 1-11).



Figure 1-11: A Jacquard loom

The punched cards could be "chained" together into long tapes to make complex, reusable patterns at a lower price.

NOTE *"Chain" became the standard command to load the next program from magnetic tapes in later electronic devices, used until the 1990s. Weaving concepts like "thread" and "warp" are also used as metaphors in modern multithreaded programming and in parallel GPUs.*

Victorian Barrel Organs and Music Boxes

Barrel-based musical instruments, similar in technology to the Baghdad automatic flute player and shown in Figure 1-12, were popular during the 19th century. The job of an "organ grinder" was to push a portable barrel organ onto a main street, then manually turn its handle to provide power. A rotating barrel with pins marking the positions of notes would then allow air into the organ pipes, as in the Baghdad version.



Figure 1-12: Two Victorian-style barrel organs (left and center) and a music box (right)

The same mechanism was (and still is) used in the music box from this period, in which a spring is wound up to store energy and then released to power a smaller pinned barrel, whose pins strike small xylophone-like metal bars directly to play a few bars of music such as a famous theme from a ballet. The rotating barrel is often topped with a small sculpture, such as a ballerina, that rotates along with the music.

Charles Babbage hated organ grinders playing outside his house and led a public campaign to rid them from the streets of London. But their barrel organs were to form a fundamental influence on his work.

Babbage's Difference Engine

Babbage designed two different machines, the Difference Engine and the Analytical Engine. The former (Figure 1-13) was first; it was successfully built and commercialized by Georg Scheutz and others from 1855 and widely used in industry until the 1930s. Recent LEGO rebuilds also exist.

The Difference Engine was designed to produce tables of values of arbitrary polynomial functions. Most mathematical functions can be well approximated by polynomials via Taylor series expansion, so the machine could be used to make tables of values for any such function. You may have used similar tables in modern exams to look up values of trigonometric or statistical functions when a calculator isn't allowed. In Babbage's time, the killer application of these tables was in shipping, for navigation purposes. Tables had previously been computed by hand and contained many expensive errors, so there was a large economic demand to perfect them by machine.

The machine can be powered either by steam or by a human cranking the handle. Like Pascal's calculator, the Difference Engine represents decimal digits by discretized rotations of gears. Numbers are represented by a vertical column of such digits (like Pascal's calculator turned on its side). The Difference Engine then extends this to a 2D parallel architecture, with multiple vertical columns arranged horizontally. Each of these columns represents a different number.

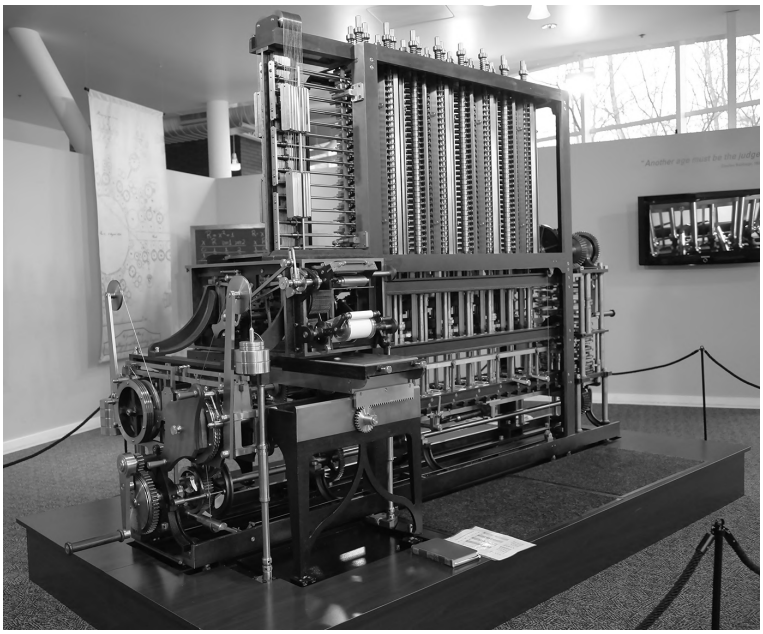


Figure 1-13: A metal rebuild of Babbage's Difference Engine

There are two dimensions of parallelization in the Difference Engine: digit-wise and term-wise. Digit-wise addition, for example, is a different algorithm from the sequential high school method of addition. Instead of starting from the rightmost column and moving left and applying carries, it adds each pair of digits at the same time, then handles any carrying afterward. For example, to add 364 + 152, the three additions 3 + 1, 6 + 5, and 4 + 2 are all performed at the same time to give 416. The carry from 6 + 5 = 11 is then added to give 516. Carrying is a difficult operation to get right in this context, and Babbage devoted most of his engineering time to it. The visual effect of carries can be seen on YouTube videos of the Difference Engine as a visible ripple of information propagating across the 2D surface of the machine. Such ripples are also seen in computations on modern parallel GPUs.

Is the Difference Engine a computer? It can run different "programs" to calculate different equations, but these equations have no obvious concept of changing their behavior during a calculation; there's nothing like an `if` statement to test intermediate results and do something different based on them. It's more like a modern media streaming device in which numbers flow smoothly through a processing pipeline.

Babbage's Analytical Engine

The Difference Engine was limited to computing tables of polynomial functions, but Babbage's second project, the Analytical Engine (Figure 1-14), was designed as a completely general-purpose, programmable machine.

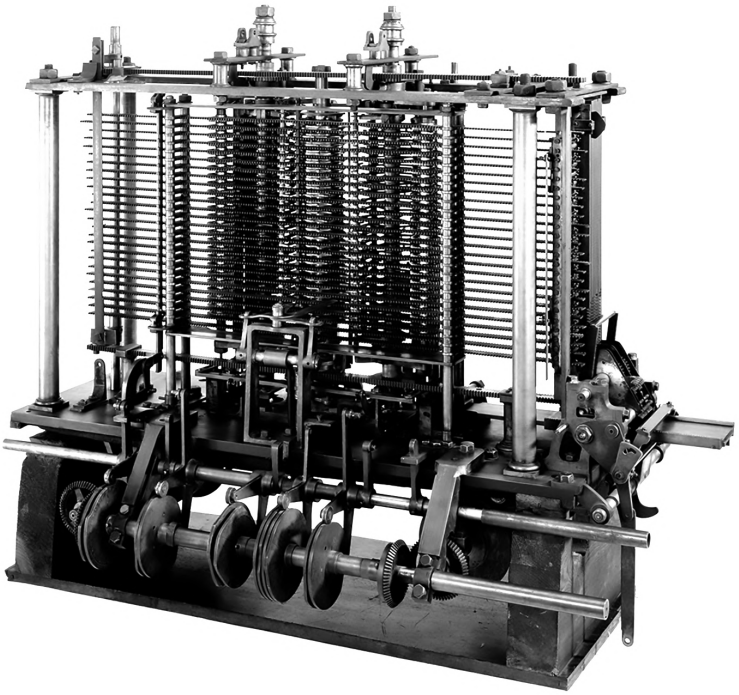


Figure 1-14: A modern partial rebuild of Babbage's Analytical Engine

To obtain this generality, the Analytical Engine provides a range of arithmetic and other operations as simple machines, together with a memory for storing data and the ability to read in programs from punch cards. The programs dictate a sequence of memory reads and writes and arithmetic operations, and allow branching depending on the state of the calculation—an `if` statement.

Babbage went through many variations of the Analytical Engine's design on paper, but physically built only a tiny part of it just before he died. He got very sidetracked with the fine details of the carry mechanism and was obsessed with constantly redesigning components rather than sticking with one version and getting them integrated to actually work. (Today this style of project management would be known as *yak shaving*.) This annoyed the research funding agencies of the time, making it hard for Babbage to get money to build anything. Thus, unlike with the Difference Engine, we don't have a working version or even a single final design document of the Analytical Engine. However, components have recently been reconstructed from Babbage's plans using modern manufacturing technologies.

With more moving parts than the Difference Engine, the Analytical Engine would have required more power; this would have had to come from a steam engine rather than a manual crank. It would have also required more precisely machined gears, as computations would need to work their way through a longer series of gears. Like the factory machines and steam locomotives of the period, it would have smelled of oil, smoke, and steam, and gleamed in polished brass: Babbage was the original steampunk.

The core of the Analytical Engine contained many independent simple machines that each performed some function, such as adding numbers and testing if one number equaled another. The adding machine was roughly a copy of Pascal's calculator, and the other simple machines were variations of it.

The Analytical Engine introduced the modern concept of computer memory. Its "store" would have consisted of a large number of copies of a simple machine, again similar to a Pascal calculator, each of which could retain a different number. Each machine would be given a numerical identifier or "address" to specify the correct one to read from or write to.

A sequence of *instructions* would have been coded in binary and punched onto paper tape, using a mechanism taken from the Jacquard loom. Each instruction would tell the engine to activate one of the simple machines. Usually, after each instruction, the machine would have line-fed the punched paper along to load the next one (a bit like a typewriter). However, the machine also would have had the ability to check the result of the latest simple machine and, depending on its value, could jump to a different line in the paper. This would give programs the ability to alter their behavior in response to intermediate results.

A program could also be made to run forever by gluing the bottom of the punched paper to its top, making a physical loop, as in the (later) paper tape machine shown in Figure 1-15.



Figure 1-15: A punch tape program loop

We don't have any examples of actual programs written for the Analytical Engine. Rather, Babbage and his collaborator Ada Lovelace wrote down example *states* and *outputs* from imaginary runs as long tables, showing them at each step of program execution. This is similar to the notations on the Babylonians' clay tablets, which illustrate algorithms by showing the effects rather than the instructions used to generate them. From these execution traces, modern readers can infer roughly what the programs and the machine's instruction set used to build them would have been.

Babbage wrote the first of these example traces for small, almost trivial mathematical functions, which illustrate roughly the full set of instructions in use. But Babbage was the hardware person, more concerned with designing the machine itself, and never wrote anything longer, thinking that

programming would be relatively trivial compared to designing the architecture. Lovelace was the software person, and she wrote much longer traces for complex functions. She also wrote speculations about what larger programs could achieve, including ideas about AI. If Babbage is claimed as "the first programmer," then Lovelace might be "the first software engineer" for thinking about programming more seriously and at scale.

Was the Analytical Engine a Church computer? Its design contains all the basic features of a modern computer: CPU, memory, a bus, registers, a control unit, and an arithmetic unit. It can read, write, and process data. It can do arithmetic. Unlike the purely calculating machines before it, it can jump (`goto`) and branch (`if`), moving to different instructions in the program according to the state of its calculations.

However, to be able to simulate any other machine, it would need to be able to read, write, and execute programs as well as read, write, and process data. But its programs were fixed on the punched paper, rather than held in memory like in a modern PC. This kind of architecture, where the data and program are stored separately, often with the program fixed as firmware, is called a *Harvard architecture*, as opposed to a *von Neumann architecture*, where the program and data are stored together.

Today, Harvard architectures are used in embedded systems, especially in digital signal processing chips. It's possible to set up a Harvard architecture that can simulate other computers, including those that modify their own programs. This can be done by writing a single *virtual machine (VM)* program on the fixed program punch cards (or modern firmware). The VM reads, executes, and writes further programs in memory.

Lovelace or Babbage could have written a VM program for the Analytical Engine, but they didn't consider it. The same could be said about many other machines, however. For example, a VM could be written for and executed on a Sumerian abacus if a programmer chose to do so. Church's thesis is about the *potential* for a machine to simulate any other machine, not the actualization of it doing so. But it depends on what "level" of machine we consider: the underlying hardware or virtual machines running at higher software levels.

And, of course, the Analytical Engine was never built or tested in full—does this need to be done to justify "being a computer," or is the basic design sufficient by itself?

Mechanical Differential Analyzers

The industrial revolution largely progressed through practical hackers building machines based on their intuitions, then testing whether they worked. But over time, mathematical theories were adapted or invented to describe and predict the behavior of many engineering systems, giving rise to academic engineering. Most of these theories made use of calculus. Developed earlier by Gottfried Wilhelm Leibniz and (independently) Sir Isaac Newton

for different purposes, calculus quickly took off as a general tool for modeling how all kinds of systems, including industrial machinery, change over continuous time, through equations such as

$$\frac{dx}{dt} = f(x)$$

where x is part of the state of the world being modeled, f is some function of it, and dx/dt is the rate of change of x. This type of equation can numerically simulate the state of the world over time by iteratively computing dx/dt and using it to update x. Like making the Difference Engine's tables of polynomials, this is a highly repetitive and error-prone process ripe for mechanical automation.

In 1836, the same year that the Analytical Engine was developed, Gaspard-Gustave de Coriolis realized that since the behavior of a mechanical device could be *described* by a differential equation, the same device could be viewed as computing the solution to that equation. So, to solve a new equation, a physical device could be designed that matched it, and that device could then be run for a period of time to give the required answer.

More general differential equations can involve acceleration and higher derivatives, and multiple variables. Coriolis's idea was extended by others, including Lord Kelvin in 1872 and James Thomson in 1876, to solve these systems, again by constructing analog mechanical devices to match them. The key component of these machines was the ball and disc integrator (Figure 1-16), in which a movable ball transfers motion from a spinning disc to an output shaft.

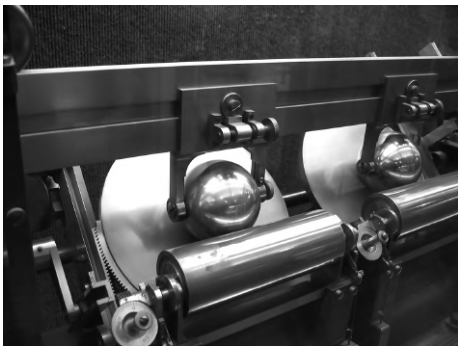


Figure 1-16: A ball and disc integrator from Kelvin's differential analyzer

Like the Difference Engine, these machines were built only to solve a single class of problems: differential equations. But much, or perhaps all, of the world and its problems can be modeled by differential equations. As inherently analog machines, they can be viewed as continuing the tradition of da Vinci's analog calculator, while Babbage's machine built on Pascal's digital calculator.

The concept of using the physical properties of the world to model itself has recently been revived in quantum computing, where simulating physical and chemical quantum systems appears to be a major application with a particularly good fit to the way quantum machines compute.

The Diesel Age

Between the purely mechanical machines of the industrial revolution and later electronic machines, there was a hybrid period in which electricity was combined with mechanical motion to build electromechanical machines.

The key electromechanical technology is the *relay*, a mechanical switch in an electrical circuit whose physical position is controlled using a magnet, which, in turn, is controlled by another electrical signal. Relays are a special type of *solenoid*, a coil of wire that generates a linear magnetic field when a current flows through it. This magnetic field can be used to physically move a magnet (called the *armature*) inside the coil, and that motion can be used, for example, to open and close a valve in a water pipe or to start a car engine. Replace the water pipe with a second electrical circuit, and the valve with an electrical switch, and you have yourself a relay.

Relays are still used today (Figure 1-17). For example, in robotics safety systems, we often need to physically connect and disconnect the main battery to and from the robot's motors. A safety monitor checks if everything is okay and makes the physical relay connection if so, but disconnects it if anything seems wrong. You can hear these relays click when the current changes and the armature physically moves.



Figure 1-17: A relay showing a wire coil

Electromechanical machines were more efficient than purely mechanical ones, and found widespread commercial and military use in the period around the turn of the 20th century and the two World Wars. Some of the machines you'll see in the next section were still in use in the 1980s. Others have uncertain fates due to ongoing government secrecy, as this period includes the cryptology machines of World War II.

The IBM Hollerith Tabulating Machine

The US Constitution requires that a census be taken and processed every 10 years, and by 1890 the population had grown to a size where human processing of its statistics was impossible. This created an embarrassing backlog of work for the government and a strong demand for an automated solution.

Herman Hollerith designed a machine to automate data processing and used it successfully in the 1890 census to do big data analytics on information from 62 million citizens. Each citizen's data was transferred from a

written census form to a punch card by a human clerk. This seems to have been inspired not by Jacquard's and Babbage's machines, but independently by inspectors punching holes in train tickets to represent different journeys or times. Each question on the census was multiple choice, and was encoded on the punch card by punching out one of the several options. Figure 1-18 shows an example of this.

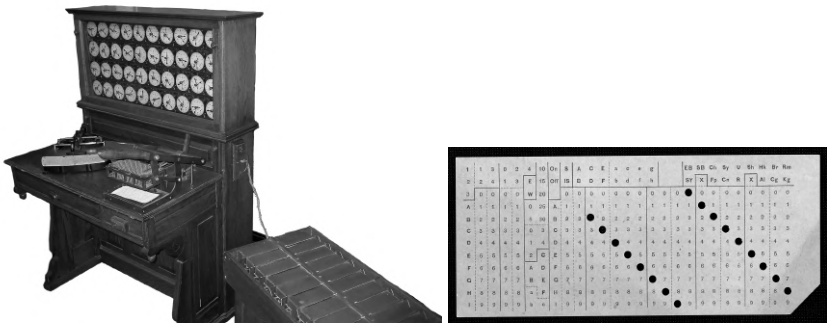


Figure 1-18: A replica of the IBM Hollerith machine (left) and a punched card (right)

Stacks of cards could be read into the machine, which would check for the presence or absence of certain features or combinations of features, then use an electrical analog of a Pascal calculator to accumulate the total count of cards having these features. As Hollerith (1894) explained:

> It is not sufficient to know simply the number of males and females, but we must know, for example, how many males there are at each age-period, as well as how many females at each age-period; or, in other words, we must count age and sex in combination. By a simple use of the well-known electrical relay we can secure this or any other possible combination. It must not be understood that only two items can be combined; in this way any number of items can be combined. We are only limited by the number of counters and relays.

This means that the machine is roughly capable of modern SQL queries, including `SELECT`, `WHERE`, `GROUP BY`, and `ORDER BY`.

Following the machine's widely reported success in the 1890 census, Hollerith incorporated the Tabulating Machine Company in 1896. It became the Computing-Tabulating-Recording Company in 1911, then International Business Machines (IBM) in 1924. IBM was described as doing "supercomputing" by the *New York World* newspaper in 1931 and performed similar commercial big data analytics for many governments and companies before 1936. It continues to do so today.

Electromechanical Differential Analyzers

Analog mechanical differential analyzers reached widespread practical use when it became possible to power them using electricity. Electrical circuits also provided a major new application for differential analyzers, as they are

often described using the same kinds of differential equations as used in mechanics. Hazen and Bush's 1928 system, built at MIT, is often credited for the mass popularization of electromechanical differential analyzers, and its concept quickly spread to research teams at the universities of Manchester and Cambridge (Figure 1-19) in the UK. Some of these British research machines were built using Meccano (similar to an Erector Set) on smaller budgets than the American versions.



Figure 1-19: Maurice Wilkes (right) with the mechanical Cambridge Differential Analyzer, 1937

Similar machines were used heavily throughout World War II to solve differential equations, such as when calculating projectile trajectories. By attaching pens to the machines' moving parts, some teams added analog plotters to draw graphs on paper. Versions of these machines were still used in the 1970s as onboard missile guidance systems.

Electromechanical Machines of World War II

Many popular histories focus on machines used during World War II for *cryptography*, the enciphering and deciphering of messages by a transmitter and receiver, and *cryptanalysis*, the cracking of ciphers. Together, these fields are known as *cryptology*. Cracking ciphers is harder than encrypting and decrypting them. Thus, cryptanalysis machines are the larger, more interesting ones. Should any of the machines from either or both categories qualify as "computers"? Their history has been concealed by government secrecy, and we're still learning more as documents are made public. This uncertainty has been useful for some biased historians and filmmakers who want their own country or community to have invented the computer.

The original Enigma (Figure 1-20) was a 1923 electromechanical German commercial cryptography product sold to banks and governments in many countries, including America and Britain.

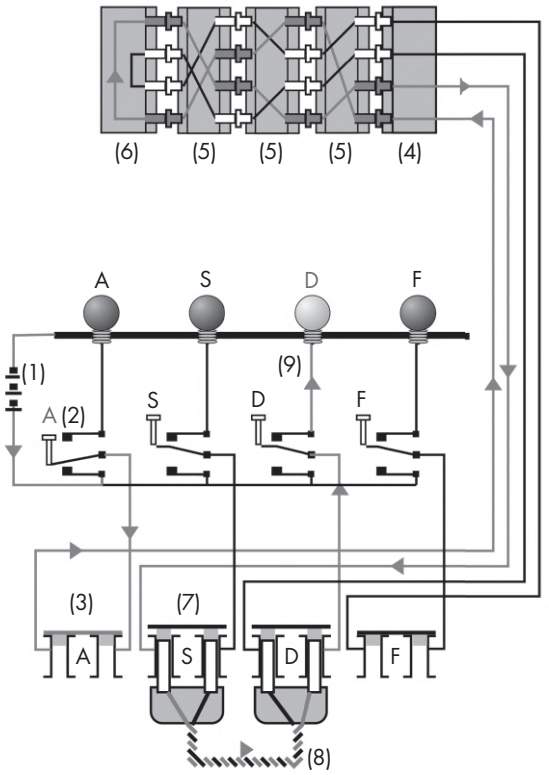


Figure 1-20: The German Enigma wiring, showing four input keys (2), four output lamps (9), three rotors (5, 5, 5), a plugboard (8), and a reflector (6)

The Enigma consists of a typewriter keyboard, output letter lamps, three rotors, and electric wiring. Each rotor acts to substitute a letter for each other letter. The input letter a is passed through the three rotors in sequence, then "reflected" (substituted for $26 - a$) and passed backward through the three rotors again. Each time this is done, the end rotor advances by 1, with carries between rotors, as in Pascal's calculator. Each configuration of a rotor produces a particular set of substitutions. All Enigma operations were symmetric: the same machine state would perform decryption on its own encrypted text. Several versions of the machine were used in the war.

The German military M3 Enigma added a stage swapping pairs of letters using a plugboard. Seven years before the war, the Polish, led by Marian Rejewski, broke its encryption by designing and using a single-purpose electromechanical machine, the *Bomba*. This incorporated physical Enigma rotors to brute-force all possible encodings of known message headers in advance. The daily keys were then looked up in a reverse-index file-card database. The Polish gave this system to the British at Bletchley Park (which later became GCHQ).

In 1938, the Germans changed protocol—not hardware—to remove the known message headers. The Polish mathematician and cryptologist Henryk Zygalski then broke the Enigma again, using optical computing. Information was transferred to punch cards, and the cards were stacked and held up to a light to very quickly find the locations where the light passes through the whole stack.

In 1939, the Germans increased the number of possible rotors to insert into the three slots from three to five. This increased the complexity beyond what Zygalski's method could break. To break this version, the British switched to IBM Hollerith machines to perform similar computations at higher speeds.

Dolphin was a stronger M3 protocol used by U-boats, including more swappable rotors and different headers. The *British Bombe* was designed based on the Polish Bomba and updated for the new task. The additional cryptology was done by Alan Turing, Gordon Welchman, and others, then the machine was designed and manufactured by Harold Keen of IBM.

Typex was the British version of Enigma. Like the Germans, they made their own modifications to the commercial Enigma for their military communications. Typex was broken frequently by the B-Dienst—the German equivalent of Bletchley Park—using IBM Hollerith machines.

In 1937, IBM president Thomas Watson met Hitler and received an award for the Hollerith machines' "services to the Reich." Hollerith machines were later leased from IBM by German concentration camps to enable the Holocaust's precision—"timing so precise the victims were able to walk right out of the boxcar and into a waiting gas chamber." They were used to merge big data sources such as census and medical records to produce lists of names and statuses of victims. IBM provided IT consultants to help with the software design, and to make monthly visits to service the machines on site.

The Zuse Z3

Konrad Zuse was a German engineer who collaborated with the Nazi Party to build the Z3 machine for its military in 1941. The *Z3* was an electromechanical machine using 2,000 electromechanical relay switches and a mechanical binary memory with 64 addresses of 22 bits. It could run up to 10 instructions per second.

In 1998, the Z3 was shown to be theoretically a Church computer, but only via a very obscure and impractical technicality. It could also potentially have very slowly simulated a von Neumann machine, but it was not used to do this.

The Electrical Age

Vacuum tubes (aka *valves*) were invented in 1904 by John Fleming as an efficient replacement for relays. Unlike relays, they have no moving parts; they're purely electrical, meaning they can switch faster than their electromechanical counterparts. They're still used today in analog audio amplification, such as in tube or valve guitar amplifiers (Figure 1-21).



Figure 1-21: A guitar amplifier made with vacuum tubes

A vacuum tube looks and works like an Edison light bulb. A vacuum is created in a sealed glass tube. Inside the tube are three components: an anode, a cathode, and a heater. The anode and cathode are the terminals of the electrical circuit that is being switched on and off, so they have positive and negative voltages, respectively. The heater is the switch. When the heater is turned on, the heat allows electrons to escape from the cathode and travel through the vacuum to the anode, enabling current to flow and switching on the circuit. When the heater is turned off, electrons no longer have enough energy to do this, so the circuit is switched off.

When we restrict the heater to being either on or off, we have a digital switch that functions like a relay, forming a basic unit of purely electrical computation. (Alternatively, for audio and other signals amplification, we may allow the heater to have a continuum of heat levels, which cause a continuum of current sizes to flow in the main circuit, creating an analog amplification effect: the small heater control current turns a much larger main circuit current up and down.)

Pure Electronic Cryptology of World War II

Pure electronic machines appeared later in World War II than the more famous electromechanical ones. They have also been shrouded in secrecy but are sometimes argued to be the "first computers."

In 1942, the German naval Enigma was upgraded to use four instead of three rotor slots (called the "M4 model" by the Germans; its traffic was called "Shark" by the Allies). Brute-force cracking this level of cryptographic

complexity required the American approach of throwing money at computing power by paying IBM to produce hundreds of new, fast, fully electronic and vacuum tube–based *American Bombes*.

Fish was a cipher produced by a different German cryptography machine, the Lorenz SZ42; this was not an Enigma, but it used similar rotors. It was discovered by the Allies later in the war than Enigma because its traffic was initially sent only over landline telegraph wires rather than radio, making it harder to intercept. It was broken by a Bletchley team led by Max Newman, using the *Colossus* machine designed and built by Tommy Flowers and his team in 1944, shown in Figure 1-22.

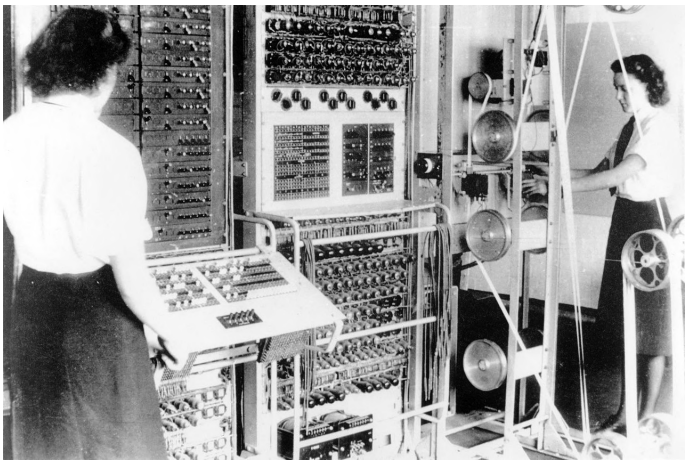


Figure 1-22: Colossus, Bletchley Park, 1943, with operators Dorothy Du Boisson and Elsie Booker

Colossus was a fully electronic, vacuum tube–based machine, like the American Bombes, but it was also able to perform different functionalities if physically rewired for them. The British continued to use Colossus to break Russian codes up to the 1960s. Like the Z3, Colossus was only recently shown to be theoretically a Church computer, but only in a convoluted, speculative configuration requiring 10 machines wired together and programmed with a novel virtual machine (VM), which was not done at the time.

ENIAC

ENIAC (Electronic Numerical Integrator and Computer) was an American vacuum tube machine developed by John Mauchly and J. Presper Eckert in the final years of World War II. It was completed in 1945 and used by the US military for ballistics calculations. It remained in service after the war, doing hydrogen bomb calculations.

Mauchly and Eckert were explicit in basing their design on Babbage's Analytical Engine, translating each of its mechanical components into equivalent vacuum tubes. Like the Analytical Engine, this gives a fully

general-purpose machine that can be programmed to execute arbitrary programs of instructions.

ENIAC was programmed by physically patching cables into sockets on its panels, as is sometimes still done today to "program" electronic synthesizer "patches." Original photographs of its programmers writing programs in this way (Figure 1-23) were sometimes mistaken for technicians simply maintaining the machine or setting it up to run programs written by other people. We now understand that this is how the actual programming itself was done and that these pictures show the actual programmers at work. As in Lovelace and Babbage's time, and Bletchley's, it was assumed that programming was "women's work" and hardware was "men's work."

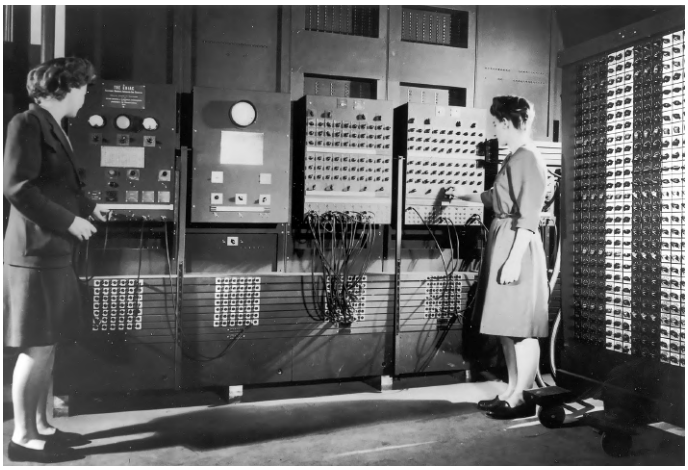


Figure 1-23: ENIAC and programmers Betty Jean Jennings and Frances Bilas at work in the 1940s

ENIAC can run any program (given enough memory), but like the Analytical Engine, it has a Harvard architecture; some might argue that the need to physically patch programs limits its claim to being the first computer. As with many other machines, we could reply that, in theory, someone could have programmed a VM to work around this problem. It was only recently that computer historians rediscovered that someone actually did this for ENIAC!

Virtual Machine ENIAC

The ENIAC programmers Betty Jean Jennings, Marlyn Wescoff, Ruth Lichterman, Betty Snyder, Frances Bilas, and Kay McNulty eventually got tired of programming ENIAC by physically rewiring cables for each new program. So, as a quick hack, they designed a program with these wires that allowed the client program to be read from a panel of switches instead. This created a virtual machine in which a single fixed hardware program emulated a computer that could read higher-level programs from the switches.

Some people argue that "the first computer" was created at this moment, as a software rather than a hardware creation. This would be a beautiful story, but there's still a problem: the architecture is still a Harvard architecture because the user program is stored in the physical switches and not in the computer's main memory. This means that a program couldn't modify its own code, which some people see as a requirement for the "first computer."

The ability for a program to modify its own code is a fairly obscure requirement, rarely necessary outside of a few unsavory security applications and obfuscated coding contests. In theory, the ENIAC programmers *could* have continued to create a second layer of VM, which could have represented higher-level programs in the data rather than program memory. That would have created a von Neumann architecture, with programs capable of modifying their own code using the same VM idea the programmers had already invented. But they never felt the need to do this. Detractors argue that the *potential* for the ENIAC programmers to have done this is no more of a claim of "first computer" status than the potential for a Z3 programmer to have built VMs, and so they assert the virtual ENIAC missed being the first computer by a gnat's whisker.

NOTE *Speaking of gnats, the world's first computer "bug"–and the origin of the modern use of the word–was caught and logged in 1947 by the programmers of another machine, the Harvard Mark II. It was a moth that had gotten stuck inside the machine, causing it to malfunction.*

The Manchester Baby

In 1948, Frederic Williams, Tom Kilburn, and Geoff Tootill demonstrated the first "electronic stored-program computer" at what is now the University of Manchester. *Stored program* means what we now call a von Neumann architecture. The machine was officially named the Small-Scale Experimental Machine and nicknamed "the Baby" (Figure 1-24).

The Baby's CPU used around 500 vacuum tubes, together with diodes and other components. It implemented an instruction set of seven instructions. In modern terms, the Baby was a 32-bit machine, with 32 addresses each storing one 32-bit word.

The Baby was built from parts including the then broken-up Bletchley Colossus machines; it was quickly scrapped and cannibalized itself to provide parts for the later Manchester Mark I machine. A replica of the Baby can be seen today in Manchester's Science and Industry Museum. This museum is especially interesting, as it also contains textile processing machines from the industrial revolution, which began in Manchester. These machines form a cultural connection between the Jacquard loom and the Manchester computers.

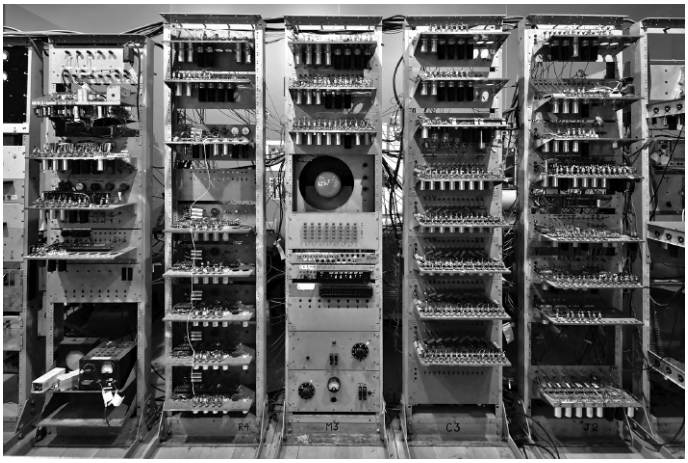


Figure 1-24: The Manchester Baby rebuilt in Manchester's Science and Industry Museum, UK. Note the CRT memory in the center, also used as a display.

The Baby can probably be programmed to play *Space Invaders* on its green CRT screen: since the modern rebuild, similar games have been demonstrated for it both in emulation and on the real machine in what is perhaps the most extreme example of retro gaming.

Having a von Neumann architecture, the Baby is also able to run programs that modify their own code. Thus, by the time we reach the Baby, we appear to have an indisputable Church computer, as long as we're happy that it could be "given as much memory as it asks for." It's not trivial to wonder how that could be done, though, as the Baby's architecture is so specific to the 32×32-bit memory design. You *could* redesign it with a larger memory, but would that really be the same Baby, or a different machine?

The 1950s and Commercial Computing

UNIVAC (Universal Automatic Computer; Figure 1-25) was delivered to its first customer in March 1951. It was Mauchly and Eckert's commercialized version of their previous ENIAC, making it the first *commercial* general-purpose stored-program computer. Like ENIAC, UNIVAC was vacuum tube–based. CBS used one to make a successful statistical prediction of the US presidential election of 1952, which brought fame and sales. Mauchly and Eckert's company still exists as the modern Unisys Corporation.



Figure 1-25: UNIVAC

IBM was slow to understand that UNIVAC and other electronic computers would destroy their still-profitable tabulating machines business, with CEO Thomas Watson making the worst futurology prediction in human history in 1948: "I think there is a world market for about five computers." After waking up to the new technology, IBM produced its own first commercial electronic computer in 1952, the IBM 701.

The Transistor Age

A *transistor* performs the same function as a vacuum tube, but it's smaller, faster, and cheaper, and it consumes less power and is more reliable. Like tubes, transistors can be used for both analog and digital tasks (they're found in analog audio amplifiers such as transistor radios and guitar amps), but for computing, they're used only for their digital properties.



Figure 1-26: A big transistor

William Shockley, John Bardeen, and Walter Brattain discovered the transistor effect in 1947 and were awarded the Nobel Prize in Physics for it in 1956. Work to commercialize transistors began in the 1950s in what is now Silicon Valley, and the technology became mainstream in the 1960s. Transistors remain the basic technology of computers today.

The 1960s and Big Transistors

The transistor "minicomputers" of the 1960s didn't use microchips, but instead were made from the "big" kinds of transistors, about 1 cm long, that you would put in a breadboard circuit today (Figure 1-26). It's still possible to make a CPU out of such transistors, and a few hobbyists do it for fun (for

example, the MOnSter 6502 project by Eric Schlaepfer and Evil Mad Scientist Laboratories).

These computers filled a rack and included the classic PDP machines (Figure 1-27) used heavily in early AI research. This was also the time when Seymour Cray began building Cray supercomputers, aiming to make the biggest and fastest machines for high-end users.



Figure 1-27: A transistor-based 1960s PDP-11 mini-computer

Uses of transistor computers in the 1960s included powering ARPANET, the predecessor of today's TCP/IP-based internet, and Margaret Hamilton's 1969 programming of the Apollo moon landing code in assembly language (Figure 1-28). The latter was actual rocket science, and required her to create the modern field of software engineering while searching for ways to make this highly critical code more correct.



Figure 1-28: Hamilton with a printout of her complete assembly program for Apollo 11

In 1965, Gordon Moore, the CEO of Intel, made an observation known since as Moore's law. As you saw in the introduction, depending on who you ask and how you count, this law says that either the speed of computers or the number of transistors per area doubles every 18 months or every 2 years.

The 1970s and Integrated Circuits

The 1970s saw the widespread commercialization of *integrated circuits* (also called *ICs*, *microchips*, or *chips*). ICs had been theorized in 1952 in Britain by Geoffrey Dummer, though the 2000 Nobel Prize in Physics was awarded to Jack Kilby—who had heard Dummer talk about them in 1952—for his invention and patent of a practical version in 1958 at Texas Instruments.

IC technology allows electric transistor-based circuits to be miniaturized, so that the same wiring that filled a 1960s rack cabinet can fit on a "chip" of silicon the size of a fingernail. From an architectural view, chips are not very exciting—if you take the wiring diagram from a 1940s vacuum tube machine and just miniaturize it, then you get a chip. If you look at a chip through a microscope, you'll see similar wiring patterns to, say, the wires on the back of a 1940s, 1950s, or 1960s rack. The silicon chip is then "packaged" inside a larger, usually black lump of plastic, with larger metal pins connecting the fine inputs and outputs of the chip to the outside world, usually a printed circuit board (Figure 1-29).



Figure 1-29: An Intel 4004 chip in its packaging

The 1970s saw the birth of some of the oldest software that is still in use today. The UNIX operating system was built by Kenneth Thompson and Dennis Ritchie in this time (Figure 1-30) and has evolved into current Linux, FreeBSD, and macOS systems.



Figure 1-30: Thompson and Ritchie creating UNIX on a teletype terminal

UNIX terminals of the time used typewriter-style print heads on paper rolls—like Babbage's Difference Engine—and programmers would interact with the machine by typing commands on a keyboard; these commands were printed as they typed, along with their resulting outputs. This teletype system is the origin of the x-terminals used today in UNIX-like systems.

In contrast to terminal-based interaction, Xerox (the photocopier company) researched graphical user interfaces at its Palo Alto Research Center, Xerox PARC. This included developing the first mouse, as well as the "desktop" metaphor, including files and folders based on physical filing cabinets.

This choice to base the interface with a computer on a middle management office, with its desks and filing cabinets—rather than on, say, a school, art gallery, or shop—has been with us, and making computing more boring than it should be, ever since. This may be starting to change, with the rise of handheld interfaces such as Android and TV-based "10-foot" interfaces such as Kodi, which provide feasible alternatives based on "apps."

The 1980s Golden Age

Any author covering computer history eventually reaches a point where the story overlaps with their own lifetime, and from then on, they may become somewhat biased. For this author, it occurs here, so you might want to find alternative accounts from others to balance mine out.

The 1980s was the golden age of computer architecture: for the first time, electronic computers became cheap and small enough to be mass-produced and bought by normal people to use in their homes. As shown in Figure 1-31, this may have been the best time in human history to be a kid interested in computers because you would get a proper computer for Christmas, with direct access to its architecture, at a time before operating systems hid the architecture away from the user.



Figure 1-31: Home computing in the 1980s: a happy child with their first computer

These machines were based initially on 8-bit CPUs, such as the 6502 used in the Commodore 64 and Apple II, and then based on 16-bit CPUs, such as the Motorola 68000 used in the Amiga and Atari ST. This period—especially in retro gaming—is known as the 8-bit era and then later the 16-bit era; it's looked back on with fondness and nostalgia by many who were there, and by many who weren't.

The IBM 5150 PC launched in 1981, based on the Intel 8088 chip. IBM and others sold this and other PCs during the 1980s for use in business offices. The PC concept is the polar opposite of the heterogeneous, architecture-driven home computer market for two reasons. First, it enforces a standardized architecture on the computer components so that multiple manufacturers can produce them to be compatible with one another. Second, it wraps all the hardware under a strict operating system, which controls all access to it via a standardized interface. IBM could use its

market clout to enforce standards on components, so it could buy them from the cheapest suppliers and make money by stamping its brand on assembled PCs.

In a reaction to the proprietary operating systems being installed on PCs and larger computers, the GNU (recursively standing for "GNU's Not Unix") project and Free Software movement were created in this decade by Richard Stallman—this later led to the Linux-based systems and philosophies that we use today.

We will study this period in more detail in Chapter 11.

The Bland 1990s

The 1990s was a bland, boring, beige decade. It was driven by a commercial focus in the industry that switched from treating users as programmers and community members to users as customers and consumers of software products, such as word processors and spreadsheets. During this time, schools stopped teaching computer science and (under the heavy influence of corporate lobbying by their creators) taught the use of commercial office software.

Computer architecture became dominated by the personal computer (PC) standard architecture, which had been used in office computing during the 1980s but was now pushed everywhere by the PC corporations, including on homes and schools. Closed source operating systems were pushed as part of the PC package, making it hard for users to see anything "under the hood" of their machines.

Physically, these machines appeared as nearly identical "beige boxes," as in Figure 1-32, and the general drabness of this middle management–style computing culture was later caricatured through Apple's "I'm a PC" TV commercials, which portrayed the PC as a generic middle manager with a boring beige outfit.

As Moore's law reliably predicted, processor speeds doubled every 18 months; this was the standard measure of how good your computer was, and many would build a new one every couple of years to take advantage of the new speed.



Figure 1-32: A 1990s desktop

Related to the move to operating systems was the move from programming in assembly and interpreted languages, such as BASIC, to compiled languages. When languages are compiled, their authors can choose to conceal the source code so that users can no longer see how they work or learn from them by changing them. Compilers had been developed since Grace Hopper's work in the 1950s, and were used in high-end computing, but this was the first time they and their generated code arrived in homes.

The computer games industry similarly became professionalized, separating consumers, who could only buy and play dedicated consoles and

games, from commercial developers with the money to access specialist programming tools. Games were sometimes fun to play, but not as much fun as they used to be to write.

The World Wide Web went online at CERN in 1990 and grew in popularity, leading to the dot-com investment craze at the end of the decade. As more hackers and eventually consumers joined the web, dedicated rack-mounted server computer designs became popular, beginning with the Compaq ProLiant in 1993. Like the Manchester Baby and 1960s minicomputers, these were designed to be stacked in 19-inch rack units, but to be always on with high reliability.

For the early modem-connected elite, 1993 also saw the birth of Linux and the beginnings of its GNU-inspired international authors figuring out how to communicate and code with one another at the level of architecture and systems programming.

The 2000s and Reconnecting the Community

The PC architecture of commodity components plus operating system continued throughout the 2000s. Moore's law, and the consequent building or buying of a new doubled-speed computer every couple of years, continued. Machines used the same basic PC computer design, with various interfaces and components getting upgraded for speed. Internet speeds also increased, enabling streaming of videos as well as the transfer of text and images. Servers were reduced in size to *blades*, many of which could be packed together in a single rack unit.

Enabled by these advances, Linux matured into a realistic alternative system to the proprietary operating systems previously bundled with PCs. Many of the people involved in older computing communities returned and joined the Linux movement. We realized that things had to go via the operating system route rather than raw architecture; for free software advocates, this was a good thing: it removed any dependency we had on any particular hardware companies. This was now okay because the operating system was free software and thus no one had to be locked in to buying anyone's specific products. With this hindsight, the 1980s was perhaps not so great because everyone was forced to develop on some non-free architecture platform and was thus utterly dependent on their corporate owners. The 1990s saw a reduction in freedom as a multitude of these corporations and platforms were replaced by a single dominant PC operating system corporation and platform, but since then, Linux life has become even better than the 1980s and 1990s, as we have an open platform and many competing hardware suppliers implementing it.

Much of the other open source software we use today developed rapidly alongside Linux, such as Firefox, Python, MySQL, and Apache. In many cases, these tools have older origins, but they only grew to a critical mass of developers and users in the 2000s.

The programmers working on the Linux operating system itself got to see and work with the underlying architecture, but for everyone else, architecture was generally still under the hood, as in the 1990s.

The 2010s and the End of Moore's Law

During the 1990s and 2000s we happily assumed that the clock speeds of our processors would double every couple of years—and they did. Moore's law became a self-fulfilling prophecy as Silicon Valley chipmakers used it as a target to be reached.

However, this all fell apart in the 2010s. Transistor manufacturing technology did continue to double the number of transistors per area, but clock speeds maxed out by 2010, at around 3.5 GHz. Suddenly, processors weren't getting faster anymore. This is due to the fundamental laws of physics around computation speed and heat. During the Moore's law period, the temperature of processors had also been rising along with speed; larger and more powerful fans and other cooling systems such as water cooling were needed. The transistors got smaller, but the fans got bigger. If this trend had continued through the 2010s, we would now have processors hotter than the surface of the sun.

A closely linked concept is power consumption. As chips give off more heat, they consume more power, and this decade also saw the beginnings of a push toward lower-power, more portable computing, especially in the form of smartphones. This was the decade when we switched from looking up to the sky to looking down at screens in our hands.

As mentioned in the introduction, the end of Moore's law has created what Turing Award winners John Hennessy and David Patterson have described as "a new golden age of architecture." Where the previous two decades saw computer architecture stagnate as a field, relying on advances in fabrication technologies to create regular gains, the field is now wide open again for radically new ideas. We can't make computers faster via the speed form of Moore's law, but we can still fit more and more transistors onto chips with its density form. We can now consider making everything parallel, performing many operations at once, rather than one at a time.

As you might expect, the 2010s were characterized by an explosion of new ideas, architectures, hardware, and software, all to enable parallelization. A key computer science question of our time is how much programmers need to worry about this. In one possible future, programmers will continue to write sequential programs, and new kinds of parallel compilers will figure out how to turn step-by-step instructions into parallel executions. In another future, we might find this is not possible, and programmers will have to write explicitly parallel programs themselves. This will completely change the nature of programming and the kinds of skills and thought processes that programmers need.

While there remain many parallel architectures still to be explored—and hundreds of university researchers and startup companies now trying to explore and exploit them—the 2010s saw three major new types of parallel architecture succeeding in the real world.

First, and most basically, *multicore* processors are simply chips manufactured to contain more than one copy of a CPU design. The decade began with duo-core systems and progressed through quad, eight, and even more cores. If you were to run just a single program on these machines, then the

programmer would have to care about parallelism. But most current computers run an operating system program that in turn enables many programs to run concurrently, sharing the computer's resources between them. A typical desktop machine might run 10 to 20 processes concurrently during normal operation through this arrangement, so adding *N* multicores gives a factor *N* speed up, but only up to this number of processes. Multicores will not scale very well beyond this if they are asked to run ordinary programs.

Second, cluster computing, shown in Figure 1-33, is another form of parallelism in which many conventional single-core or multicore machines are weakly linked together. Computing work is then split into many independent chunks that can each be assigned to a machine. This requires programs to be written in a specific style, based around the split into independent jobs, and works only for certain types of tasks where such splits are possible.



Figure 1-33: A 2010s parallel supercomputing cluster

Cluster computing has been especially useful for "big data" tasks where we usually want to repeat the same processing independently on many data items, and then collate the results (this is known as *map-reduce*). The Search for Extraterrestrial Intelligence project (SETI@home) pioneered this approach in the 1990s, using compute time on millions of home computers donated by their users to run in the background, analyzing big data from radio telescopes to look for alien messages. The method is also used by search engine companies: for example, a company might assign one commodity Dell PC out of many in a large warehouse to be responsible for storing all the locations on the web containing one particular word, and handling queries about that word. During the 2010s, the underlying map-reduce process was abstracted and open sourced by the Hadoop and Spark projects, which enabled everyone to easily set up and use similar clusters.

The third approach, and most interesting architecturally, has been the evolution of graphics cards (also called graphics processing units, or GPUs) into general-purpose parallel computing devices. This presents a completely new silicon-level design concept that also requires a new style of programming, somewhat similar to cluster programming. Now that its graphical

roots have been left behind, the concept is continually evolving into many novel architectures, such as recent tensor and neural processing units found on mobile phones.

It's not clear whether the concept of a "programmer" will survive if some of these new parallel architectures become dominant; for example, we might "program" machines by creating specific parallel circuits in hardware, where everything happens at the same time, rather than thinking of a "program" as a set of instructions to be run in series.

The 2020s, the Cloud, and the Internet of Things

This is the current decade at the time of writing, so any trends identified are somewhat speculative. With that said, the systems that we can see in development labs today suggest that the present decade will see a fundamental split of architectures into two main types.

First, increasingly small and cheap devices will be embedded into more and more objects in the real world. This concept, known as the *Internet of Things (IoT)*, promises to see smart sensors and computers in cities, factories, farms, homes, and pretty much everywhere else.

"Smart cities" will be covered in these devices to enable the monitoring of every individual vehicle and pedestrian, to make traffic control and use of city facilities more efficient. "Smart factories" will have tiny devices attached to every item of stock and track them through the manufacturing process. Smart transport, retail, and homes will track the same items right through their supply chains, "from farm to fork" in the case of food. For example, your fridge will sense that you're running out of cheese, using either the weight of your cheesebox or machine vision looking for cheese, and automatically place an order to your local supermarket to replenish it. The supermarket will aggregate these orders and balance the demand with orders from their distribution centers. Small autonomous robots will then deliver your cheese from the supermarket to your doorstep.

The second trend is in the opposite direction. The low-power IoT devices won't do much computing, but will instead exist primarily to collect and act upon "big data" in the world. This data will then be processed on massive scales in dedicated computing centers: buildings the size of warehouses that are packed with computing power.

Computing centers are related to *data centers*, similar-looking buildings already in existence that exist primarily to *store* data and make it available over the web, rather than to perform heavy computation on it. This type of computing appeared first at search engine companies, which used many cheap commodity PCs running together to process web crawls and searches. Search companies, and their online shopping peers, discovered they could make a profit by hiring out the spare machines that were sitting idle for general computing use by customers. This style of computing is quite like the big machines of the 1960s and 1970s, whose users would dial in from terminals and share time on them. (Perhaps Thomas Watson's guess that there is a world market for only five computers will actually turn out to be true if we

count each of these cloud computing centers as one computer and ignore the IoT devices.)

The IoT devices create a particular interest in low-energy design, but related energy issues also occur in huge cloud computing centers. These buildings can use as much power as factories, give off significant heat, and cost a significant amount to run. Computing centers powered most of the world's video calls and collaboration tools during the COVID-19 pandemic, enabling many jobs to switch to remote work for the first time. Some computing centers saw shutdowns in 2022 due to an extreme heatwave. Recently, some computing centers have been deliberately located in places such as the Arctic to make use of the natural cooling.

So, like Moses, in this decade we will download from the cloud onto our tablets. The two trends of the IoT and the cloud are likely to continue and become more extreme during the 2020s, pulling architecture in two opposite directions. Medium-sized desktop computers seem likely to fall in importance.

Already we're getting used to computing on physically small devices such as tablet computers and the Nintendo Switch, which are starting to make larger desktop machines look a bit silly. "A computer on every desk" was the aim in the 1990s, but these are disappearing and being replaced by a mix of computers in our pockets, streets, and cloud centers. Similar setups have been suggested previously from time to time, including 1950s dial-in mainframes and 1990s "thin clients," but in the 2020s they seem to be taking off via mobile phones, Amazon Echo, Nest home automation, and Arduinos, as well as cloud providers such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

So Who Invented the Computer?

The modern concept of computation was defined by Church. Commercial electronic machines of the 1950s, beginning with UNIVAC, through 1960s minicomputers and 1970s microchips up to the present day seem clearly recognizable as computers. But should anything before them be credited as "the first computer"?

The Manchester Baby is a Church computer if you are happy that it could be "given as much memory as it asks for," but it's not very clear how this would be done. Looking at later commercial machines gives more of a feeling that they could easily be extended with more memory, for example, by plugging in extra circuit boards or hard disks. But in principle they all still have the same problem as the Baby.

ENIAC-initial has the potential to be a Church computer if programmed in a certain VM way. ENIAC-VM actually *was* programmed that way, but was still a Harvard architecture. It needed another layer of unrealized VM to get to RAM programs. Colossus and Zuse Z3 programmers could theoretically have done all of this, too—but didn't. The same goes for Analytical Engine programmers.

IBM has been doing big data analytics on machines described by the media as "supercomputers" since the 1890s, but data analytics isn't general Church computation unless you can find a way to make any problem look like an SQL query.

People have probably been calculating since 40,000 BCE, with abaci, mechanical calculators, paper, pens, clay tablets, bones, rocks, their fingers, and natural numbers in their heads. All the above are theoretically Church computers because they can simulate any machine if programmed in a certain way. So perhaps we have always had computers—and Church was just the first to notice them.

Summary

In this chapter, we took a whirlwind tour through the history of computing. Beginning with bones and ending in the cloud, we considered a number of inventions that might, or might not, be called a computer. We also saw a few hypotheses for what makes a computer. Initially we suggested that a computer was anything that could be programmed to play *Space Invaders*. We then formalized this hypothesis by looking at Church's thesis, which argues that a computer is a machine that can simulate any other machine, given as much memory as it asks for.

Our survey of the history of computing has briefly introduced the big ideas of architecture. In the next chapters, we'll dive into the details of data representation and CPU computation to see how some of the historical systems work in more detail. This will set us up for Part II's study of modern electronic hierarchy and the many particular modern architectures of Part III.

Exercises

Calculating with an Abacus Simulator

1. Use an abacus simulator (or a real abacus if you have one) and a tutorial to understand abacus arithmetic. These operations are still the basis for some modern CPU operations, and learning to do them on the abacus will help you understand them in CPUs. A simulator can be found here: *https://www.mathematik.uni-marburg.de/~thormae/lectures/ti1/code/abacus/soroban.html* and a tutorial for using it at *https://www.wikihow.com/Use-an-Abacus*.
2. Take the last three digits of your phone number as one number and the preceding three digits as a second number, and add them together on the abacus.
3. Take the same pair of numbers and subtract the smaller one from the larger one.
4. Take the last two digits of your phone number as a two-digit number and the preceding two digits as a second two-digit number, and multiply them using the abacus.

Speculative History

1. How do you think world history could have been different if the Antikythera mechanism had arrived safely in Rome and inspired the Roman Empire to use similar machines?
2. How do you think world history could have been different if the Analytical Engine had been fully constructed and commercialized in the British Empire?

Challenging

Search the internet for examples of advanced operations using an abacus, such as square roots or prime factorization, and try to run them. You may need to use more than one abacus to provide enough columns for some of them.

More Challenging

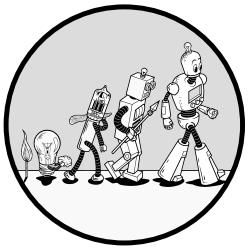
1. Write a speculative fiction short story or novel based on one of the premises raised by the "Speculative History" exercises.
2. How could you implement a Church computer using an abacus?
3. Research the SQL-like functions available on the Hollerith machine. Can a Church computer be made from them?

Further Reading

- For details of the Hollerith machine, see H. Hollerith, "The Electrical Tabulating Machine," *Journal of the Royal Statistical Society* 57, no. 4 (1894): 678–689, *https://www.jstor.org/stable/2979610*.
- For details of Hollerith machines' role in World War II, see Edwin Black, *IBM and the Holocaust: The Strategic Alliance Between Nazi Germany and America's Most Powerful Corporation* (Washington, DC: Dialog Press, 2012).
- To learn more about 2020s IoT computing, see S. Madakam, R. Ramaswamy, and S. Tripathi, "Internet of Things (IoT): A Literature Review," *Journal of Computer and Communications* 3, no. 5 (2015), *http://dx.doi.org/10.4236/jcc.2015.35021*.
- To learn more about 2020s cloud computing, see I. Hashem, I. Yaqoob, N.B. Anuar, et al., "The Rise of 'Big Data' on Cloud Computing: Review and Open Research Issues," *Information Systems* 47 (2015): 98–115.
- For a dieselpunk novel featuring World War II cryptography, see Neal Stephenson, *Cryptonomicon* (New York: Avon, 1999).

2

DATA REPRESENTATION



A computer is a device that represents things in the real world and performs manipulations on these representations. Entities that we might want to represent and make computations about include physical objects, numbers, words, sounds, and pictures. This chapter examines systems for representing each of these types of entities.

We'll begin by exploring the history of how representations of objects, numbers, and text have evolved. We'll then look at the modern symbol systems used to represent numbers—including decimal, binary, and hex—and use number representations to construct representations of further entities, such as text, audio, and video.

In this chapter, the modern representations are built from 0s and 1s, which are left as symbols themselves. In later chapters, we'll consider how to instantiate these zero and one symbols in digital electronics and make use of them in computations.

A Brief History of Data Representations

The concepts of representation and computation are closely linked. Humans often need to represent the state of part of their world, as an aid to their own memories or as proof to other humans that something has happened or is going to happen. Once you have a representation, you can also use it to perform computations, to simulate what would happen if certain actions were done, or to deduce conclusions from what is known.

For example, we often need to keep track of who owns what and who has debts to be paid. Static representations are useful for these purposes, and once these exist we can use them in computations to answer questions such as what would happen if we bought something, or how long it will take to repay a debt. Representation thus comes before computation, both conceptually and historically. Let's trace how it evolved, from humanity's first efforts to the symbol systems that we use today.

Tally Sticks and Trading Tokens

The oldest-known data representation is the use of *tally sticks*, such as the Lebombo bone shown in Chapter 1. These are simple sticks with several marks, where each mark represents one object. For example, the number 13 is represented by 13 marks, usually made in a row, as in Figure 2-1.

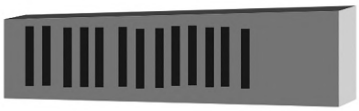


Figure 2-1: A simple tally

By Sumerian times (4000 BCE), physical tokens were used to represent objects, as in Figure 2-2. A small clay model of an animal, for example, represented the actual animal and could probably have been exchanged for it. This would have simplified trading, as you could travel from the city of Ur to the city of Uruk with 10 animal tokens and make a deal by swapping them for, say, 20 beer tokens, with the actual objects only being moved around later, after a successful deal. These tokens could also have been divided between groups of people or given as tax to the king.



Figure 2-2: Sumerian trading tokens

Computing with tally sticks and tokens is, however, quite slow. To add m tallies or tokens to n, you have to go through the process of adding each of the m into the n, one at a time. If you've studied complexity theory, this means that addition is order $O(m)$ in the size of the numbers being added.

By 3000 BCE—still before the abacus—the Sumerians sped up their calculations by sealing many tokens in a clay "envelope" called a *bulla*, shown in Figure 2-3. The bulla was sealed both physically, by joining its clay to encapsulate the contents, and informationally, by stamping a complex, unforgeable mark onto it. (This is the origin of ceremonial seals still used today on royal and governmental documents, such as the Great Seal of the USA. It's also the origin of later digital signatures.) The seal guaranteed, probably in the name of the king or another powerful and trustworthy person, that a certain number of tokens were contained within it. This way, instead of counting out 12 animal tokens, you could hand over a bulla of 12 animals at a time. The bulla would function like a 12-token coin or banknote, but one that physically contained the 12 tokens inside.



Figure 2-3: A bulla

A similar development to the bulla is found in tally sticks from this period, where tally marks started to be grouped together, as shown in Figure 2-4. Counting out n scratches usually requires n operations, but if we replace, say, every fifth vertical stroke with a diagonal one going through the previous four, we can quickly count how many groups of five we have.



Figure 2-4: A grouped tally

Roman Numerals

In a closely related notation to grouped tallies, we can replace the fifth stroke with two shorter diagonal strokes to make a V, and the tenth with an X, as in Figure 2-5, forming the beginnings of the *Roman numerals*.

Roman numerals developed further to closely represent the human perception of numbers. Humans appear to perceive the sizes of sets of 1, 2, 3, and 4 objects directly and immediately. Beyond this, our immediate perception is of *numerosity* or approximate size rather than exact number, based on sizes roughly around 5, 10, 20, 50, 100, and 1,000. Most number symbol alphabets reflect this, with Egyptian,



Figure 2-5: Early Roman numerals

Chinese, and Arabic numbers having special symbols for 1, 2, 3, and 4 that feature the corresponding number of strokes, and more abstract symbols for 5 and above. The Roman numeral system also uses symbols to represent the "landmark" numbers, such as V = 5, X = 10, L = 50, C = 100, and M = 1,000, with smaller-valued symbols positioned before or after to indicate adjustments to the landmark, such as VI = 6 and IX = 9.

Roman numerals have the advantage of being a close model of how humans actually think about numbers, but if you try to do large-scale arithmetic such as adding and multiplying with them, you'll quickly run into difficulty. This is a classic example of how the choice of representation can heavily affect your ability to do certain types of computation.

Split Tallies

A *split tally* was a variation of the tally stick, in which the stick was marked up, then split into two pieces down its length, as in Figure 2-6.



Figure 2-6: A split tally

Both halves contain parts of the same notches, and both halves could be reunited to show they were genuine and fitted together. They were used to record loans, with the long and short halves (the *stock* and *foil*) given to the lender and borrower, the origin of our modern financial *long* and *short* positions in stocks. The British government continued to use split tally sticks until it burned its last wooden tallies to modernize its IT systems around 1836, the time of Babbage's Analytical Engine.

Arabic and Other Numerals

Other civilizations developed number representations using copies of symbols for large numbers, as shown in Figure 2-7.

For example, the ancient Egyptians had symbols for 10, 100, 1,000, and 10,000. The number 23 would be shown using two copies of the 10 symbol (a heel) and three copies of the 1 symbol (a tally stroke). The number 354,000 would be shown using three copies of the 100,000 symbol (a tadpole), five copies of the 10,000 symbol (a finger), and four copies of the 1,000 symbol (a lotus flower).

Eastern Arabic numbers appeared in the Islamic Golden Age, based on an earlier Indian system from around 500 CE. This system introduced the base-exponent method that we use today, with fixed columns containing symbols for numbers of 1s, 10s, 100s, 1,000s, and so on. Importantly, this introduced the need for a concept and symbol of zero to fill in columns having no counts, which is missing from ancient Egyptian and similar systems. These symbols evolved into the Arabic numerals (1, 2, 3, and so on) used in the West today.

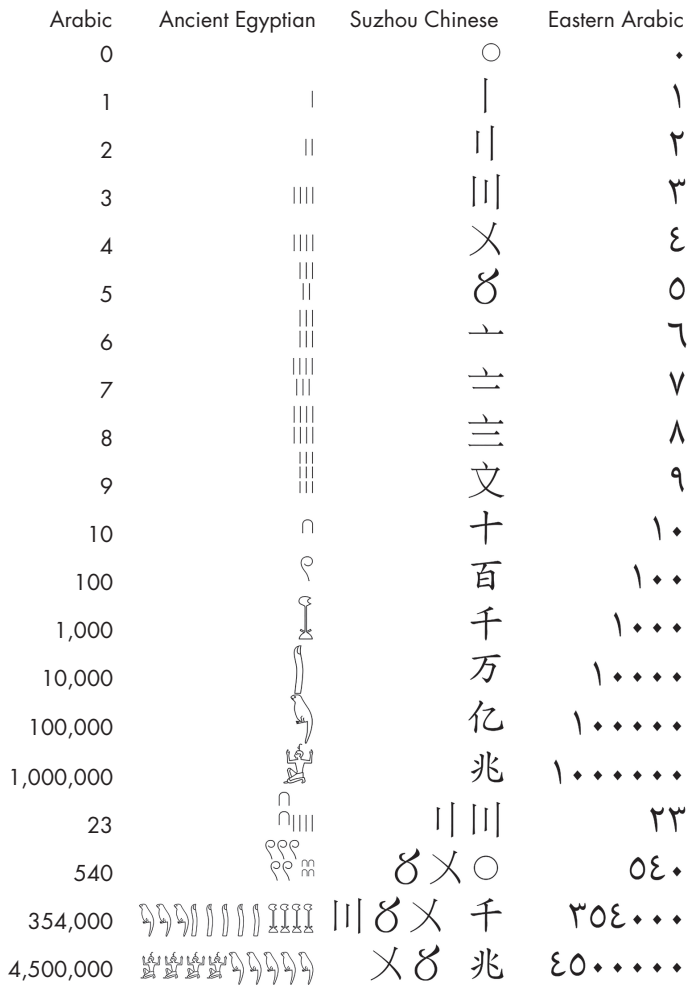


Figure 2-7: Modern Arabic, Ancient Egyptian, Suzhou Chinese, and Eastern Arabic numerals

Suzhou Chinese numerals evolved from ancient Chinese symbols relating to the base 10 abacus seen previously in Figure 1-4, and are occasionally still in use today. You can see the symbols for 1 to 4 are based on tally strokes, while those from 5 to 9 are similar symbols placed under a "bead" for 5. For a few significant digits, Suzhou uses a column system similar to Arabic numerals. For larger numbers, however, it uses a more advanced representation that shows the first few significant digits, followed by a separate symbol denoting what power of 10 they are multiplied by. In English we sometimes do this by writing *354 thousand* or *354k* rather than *354,000*.

This history of number *representation* belongs more properly to computer science than to mathematics. We can see that, historically, *typed* quantities such as "five cows plus three cows" were represented and computed with before more abstract mathematical number concepts such as "five plus three." Mathematics takes numbers for granted and performs proofs about

their properties. By contrast, the business of representation, both of actual objects and of abstract number concepts derived from them, is computer science, as is the question of how to build algorithms and machines based on these representations.

Modern Number Systems

We've seen how our modern concept of numbers evolved from tallies into the symbolic, Arabic system used in everyday life today. The key innovation of the Arabic system is the use of columns to represent digits in a base. This (as we'll see when we start computing) makes algorithmic arithmetic easier, and also reduces the size of representations. For example, you only need four symbols to represent the number 2,021, rather than 2,021 clay tokens.

Our everyday Arabic numbers are decimal, using base 10, but this isn't necessarily the base for computers. This section generalizes the idea of bases and exponents and presents several related systems that are useful in computers.

Bases and Exponents

We will make heavy use of exponentiation in representing numbers. *Exponentiation* is the repeated multiplication of a *base*, such as:

$$2^3 = 2 \times 2 \times 2$$

Here, 2 is the base and 3 is the exponent. This may also be written as 2^3. In some computer languages, it appears as `2**3`, or is written via a power function, such as `pow(2,3)`. Exponentiation is sometimes called "raising to the power," as in "two to the power of three."

More generally, we write a base b to the power of an exponent n as

$$b^n = b \times b \times b \times \ldots \times b$$

meaning there are n copies of b. Zero and negative exponentiation are defined as:

$$b^0 = 1 \text{ and } b^{-n} = \frac{1}{b^n}$$

If we choose a base b, we may then define a *number system* as a mapping from a list of *numeral symbols* to a *number*. Symbols are marks on a piece of paper or entries in some other type of storage system; numbers are the actual mathematical objects being represented.

To write base b numbers, we need an alphabet containing b symbols. Strings of N of these symbols can have b^N different states, which are used to represent numbers from 0 to $b^N - 1$.

When we work with symbols in different bases, we will sometimes use a subscript to indicate what base the symbols are written in. For example, 123_{10} means one hundred and twenty-three in base 10, while 1001_2 means one 8, no 4s, no 2s, and one 1 in base 2 (which equals 9_{10}). In other cases, we'll omit the subscript where the base is clear from the context.

Base 10: Decimal

Everyday arithmetic uses base 10, in which, for example, the string of symbols 7, 4, 3, written as 743, is interpreted as representing the number seven hundred and forty-three. We can see this mathematically using exponents of 10:

$$743 = 7 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$

Using a point notation and negative exponents, we can represent fractional numbers. For example:

$$743.29 = 7 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2}$$

For base 10 we have an alphabet of 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Strings of n symbols from this alphabet can specify one of 10^n numbers; for example, with $n = 4$, there are 10,000 numbers, 0 to 9,999 inclusive.

Base 2: Binary

Base 2 is known as *binary* and is used in almost all modern computers. It has an alphabet of two symbols, usually written as 0 and 1, but sometimes as T and F for *true* and *false*. In electronic computers, the two symbols are represented using high and low voltages. *High* is usually the system's positive voltage, such as 5 V or 3.3 V, while *low* is usually ground or 0 V. Binary is useful for electrical machines because real voltages are noisy, and attempts to include extra symbols such as *medium* have been doomed to failure. But *high* and *low* can more easily and cheaply be separated into two clear classes.

A single symbol in base 2 is called a *bit*, short for *bi*nary dig*it*. A string of N bits can represent one of 2^N numbers, such as ranging from 0 to $2^N - 1$. The columns of the string represent powers of 2. For example:

$$\begin{aligned} 10011101_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 157_{10} \end{aligned}$$

The powers of two that appear in this calculation (0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024, 2,048, and so on) should be instantly recognizable to anyone used to computing. They often appear as sizes of memory capacity and as sizes in bits or bytes of data structures used at the hardware level. If you plan to work at or near the hardware level, you'll need to memorize these powers of two for everyday use.

To convert from binary to decimal, add up the powers of two for each column that has a 1 in it. To convert from decimal to binary, at each step, try to subtract the highest power of two from the decimal, and make a note of which powers of two have been subtracted. Write 1s in those columns and 0s in the others.

Some mathematical operations are faster or slower in different bases. In base 10, you can quickly multiply or divide by 10 by shifting the decimal (radix) point one place to the left or right. Where numbers are represented

in binary, you can use a similar trick to quickly multiply or divide by 2. This is known as *binary shift* and is implemented in hardware by most CPUs. In the C language, for example, a fast multiplication by 8 (2^3) can be done by writing `y=x>>3;`.

Alternate notations used in some books and programming languages for binary include 1110_2, 0b1110, and 1110b.

NOTE *A famous computer science joke says, "There are 10 kinds of computer scientists: those who know nothing, and those who know binary."*

Base 1,000

As a means of introducing other notations—hex and byte codes—let's look at decimal notation in a different way, which we'll call *compound notation*. It's common to write large numbers by grouping them into chunks of three digits separated by commas, to make them easier to read. For example, the number 123,374,743,125 symbolizes the value one hundred and twenty-three *billion*, three hundred and seventy-four *million*, seven hundred and forty-three *thousand*, one hundred and twenty-five. (The "one hundred and twenty-five" at the end refers to the number of *ones*.)

Imagine for a moment that these chunks are individual symbols, from an alphabet containing 1,000 symbols from 0 to 999. Don't think of 999 as three 9s, but as a single symbol. Under this view, we can consider the comma-separated string as a string of 4 symbols in base 1,000, rather than 12 symbols in base 10:

$$123{,}374{,}743{,}125 = 123 \times 1{,}000^3 + 374 \times 1{,}000^2 + 743 \times 1{,}000^1 + 125 \times 1{,}000^0$$

This reflects our spoken language more accurately than thinking in base 10: we have names for powers of 1,000 (thousand, million, billion, trillion), but we don't have names for 10,000 or 100,000 or 10,000,000. Scientific units also follow this base 1,000 convention: kilo, mega, giga, and so on.

What's interesting about base 1,000 is the special relationship it has to base 10. Usually when we change bases, we expect the symbols to have a completely different appearance in the two bases. But when we switch between base 10 and base 1,000, the written symbols don't change at all. We've simply gone from thinking of, say, 123 as three symbols in base 10 to thinking of it as a single symbol in base 1,000. This makes it very easy and convenient to convert between the bases, as we do in our heads whenever we see or hear large numbers in everyday life.

Base 60: Sexagesimal

Let's talk about *sexagesimal*, also known as *base 60*. This system is relevant to modern computing for two reasons: first, like base 1,000, it's another example of the sort of compound notation we'll explore later; and second, it's still in heavy computational use today.

We believe that some prehistoric human groups counted in base 12. When we reach the time of the first cities (4000 BCE), the Sumerians

switched to base 60 for their scientific studies, which included astronomy and the invention of an algorithmic version of the Pythagorean theorem. This may have arisen through a fusion, collision, or compromise between people using bases 10 and 12, as 60 is readily divisible by both.

Rather than invent an alphabet of 60 distinct symbols, which would have required a large effort to learn, the Sumerians used a hybrid notation. They wrote the numbers from 0 to 59 (inclusive) in the existing base 10, but they treated these compound symbols as individual numerals in a base 60 system. For example, the symbols (using modern Arabic digits with compounds separated by colons) 11:23:13 would represent the following number:

$$11 \times 60^2 + 23 \times 60^1 + 13 \times 10^0 = 39{,}889_{10}$$

We still use a sexagesimal system today to represent time: the number above means 23 minutes and 13 seconds past 11, which is equal to 39,889 seconds into the day. Modern databases, data science systems, and date-time libraries therefore need to be carefully designed to handle conversions between sexagesimal, binary, and decimal.

Base 16: Hexadecimal

Let's talk about hex! Short for *hexadecimal* or *hex code*, *hex* is a base 16 system. Its symbols are a mix of the digits 0 through 9 and the letters a through f (for the decimal numbers 10 through 15), often prefixed by 0x to indicate they are hex.

You've probably seen hex numbers around in any computer programs in languages that allow direct access to and use of memory, including C and assembly. They also appear in higher-level languages as a way to differentiate copies of objects that otherwise have the same properties. For example, if you copy a `Cat` object (in an object-oriented language) with properties `numberOfLegs = 4` and `age = 6`, you'll get a second `Cat` object with those same properties, but the two copies are distinct because they have different names and are stored in different locations in memory. Some debugging tools will show these memory locations to allow you to see which object is which. For example, when you ask Python to print an object, you'll see a hex address, like this:

```
>> print(cat)
<__main__.Cat at 0x7f475bbf6860>
```

Human interfaces to low-level computer architecture, such as memory locations, often use hex as an alternative, more human-readable way to display what is binary information. The address in the output above is really a long string of 0s and 1s in binary, but this would be hard for a human to recognize, for example, when comparing two addresses to see if they're the same or different. Comparing hex numbers is much easier.

Hex is used for displaying binary, rather than some other system, because it has a similar relationship to binary as base 1,000 has to base 10. Because 16 is a power of 2, just as 1,000 is a power of 10, there's a one-to-one

relationship between groups of columns in binary and columns in hex. This allows for fast, easy conversion between the two systems. Consider a binary number with its digits organized into groups of four: 0010,1111,0100,1101. We can view this as

$$0010_2 \times 2^{12} + 1111_2 \times 2^8 + 0100_2 \times 2^4 + 1101_2 \times 2^0$$

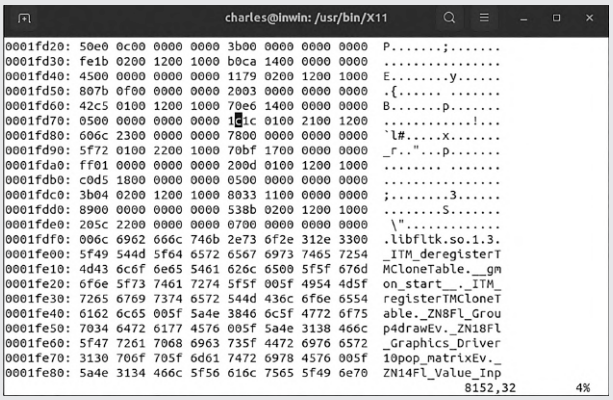
which is the same as:

$$2_{10} \times 16^3 + 15_{10} \times 16^2 + 4_{10} \times 16^1 + 13_{10} \times 16^0$$

Each of these powers of 16 has a number from 0 to 15 inclusive, so if we use the letters a_{16} to f_{16} to denote 10_{10} to 15_{10}, then we can write the number in hex as $2f4d_{16}$. Every 4 bits in the binary number (a quantity sometimes called a *nybble*) corresponds to one hex digit: the 2 in hex corresponds exactly to the first 4 bits, 0010; the f to 1111; the 4 to 0100; and the d to 1101. This four-to-one correspondence makes it easy to convert back and forth between hex and binary—much easier than, say, converting between decimal and binary.

HEX EDITORS

Hex editors (for example, in Vim, `%!xxd`, as shown in the following image) display the contents of files or memory in byte notation, sometimes together with other translations such as ASCII characters. They allow you to edit the corresponding binary data directly. This is useful for editing binary data and executable (compiled program) files on disk, or poking (overwriting programs and data) in the computer's memory, such as programs currently running. These editors have many interesting security-related applications. For example, you might use one to try to find and circumvent parts of a proprietary program that check for verified purchases, or to overwrite your number of lives in a computer game to get 255 instead of 3.



Hex is a convenient tool for humans to think about binary numbers in a computer, but it's important to recognize that hex isn't a tool the computers themselves use. We don't build physical computers using hex as a base; we build them using binary. Then we chunk the computers' binary numbers into fours and translate them into hex to make them more human-friendly. After all, 16 is just a bit more than 10, and so is the kind of number that humans can get used to thinking in, rather than binary.

Alternate notations used in some books and programming languages for hex include $2F4D_{16}$, 0x2f4d, 2F4Dh, &2F4D, and $2F4D.

Base 256: Bytes

Using the base 1,000 trick again, it's common to see hex code grouped into *pairs* of hex digits, such as 2D 4F 13 A7. Here, each pair can be viewed as a single symbol from an alphabet of 256 symbols, with each symbol representing 8 bits, known as a *byte*. Bytes were the main unit of computation in the 8-bit era. The nybble is so-called because it's half a byte. Remember that a nybble is one hex digit; a byte is a pair of hex digits.

How to Convert Between Bases

To convert from any base b representation to decimal, sum the decimal values of each of the base b columns:

$$x_n b^n + x_{n-1} b^{n-1} + \ldots + x_0 b^0$$

For example, here's how to convert a number from base 19 to decimal:

$$\begin{aligned} 6\text{H}92\text{A}8_{19} &= 6 \times 19^5 + 17 \times 19^4 + 9 \times 19^3 + 2 \times 19^2 + 10 \times 19^1 + 8 \times 19^0 \\ &= 14{,}856{,}594_{10} + 2{,}215{,}457_{10} + 61{,}731_{10} + 722_{10} + 190_{10} + 8_{10} \\ &= 17{,}134{,}702_{10} \end{aligned}$$

To convert from decimal to base b, use repeated integer division by b with remainders. Table 2-1 shows the steps of converting 186_{10} to binary.

Table 2-1: Converting 186 to Base 2

Step	Result	Remainder
186/2	93	0
93/2	46	1
46/2	23	0
23/2	11	1
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

Here, the binary form of 186_{10} is obtained by reading up the remainder column: 10111010_2.

Most programming languages provide functions that automatically carry out common conversions, with names like `bin2hex` and `hex2dec`.

Representing Data

Once you have a basic representation for whole numbers, such as any of the base systems we've discussed, you can use it as a first building block to construct representations of other things: more complicated types of numbers, text, multimedia, and any general hierarchical data structure. Here we'll see such representations, often using systems we've already defined as components of other higher-level systems. This can be as simple as using a pair of whole numbers to represent a fractional number or as complex as using billions of floating-point numbers grouped into spatiotemporal hierarchies to represent a multimedia stream of video, multilingual audio, and text subtitles, as found in your movie player.

Natural Numbers

The *natural numbers* (traditionally denoted by the set symbol $\mathbb{N}$) are the numbers 0, 1, 2, 3, 4, and so on. They're often used to represent numbers of *physical* things in the world, such as rocks or cows.

Natural numbers can be represented in many ways, including tallies and Roman numerals. In computer architecture, the most obvious way is to use one of the base-exponent systems we've discussed. Some computers have used the decimal base (see the "Decimal Computers" box), while most modern machines use binary. For example, using light bulbs that can be either on or off, we can represent the binary columns of the number 74 (one 64, one 8, one 2), as in Figure 2-8.



Figure 2-8: A representation of the number 74 in binary

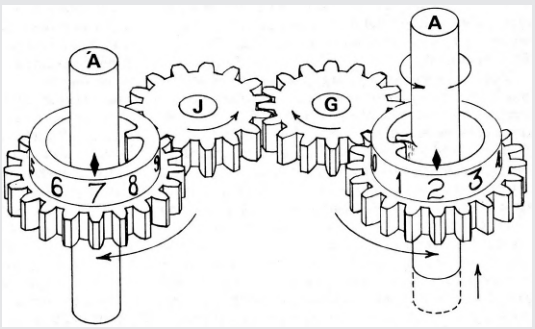
There's some subtlety to this, which will become important in more complex representations. First, you need to choose a convention for how to read the bulbs. In this case, we've chosen to put the highest power on the left, as with human-readable decimal numbers. We could equally have chosen the other way around, however, with the highest power on the right. Second, we've assumed in our example that eight bulbs are available and are being used. This means we can only represent the numbers 0 through 255. If we want to represent larger numbers, or even communicate the fact that we've run out of bulbs to represent a larger number, we'll need a new plan.

DECIMAL COMPUTERS

Decimal computers have a long history, spanning the mechanical and electronic ages. Here are some details of how they worked.

Babbage's Analytical Engine

Like Babbage's Difference Engine, his Analytical Engine uses a decimal representation, with gear wheels containing the digits 0 through 9. A gear's orientation represents a particular decimal digit, *d*, when that digit is oriented toward a fixed marker, as shown in the following figure. As in Pascal's calculator (and unlike da Vinci's), the gear never stops at continuous angles in between digits; it has only 10 discrete states.



The gear is hollow, and there's a shaft inside it. The gear and this shaft may connect via two tappets, one attached to each of their circumferences. These tappets are arranged so that if the shaft is rotated by a full circle, the tappets will connect for part of the circle, with the effect of rotating the gear by the value of its digit rather than the full circle. To read the represented number, you rotate the shaft by a full circle. For the first part of this rotation, the tappets aren't in contact and the gear doesn't move. For the second part of this rotation, the tappets come into contact and the rotating shaft makes the gear rotate along with it for *n*-tenths of a rotation, where *n* is the number represented. This rotation of the gear is what gives you access to the number. For example, if you first connect the gear to a second gear, it will have the effect of advancing that second gear's digit by *n*.

Importantly, the data is lost from the first gear when it's read, as the tappet always moves the gear into its zero position during the second part of the rotation. The act of reading the data is thus a *move* rather than a *copy*.

Many gears can be stacked vertically to represent digits of larger decimal numbers. Likewise, many of these vertical stacks are arranged horizontally in the Analytical Engine to represent many numbers together.

(continued)

Electronic Decimal Machines

Less well known in history are decimal machines of the early electronic age. The first commercial computer, UNIVAC (1951), discussed in the previous chapter, was one of them. Its main console (see the following figure) is characterized by many groups of 10 lights, used for displaying various decimals.



The IBM 650 in the following figure, dating from 1953, was notable for its use of a "bi-quinary" representation. Exactly as in the abacus, this involved a mixture of units and fives making up decimal columns.



Binary isn't the only way for digital systems, such as light bulbs, to represent natural numbers. Sometimes a one-of-*N* representation is more useful, as in Figure 2-9.



Figure 2-9: A representation of the number 5 in a one-of-N system (the leftmost bulb represents 0)

Here, we assume that *N* bulbs are available and being used, and that exactly one of them will be on at any time. This can be wasteful, because we don't use most of the possible states of the set of bulbs. But it can also be useful: for example, if we want to actually shine a light on some object in the real world, such as the fifth car in a row, we now have a single physical light bulb dedicated to that purpose. This will be very useful in computer architecture, as we very often want to switch on and off one of *N* physical circuits in a similar way. As with binary, we need to agree on a left-to-right or opposite convention, and there's no way to indicate that we've run out of bulbs if a number is too large.

Integers

The *integers* (set symbol $\mathbb{Z}$) are the numbers . . . , −3, 2, −1, 0, 1, 2, 3, and so on. They can be defined as pairing natural numbers with positive or negative signs (apart from zero, where +0 = −0). Table 2-2 shows three different options for encoding them in binary.

Table 2-2: Three Possible Binary Encodings for Integers

Integer	Signed	One's complement	Two's complement
3	011	011	011
2	010	010	010
1	001	001	001
0	000 and 100	000 and 111	000
–1	101	110	111
–2	110	101	110
–3	111	100	101
–4	n/a	n/a	100

A naive way to represent integers is to use binary codes for the natural numbers corresponding to their absolute values, together with an extra bit for their sign, as in the signed column of Table 2-2 (the leftmost bit indicates the sign). It's difficult to build machinery to correctly process these representations, however, as the sign has to be handled separately and used to select what should be done with the rest of the number. Having two different representations of the number 0 may also be a problem, requiring extra machinery to sort out.

Consider for a moment the alternative one's complement representation of the same integers given in the table. (Few people actually use this, but it will help you understand the next one.) In this representation, the codes for the positive integers are the same as for natural numbers, but the codes for negatives are obtained by inverting all of the bits for their corresponding naturals. For example, to get the code for −2, we start with the code for +2, which is 010, and invert all the bits to get 101.

Now consider the two's complement representation of integers in the table. This is formed by taking the one's complement codes and adding 1 to them for negative numbers only. For example, −2 becomes 110, which is 101 + 1. This may seem like a random thing to do, but as you'll see later, the two's complement approach turns out to be very useful. It simplifies the required arithmetic machinery, which is why today's computers typically use it.

Rationals

The *rationals* (set symbol $\mathbb{Q}$) are defined as, and may be represented by, pairs of integers a/b, with $b \neq 0$. Examples include 1/2, −3/4, 50/2, −150/2, and 0/2. Many rationals are equivalent to one another, such as 4/2 and 2/1. Detecting and simplifying equivalences requires dedicated computational work, and without this work rationals tend to expand to silly scales such as the representation 1,000,000,000/2,000,000,000 representing the number 1/2.

Representing rationals is our first example of combining multiple existing representations: we need to use a pair of integers. For example, consider Figure 2-8, which we previously interpreted as a single natural number; this figure could instead be viewed as representing the rational $4/10 = 2/5$ by assuming that the first and second groups of four bulbs represent 4 and 10.

There's some subtlety in this: we need to agree that the first four of the eight bulbs are to represent the first integer, and the second four the second integer, plus we need to agree on conventions for the integers themselves (how to convey positive versus negative values), as discussed earlier. We will end up with multiple representations for many rationals, such as 4/10 and 2/5, which may initially confuse us if we want to ask whether two rationals are equal.

Fixed Point

Fixed-point numbers, such as 4.56, 136.78, and −14.23, are numbers with a limited number of digits before and after the point. In these examples there are always two digits after the point. Formally, fixed-point numbers are a subset of the rationals, as they can always be written as an integer divided by some power of 10. They can be easily represented in computers as pairs of integers, corresponding to the two parts of the number before and after the decimal point, provided we agree on a convention for their ordering and size, as well as a convention for the integers themselves.

For example, the bulbs in Figure 2-8 could now represent the fixed-point binary number $0100.1010 = 4\frac{5}{8}$ if we agree that the point is fixed after the fourth bulb. Note that these are exactly the same bulbs that we previously used to represent the rational $4/10$ and the integer 74; to interpret data as a representation, we need to agree on which representation system is being used.

Floating Point

Floating-point numbers, such as 4.56×10^{34} and -1.23×10^{-2}, are a computational version of the Suzhou place notation system seen previously in Figure 2-7, and are composed of a fixed-point mantissa (here, 4.56) and an integer exponent (here, 34). They are easily represented in computers by pairing together an integer representation and a fixed-point representation.

To do this in practice, you need to choose specific representations for the fixed-point and integer parts, with specific bit lengths and a specific ordering for how to pack them together into a pair. It's also useful to reserve a few bit strings for special codes, such as plus and minus infinity (which can be used to code results for $1/0$ and $-1/0$) and "not a number" (*NaN*, used to code exceptions such as when trying to compute $0/0.0$). *IEEE 754* is a commonly used standard for making these choices. It includes a set of bit orderings to best make use of 8, 16, 32, 64, 128, or 256 bits as floating-point representations. For example, IEEE 754's 64-bit standard specifies that the first 53 bits should be used as the fixed-point mantissa in a signed encoding, with the first bit holding the sign; the remaining 11 bits should serve as a two's complement integer exponent. Some bit patterns are reserved for infinities and NaNs.

COMPUTABLE REALS

Beyond floating points, computer science has its own concept of *computable real numbers*, sometimes written as $\mathbb{T}$, which are different from—and better than—the *real numbers* used in mathematics, denoted with $\mathbb{R}$. Computable reals are all the numbers that can be defined by programs. In contrast, the much larger set of mathematicians' real numbers are useless as they can't be individually defined or used in computation.

Imagine a physical turtle robot controlled by a language like Scratch, moving left and right along a number line. The computable reals are all the locations on the number line that you can write a program for the turtle to stop at. Specifically, they're all the numbers whose *n*th digit can be specified by some finite-length computer program.

(continued)

For example, we can write a function `pi(n)` that takes an integer `n` as input and returns the *n*th digit of π. Likewise, we can add two computable reals, `a(n) + b(n)`, by forming a new program from the programs `a(n)` and `b(n)`. The new program will take `n` as an input and call `a()` and `b()` one or more times to generate the *n*th digit of the output.

Computable reals have many fascinating and almost paradoxical properties, which have deep implications for both computer and human arithmetic. For example, it's generally impossible (uncomputable) to know whether two computable reals are equal or different! The programs formed from performing just a few basic arithmetic operations on computable reals can quickly get quite large and unwieldy. It would be nice if we could optimize them by replacing them with shorter (or shortest) programs that give the same outputs, but it's impossible to do this. There is a "countable" number of computable reals, which is the same "size" of infinity as the integers. This is different from the mathematicians' reals, which have a larger "size" that's "uncountable."

Alan Turing defined the computable reals in his great paper "On Computable Numbers," hence the letter $\mathbb{T}$. They are his true genius contribution to computer science, rather than "inventing the computer" (the title of this paper is a clue that it's about computable numbers, rather than about computers). Turing's theory is still underappreciated. If it were more widely developed and used, we might one day get rid of the errors caused by floating-point approximations and be able to make perfectly accurate computations.

Arrays

A *one-dimensional array* is a sequence of R values:

$$\{a_r\}_{r=0:R-1}$$

A *two-dimensional array* is a collection of $R \times C$ values (standing for numbers of rows and columns), where:

$$\{a_{r,c}\}_{r=0:R-1,c=0:C-1}$$

A *D-dimensional array* is a collection of values with D indices, such as the 3D $R \times C \times D$ array with the following elements:

$$\{t_{r,c,d}\}_{r=0:R-1,c=0:C-1,d=0:D-1}$$

The values in arrays may be numbers (of any of the types of numbers we've discussed) or other types of data.

Often numerical arrays are used to represent vectors, matrices, and tensors. These are mathematical concepts that *extend* the data structure with specific, defined, mathematical operations. For example, a *vector* is a 1D array with specific rules for addition, multiplication by a scalar, and computing dot products and norms. A *matrix* is a 2D array with specific rules such as for multiplication and inversion. A *tensor* is an N-dimensional array with specific rules for covariant and contravariant coordinate transforms, in addition to multiplication and inversion. Vectors and matrices are special cases

of tensors. (Many computer scientists use the term *tensor* incorrectly to refer only to the *N*-dimensional data structure, forgetting the other mathematical requirements of true tensors.)

A basic data representation for all types of arrays is to "pack" them into a series of individual numbers in contiguous areas of computer memory. For example, Figure 2-8 might represent the 1D array of integers [1,0,2,2] if we agree on a convention that each integer is represented by two bulbs. Similarly, it might represent this 2D array of integers:

$$\begin{bmatrix} 1 & 0 \\ 2 & 2 \end{bmatrix}$$

In this case we're considering each of the 2D array's rows as a 1D array, [1,0] and [2,2]. We encode each 1D array using two bulbs per integer and store the series of encodings for the rows in order. By extension, for a general *N*-dimension array, we may do the same: split it into a series of $(N - 1)$-dimension arrays, encode each of them, and store the series of encodings in order.

Optimizing data representation and computation architectures for vectors, matrices, and tensors has become a major driver of the tech industry. GPUs were first built to perform fast 3D vector-matrix operations for real-time 3D games, and have more recently been generalized for fast tensor computations, which have found important applications in neural network acceleration. Google's tensor processing units (TPUs) are designed specifically for this task.

Text

Let's talk about text. Once you have a finite, discrete alphabet of symbols, such as the characters we use to write human-readable text, you can assign a natural number to represent each one. You can then use a bunch of natural numbers in an array to represent *strings* of text. This idea has evolved from the long-standard-but-now-outdated ASCII to modern Unicode.

A HISTORY OF TEXT

Numbers aren't very useful by themselves: we need to know *what* is being counted. Sumerian trading tokens were "typed"—three cow tokens to represent three cows. But when we moved from tokens to more abstract numerals, we lost the information about what the numbers were supposed to represent. The numbers needed to be accompanied by extra symbols describing the type, as in "3 cows." Writing thus emerged from the same trading tokens as numbers, but it forked to become pictograms and then text.

(continued)

The first writing appeared in Sumeria around 4000 BCE. It used pictures of objects (pictograms) to represent them. Pictograms appeared in many cultures, then gradually transformed into phonetic symbols. The phonetic and semantic uses may coexist for a while—as in modern Chinese—but the phonetic use usually becomes dominant. Text symbols also evolved over time to become simplified and easier to write, losing the original pictorial similarities to their objects. Where writing was carved on stone, the symbols evolved to be made from straight lines that are easier to carve. The most common symbols evolved fastest into quick-to-write shapes. They thus became the handiest to use in phonetic transcriptions, so the phonetic letters that survived the transition from pictures to sounds tended to come from the most common words.

Text isn't always written from left to right. Arabic and Hebrew go right to left, and many East Asian languages can be written from top to bottom.

Morse code was developed around the great computing year, 1836, to enable operators of the Victorian internet—the telegraph—to communicate quickly. Samuel Morse studied the frequency of letter usage in English to give the common ones the shortest representations. Morse is *almost* a binary code, as it uses sequences of two symbols to represent letters, but they're usually used together with a third symbol, empty space, to show breaks between words.

Braille was also developed around 1836 by Louis Braille. It's a true binary code, with each letter represented by binary states of a 2×3 grid. It was originally developed for secret use by soldiers but became popular for its present-day use by blind readers.

ASCII

The *American Standard Code for Information Interchange (ASCII)*, shown in Figure 2-10, represents each character as a unique 7-bit code, meaning it can represent 128 characters in total. This allows for uppercase and lowercase letters, digits, symbols, and punctuation, as well as historical *controls* such as delete, carriage return, line feed, and ring a bell.

In old email systems, ASCII control codes would sometimes be transmitted and displayed as part of the email message rather than actually being executed. The backspace control code was particularly prone to this effect, so you would get emails such as:

> The team has identified several fuckups^H^H^H^H^H^H^Hchallenges in the plan.

Today, old-timers sometimes type out similar "backspace fails" on purpose for humor.

Some operating systems use different conventions to represent the ends of lines, involving line feeds (code 10) and carriage returns (code 13) in text files, which may need to be fixed if you move text files between systems. In the days of typewriters and then teletype machines, these were two different physical controls, one to advance the paper in the machine by a row, and the other to return the print head carriage back to the left side of the paper.

ASCII code 0 is commonly used to represent the end of a string. If a string is laid out in memory, programs need a way to work their way through

it one character at a time. The convention is that when they reach a zero, they know to stop.

As a 1960s American standard, ASCII is a product of a time before globalization and the internet, and it has come to show its age. It can only represent characters from the Latin alphabet, so it can't directly represent characters needed by languages other than English. Many European languages require multiple versions of Latin characters containing accents, for example, while languages such as Chinese and Arabic use completely different alphabets.

However, in one of the most foresighted design decisions ever made in computing, coupled with coincidence, the designers of ASCII were aware of this potential future issue and planned for it. The coincidence was that the machines of the time used groups of 8 bits, while the size of the set of characters needed for English was just under 7 bits. The design decision was to thus use 8-bit representations for ASCII characters but to always make the first bit a 0. In the future, if additional characters were needed, this first bit could be used for other purposes. This has now happened, giving rise to the modern Unicode Standard.

Decimal	Hex	Description	Decimal	Hex	Char.	Decimal	Hex	Char.	Decimal	Hex	Char.
0	0	Null	32	20	[SPACE]	64	40	@	96	60	`
1	1	Start of heading	33	21	!	65	41	A	97	61	a
2	2	Start of text	34	22	"	66	42	B	98	62	b
3	3	End of text	35	23	#	67	43	C	99	63	c
4	4	End of transmission	36	24	$	68	44	D	100	64	d
5	5	Inquiry	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	38	26	&	70	46	F	102	66	f
7	7	Bell	39	27	'	71	47	G	103	67	g
8	8	Backspace	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	41	29	)	73	49	I	105	69	i
10	A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End of trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 2-10: The ASCII character representations

ASCII AS BASE 256

Suppose you've written a program in your favorite language, such as the following in BASIC:

```
10 PRINT "HELLO"
20 GOTO 10
```

Then suppose you encode the characters of this program as ASCII characters and save them in a text file. Each one of these characters is a byte. If you open your program in a hex editor rather than a normal text editor, you'll see your program represented as a list of byte codes, such as:

```
31 30 20 50 52 49 4E 54 ... 30
```

Think about the compound notation concept we've previously used in base 1,000, sexagesimal, and byte codes themselves, and apply it to this whole list of byte codes. Consider each byte code as a base 256 digit, and form the *entire program* into a single very large number, such as:

$$31_{256} \times 256^{27} + 30_{256} \times 256^{26} + 20_{256} \times 256^{25} + 50_{256} \times 256^{24} \ldots + 30_{256} \times 256^{0}$$

This calculation would give a single astronomically sized integer. This means we have a mapping between programs and integers: we can represent any program with a single integer. When you write a program, you're just choosing which integer to apply. This view can be used in computational theory, because it allows math about numbers to talk about programs.

Unicode

What we collectively know as *Unicode* actually refers to three different but related standards defined in 1991: UTF-8, UTF-16, and UTF-32. *UTF-8* extends ASCII by making use of the previously unused eighth bit. If it's a 1, then a second byte follows to enlarge the space of symbols. If the second byte starts with a 1, then a third byte follows as well. If the third byte starts with a 1, then a final, fourth byte is also used. UTF-8 thus allows for more than one million different characters. Its standard doesn't use all of them, but it includes mappings to symbols needed for all major world languages. So many character encodings are available that international communities have been able to add to the standard, including symbols for less widely spoken languages, ancient languages such as Cuneiform, fictional languages such as Klingon, other symbols such as from mathematics and music, and a large number of emoji (see Figure 2-11).

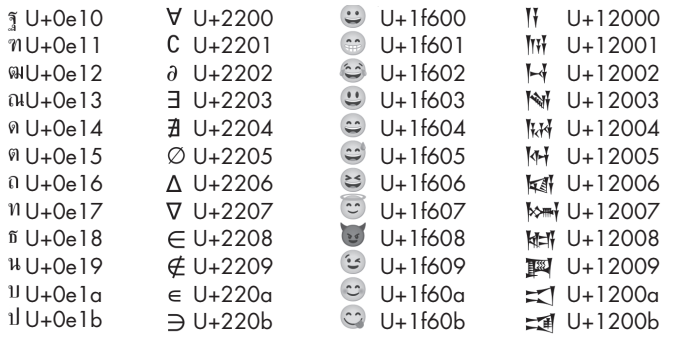


Figure 2-11: Unicode Thai, math, emoji, and Cuneiform sectors

For efficiency, the most widely used languages are given the symbols that require only 2 bytes, with the rarer ones requiring 3 bytes and the comedy ones requiring 4 bytes. There are sometimes lively debates about which of these sectors a newly proposed character set should be assigned to. The next time you send a text with just the right emojum to express your feelings, you can thank the ASCII designers for their foresight.

UTF-32 is a fixed-width encoding that uses all four available bytes in *every* character. From a storage standpoint, this is obviously inefficient, but for some applications it may speed up the process of looking up symbols. For example, if you want to read the 123rd symbol, then you can find it right away in bytes 123×4 to 123×5.

UTF-16 is like UTF-8, but at least 2 bytes are always used, even for ASCII characters. This covers a large set of symbols in common use around the world, so it can often act as if it were a fixed-width coding, to enable fast look-ups as in UTF-32. It's a compromise encoding.

Converting files between the different UTF formats is a modern version of the pain we used to have with carriage returns and line feeds in ASCII. Especially with CSV spreadsheet files, using the wrong UTF import can make good files look like garbage.

Multimedia Data Representation

Data representation gets more fun as we move to images, video, and audio to bring our computers to life. These representations are all built on the arrays of numbers we've previously constructed.

Image Data

Grayscale images can be represented by 2D arrays of numbers, with each element representing a pixel, and its value representing the shade of gray. The type of integer representation within this array affects the quality of the image: if 1-bit integers are used, then each pixel can only be black (0) or white (1), whereas if 8-bit integers are used, then 256 shades of gray are available between black (0) and white (255).

Human eyes are receptive to three main frequencies of light: red, green, and blue. This means the experience of seeing an image in color can be

reproduced by shining lights at each of these frequencies from each pixel. To represent a color image, we can therefore take three grayscale image representations; use them to represent the red, green, and blue channels of the image; and somehow store them together. Different systems may use different approaches to this storage. For example, we might store the complete red image first, then the green after it, then the blue. But some computations may run faster if we *interleave* the channels, with the red, green, and blue values for the top-left pixel stored first, one after the other, then the red, green, and blue values for the pixel next to it, and so on.

For some applications, it's useful to add a fourth channel, called *alpha*, to represent the transparency of each pixel. This representation is known as RGBA. For example, in sprite-based games this tells the graphics code how to mask each sprite, leaving the background intact behind its shape. Nonbinary alphas can also be used to blend images together by making them partially transparent to various degrees. Including an alpha channel is especially convenient because having four channels makes a power of two, which plays nicely with binary architectures. For example, it is common to use 32-bit colors with four 8-bit channels, rather than 24-bit colors with three 8-bit channels, on a 32-bit machine. Of course, this requires more storage, so pixel values might be stored as 24 bits and converted to 32 bits when loaded into memory. (Since 24-bit RGB is usually considered the maximum color depth that humans can distinguish, there's little point in going to 64-bit color, even on 64-bit machines.)

Video can be represented (most basically) as a sequence of still images packed in temporal sequence.

Audio Data

Continuous sound waves can be represented as a series of discrete samples. The samples need to be taken at a rate of double the highest frequency present in the signal. Human hearing ranges from around 20 to 20,000 Hz, so common audio sample rates are around 40,000 Hz. Each sample is a number, and as with color depth, the choice of the number of bits to devote to each sample affects the sound quality. Consumer media such as Blu-ray uses a depth of 24 bits, around the maximum distinguishable to humans, while 32 bits may be used internally and by audio producers, as it's a power of two and gives more robustness to editing manipulations.

Stereo or multichannel audio can be thought of as a collection of sound waves meant to be played together. These might be stored as one whole wave at a time in memory, or interleaved over time, with one sample from each channel stored contiguously for each sample time.

Almost all sound representations use either integer or fixed-point representations for the individual samples. A consequence of this is that there are clear minimum and maximum values that samples can take. If the signal goes out of this "headroom" range it will *clip*, losing information and sounding distorted. Musicians and voice actors often curse these data representations if they have just performed a perfect take but it got clipped and they have to do it again. A recent trend in professional audio systems is a move

to all floating-point representations, which are much more computationally intensive but free the artists from the clipping problem.

When dealing with *multi*media, such as movies that include video and audio together, the interleaving representation concept is often extended so that data from each medium for a point in time is coded together in a contiguous area of memory—for example, all the data for one video frame, plus an audio segment for the duration of that frame. The interleaving schemes are known as *containers*. *Ogg* and *MP4* are two well-known container data representations used for movies.

Compression

The simple media representations for images, video, and audio that we've just discussed are good during computation, but they aren't usually ideal in terms of storage. For efficiency, we often look for ways to *compress* the data without changing the human experience of it.

The natural world tends to contain a lot of redundancy—that is, it has regions of space and time that are composed of similar stuff. For example, in a video of a thrown red ball, if you see one red pixel belonging to the ball, then it's very likely that the pixels around it are also red, and that this pixel or nearby pixels will be red in the next frame as well. Also, the human senses have particular focuses and blind spots, for example being sensitive to the amplitude but not the phase of audio frequencies, and not hearing some frequencies in the background when others are present.

Information theory explains how to compress media data by exploiting and removing these redundancies and perceptual blind spots. This way, a smaller number of bits can be used in a more complex way to represent the same or perceptually similar media data. This is useful both to reduce physical storage needs, such as the size of a Blu-ray disc, and also to reduce network use when streaming media. However, it comes at a cost of additional computation: we usually need to convert the compressed representations back to the raw ones, which can be quite complex, depending on the compression scheme used. Most schemes rely on mathematical operations like Fourier transforms to find spatial or temporal frequencies. These can be costly for conventional CPUs to compute and have been a major driver of specialized signal processing architectures to accelerate them. Implementations of compression algorithms are known as *codecs*.

Data Structures

Any data structure, such as the structs and objects found in most programming languages, can be represented through *serialization*, whereby the data is transformed into a *series* of bits to store in memory. Serialization can be performed hierarchically: if a complex structure is composed of several smaller structures, we serialize it by first serializing each of these components, then joining their representations together in series to make the total representation. If the component structures are themselves complex, the process becomes recursive, but eventually we always reach a level of simple

elements such as numbers or text, and we've already discussed how to represent these as a series of bits (that is, serialize them).

To give an example, say we have the following data structure:

```
class Cat:
  int age
  int legs
  string name
```

This will be serialized as a bit sequence beginning with the encoding for integer `age`, followed by the encoding for integer `legs`, and then perhaps a Unicode sequence for the string `name`.

Now say a `Cat` object is included in another structure:

```
class Game:
  Cat scratch
  int lives
  int score
```

The `Game` object will be serialized with its first bits being the encoding of the `Cat` object (itself a serialization of various components), followed by the encodings of the `lives` and `score` integers. We can continue to build higher and higher levels of structure in this way, which is how real-world large-scale programs work.

Measuring Data

The basic unit of data is the *bit (b)*, which can take one of two possible states, usually written as 0 and 1. When studying data, we'll often be working with very large numbers of bits, however, so we need notations and visualizations to handle these.

SI (Système Internationale) is an international organization of scientists and engineers that sets generally accepted standards for scientific measurement units. This includes defining standard prefixes for powers of 1,000, as shown in Table 2-3.

Table 2-3: Large SI Prefixes

Name	Symbol	Value
kilo	k	10^3 =1,000
mega	M	10^6 = 1,000,000
giga	G	10^9 = 1,000,000,000
tera	T	10^{12} = 1,000,000,000,000
peta	P	10^{15} = 1,000,000,000,000,000
exa	E	10^{18} = 1,000,000,000,000,000,000
zetta	Z	10^{21} = 1,000,000,000,000,000,000,000

To visualize the large scales represented by SI prefixes, it can be useful to imagine 3D cubes, based on a cubic meter. Perhaps the reason we give special names and prefixes to powers of 1,000 is that 1,000 is three scalings of 10, which in 3D means scaling an object by 10 in all three of its dimensions.

Using SI prefixes with bits should be the preferred standard for describing quantities of data, according to SI—for example, 5 megabits means 5,000 bits. In fact, network speeds are often measured in megabits per second. However, at the architectural level we more commonly need to work with numbers that are exact powers of 2, not 10. For example, a 10-bit address space provides 2^{10} = 1,024 addresses, while a 16-bit address space provides 2^{16} = 65,536 addresses. Before architects adopted the SI standards—during the 8-bit era, for example—it was common for architects to abuse the prefix "kilo" to refer to 1,024 instead of 1,000.

This naturally led to much confusion. Data sizes have gotten larger, and most computer people operate at a higher level, where working in proper SI units makes more sense. As a compromise, in 1998 the International Electrotechnical Commission defined an alternate set of prefixes to distinguish the powers of two from the SI prefixes. These have the morpheme *bi* in them, from the word *binary*. For example, 2^{10} has become kibi, 2^{20} has become mebi, and so on, as in Table 2-4.

Table 2-4: Large Binary Prefixes

Name	Symbols	Value
kibi	k_2, ki	2^{10} = 1,024
mebi	M_2, Mi	2^{20} = 1,048,576
gibi	G_2, Gi	2^{30} = 1,073,741,824
tebi	T_2, Ti	2^{40} = 1,099,511,627,776
pebi	P_2, Pi	2^{50} = 1,125,899,906,842,624
exbi	E_2, Ei	2^{60} = 1,152,921,504,606,846,976
zebi	Z_2, Zi	2^{70} = 1,180,591,620,717,411,303,424

Binary prefixes are slightly larger than their SI counterparts. Not everyone is using them yet, and many older people and machines still use SI names to refer to binary units. Unscrupulous hardware manufacturers often exploit this ambiguity by picking whichever interpretation of the SI names will give them the best-looking numbers on their products.

Summary

Computers usually need to represent various types of numbers, text, and media data. It's convenient for modern machines to do this using binary. Hex representations chunk binary together to appear more readable to humans. Different representations make different computations easier to perform.

Once we have methods for representing data, we can begin to build up methods for computing with the data. In the next chapter, we'll preview a simple but complete computer that does this. We'll then build up a more detailed modern electronic computer to do similar.

Exercises

Base System Conversions

1. Convert your phone number to binary.
2. Convert your phone number to hex. Making use of the binary from before might be helpful. Convert it again to byte codes, and convert the bytes to ASCII characters. What do they spell out?
3. Negate your phone number and convert this negative number to its two's complement.
4. Place a decimal point halfway through your phone number to make a floating-point number. Write it in IEEE 754 standard binary.

Text and Media

Find out how to type in Unicode on your computer. On many Linuxes, for example, you can press and release SHIFT-CTRL-U, then type a series of hex numbers such as 131bc to enter an ancient Egyptian digit at your command line or in your editor.

Measuring Data

Obtain street, aerial, and satellite photos of an area you know, and draw a kilocube, megacube, gigacube, teracube, petacube, exacube, and zettacube on them, where, for example, each side of a kilocube is 10 m long.

More Challenging

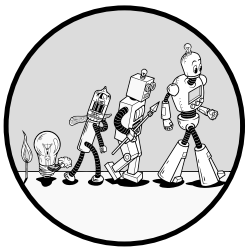
Use a hex editor and the internet to reverse engineer and modify some of your favorite media files.

Further Reading

- For discussions and psychological models of numerosity, see Stanislas Dehaene, *The Number Sense* (Oxford: Oxford University Press, 2011), and the "Numbo" chapter in Douglas R. Hofstadter, *Fluid Concepts and Creative Analogies* (New York: Basic Books, 1995).
- For an advanced but classic paper full of details on floating points, see D. Goldberg, "What Every Programmer Should Know About Floating Point Arithmetic," *ACM Computing Surveys (CSUR)* 23, no. 1 (1991): 5–48.
- For an extremely advanced but blindingly beautiful book on Turing reals, see Oliver Aberth, *Computable Calculus* (San Diego: Academic Press, 2001).

3

BASIC CPU-BASED ARCHITECTURE



Modern CPUs are some of the most complex structures known to humanity, but the basic concepts underlying them, such as executing instructions sequentially or jumping forward or backward to different instructions, are actually quite simple and haven't changed for over 150 years. To ease our way into the study of CPU architecture, this chapter introduces these fundamental concepts by looking at a related but simpler system: a mechanical music player. You'll then see how the same concepts, together with RAM, form the basis of Charles Babbage's Analytical Engine. Studying—and programming—this mechanical system will make it easier to understand what's going on when we turn our attention to electronic systems in Chapter 4.

A Musical Processing Unit

For a machine to be a computer, it needs to be *general purpose*, meaning it must be able to perform different tasks according to a user specification. One way to arrange for this is to have the user write a sequence of instructions—a program—and have the machine carry them out. A musical score can be viewed as a program, and so we can think of a machine that reads and performs musical scores as a kind of musical computer. We'll call such a device a *musical processing unit*.

In Chapter 1 we looked briefly at musical processing units such as barrel organs and music boxes. After Babbage, musical automata and their programs continued to evolve. Around 1890, "book organs" replaced barrels with continuous, joined decks of punch cards ("book music"), which could accommodate arbitrarily longer compositions without the size limit imposed by a barrel. By 1900 these had evolved to pianolas, or player pianos (Figure 3-1), which used punched paper *piano rolls* instead of cards to drive domestic pianos, rather than church organs. Player pianos are still found today; you might hear one providing background jazz in a mid-range hotel that can afford a piano but not a pianist.



Figure 3-1: A player piano (1900)

Let's think about some of the types of instructions found in musical scores that might be playable on these machines. These will be similar to but perhaps more familiar than concepts that we'll need later to make computers. We'll consider only a monophonic instrument here, meaning it can only play one note at a time.

The set of possible instructions that we can give to an automated musical instrument usually contains one instruction per available note. This might be an instruction to "play middle C" or "play the G above middle C," for example. Each row of a player piano's paper roll represents a time and contains one column per musical pitch, which is specified to be either on (punched) or off (not punched) at that time. Modern computer music software such as Ardour 5, released in 2018, continues to use this type of

piano roll notation (turned on its side for human viewers, so time scrolls more intuitively from left to right) to generate electronic music (Figure 3-2).

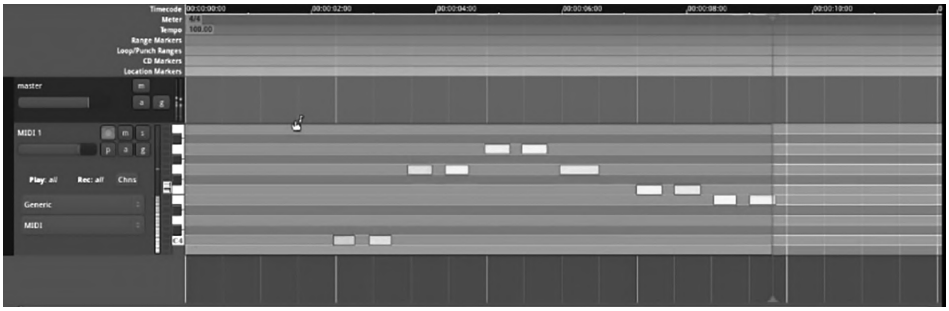


Figure 3-2: An Ardour 5 piano roll interface (2018)

When a player piano reads a piano roll, one row at a time is placed into a reader device. Let's call this *fetching* the instruction. The instruction is then *decoded* by some machinery that looks at the punch-hole coding and turns it into a physical activation of some machinery that is going to play the note, such as by opening a tube for air to flow into an organ pipe. Then this machinery actually *executes* the performance of the note.

Usually when a human or mechanical music player is following a music program (score), they will execute (play) each instruction (note) and then move on to the next one, advancing their position in the program by one instruction. But sometimes there will also be special additional instructions that tell them to *jump* to another place in the program rather than advancing to the next instruction. For example, *repeats* and *dal segno (D.S.)* are used to jump back to an earlier instruction and continue execution from there, while *codas* are instructions to jump forward to a special ending section. Figure 3-3 shows a musical program.
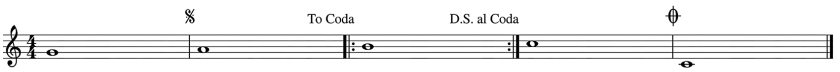


Figure 3-3: A musical program with notes G, A, B, high C, and low C, as well as jumps shown by repeats, dal segno, and coda

You can build a barrel organ or player piano that encodes these jump instructions using extra, non-note columns in their punch cards. When one of these is punched, it might be interpreted as an instruction to fast-forward or rewind the barrel or punch cards to a previous or later line. Figure 3-3 could then be coded with punches representing something like:

```
1.  play note: G
2.  play note: A
3.  check if you have been here before
4.  if so, jump to instruction 10
5.  play note: B
6.  check if you haven't been here before
```

```
7.  if so, jump to instruction 5
8.  play note: high C
9.  jump to instruction: 2
10. play note: low C
11. halt
```

If you don't read music, this program explains exactly what the musical score does!

From Music to Calculation

It's a small conceptual step from this musical processing unit to building a machine that performs arithmetical, rather than musical, operations.

Suppose you've already built several small mechanical devices that each perform some arithmetic operation. For example, Pascal's calculator is a machine that performs integer addition. With some thought, we could similarly construct machines like Pascal's calculator to perform integer multiplication, subtraction, division, and column shifting. We could then write a program, much like a musical score, that would specify the sequence in which we'd like each of these simple machines to be activated.

Assuming that your arithmetic machines all share a single accumulator where the result of each operation is stored, you could describe calculations similarly to sequences of instructions for pressing buttons on a calculator, such as:

```
1. enter 24 into the accumulator
2. add 8
3. multiply by 3
4. subtract 2
5. halt
```

This program would halt with the result 94 in the accumulator. The program could be executed by a human, activating the simple machines in sequence, or we could use a player piano–style roll of punch cards to specify the sequence of instructions, and a Jacquard loom–style mechanical reader to read them and automatically activate the corresponding simple machines in turn.

From Calculation to Computation

To make a Church computer, it's not enough to run programs of fixed sequences of arithmetic instructions. Computation theory tells us that some functions can only be computed using decisions and jumps, so we need to add similar instructions to those of our musical processing unit, facilitating repeats, codas, and the like. This would enable programs such as:

```
1. enter 24 into the accumulator
2. add 8
3. multiply by 3
```

```
4. subtract 2
5. check if the result is less than 100
6. if so, jump to instruction 2
7. halt
```

Computation theory also tells us that some computations require memory to store intermediate results. To distinguish between these results, we'll give each value an *address*, which for now is just an integer identifier. Memory that is addressable in this way is widely called *random-access memory (RAM)*. (This is not quite the correct definition of RAM, but you'll get to that in Chapter 10.)

Having RAM available means that we can add instructions to *load* (read) and *store* (write) to addresses, as in this program:

```
1. store the number 24 into address 1
2. store the number 3 into address 2
3. load the number from address 1 into the accumulator
4. add 8
3. multiply by the number in address 2
4. subtract 2
5. check if the result is less than 100
6. if so, jump to instruction 4
7. halt
```

Computation theory tells us that we can simulate any machine if we have the three kinds of instructions I just demonstrated: those that do the actual work of the arithmetic operations; those that make decisions and jumps; and those that store and load from RAM. This is exactly how Babbage's Analytical Engine was designed.

Babbage's Central Processing Unit

Despite its age, Babbage's Analytical Engine is a striking modern design: its basic architecture is still used in all modern CPUs. At the same time, it has only the most essential CPU features, so studying it provides a simplified introduction to the basic concepts underlying more modern CPUs. The motion of the Analytical Engine's mechanical parts also makes it easier to visualize how it works compared to today's electronic computers.

In this section I use modern terminology to describe the Analytical Engine's parts and functions. These aren't the terms Babbage used, but they'll help later when I transfer the concepts to modern machines. (Some of Babbage's original terms are included in parentheses in case they're of interest.) Babbage and Lovelace never left documentation for their instruction set, but it's been largely inferred or fantasized from other documents. I assume the instruction set and assembly language notation used by the Fourmilab emulator, an online re-creation of the Analytical Engine (*https://www.fourmilab.ch/babbage/*).

Both my presentation and the Fourmilab emulator take some liberties with the historical truth. This is easy to do because the original source documents are messy and often contradictory. There was never a single definitive design, so we can pick the versions that best suit our story. Our purpose here is really to understand *modern* CPU concepts, so I sometimes simplify, modernize, or outright lie about some of the engine's details to make this study easier.

High-Level Architecture

The Analytical Engine consists of three things: a CPU, which executes programs; RAM, which stores data and allows the CPU to read and write it; and a bus that connects them. If that sounds similar to the overall architecture of a modern, single-core computer, that's because it is! This isn't a coincidence: the Analytical Engine's architecture was explicitly used in ENIAC (after translating its mechanics into electronics), and ENIAC then became the template for our modern electronic machines.

Physically, the Analytical Engine is made of 50 copies of the slice (what Babbage called a "cage") shown in Figure 3-4, stacked vertically, one on top of the other, as in Figure 1-14.

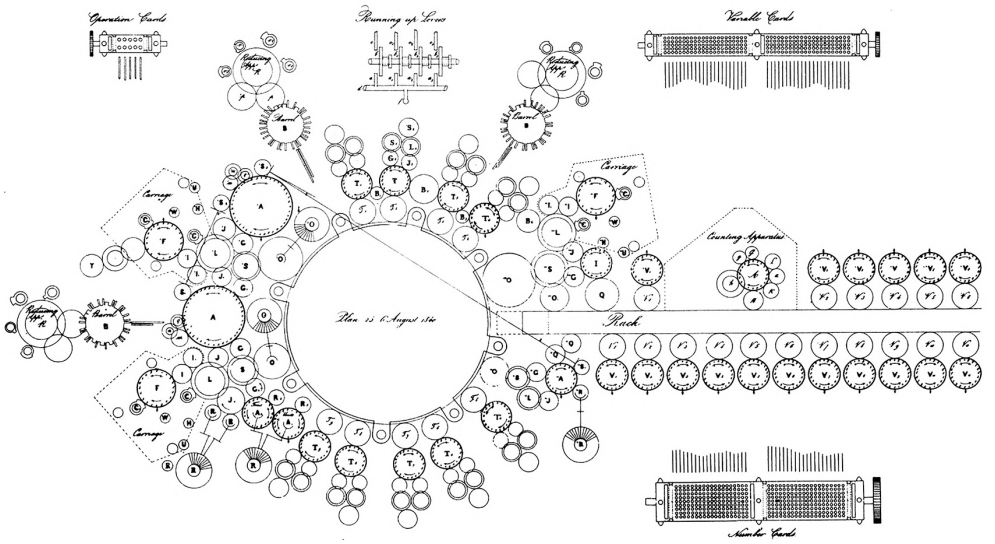


Figure 3-4: Babbage's Analytical Engine architecture (1836)

The circles are mechanical gears. The CPU, RAM, and bus each extend through all slices, and we can see each of them in Figure 3-4. For each number represented in each structure of the machine, the slice shows and handles one of its many digits. The stack of all the slices together handles all digits.

The RAM ("store axes") consists of 100 stacks of gears, with each stack representing one 50-digit decimal integer number. It appears on the slice as

the large homogeneous area on the right side of Figure 3-4. Each of these locations in the RAM has an address, numbered from 0 to 99 inclusive; this address distinguishes the location from the other locations and is used to identify it.

The RAM locations are all physically close to, but not usually touching, a mechanical bus ("rack"). The bus is a rack gear—exactly like the one found in modern car steering racks and LEGO Technic sets (Figure 3-5).

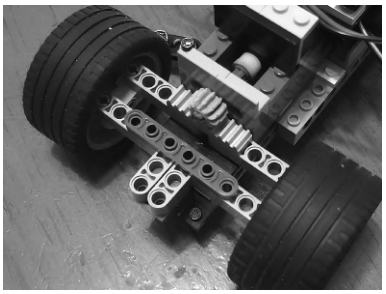


Figure 3-5: A rack (linear gear) and pinion (rotating gear)

The rack gear can *physically* shift left and right. Each of the RAM locations can be brought into contact with the rack by levers. The gears in that RAM location then act as pinions so that giving off the number from the location makes the bus physically shift to the left by that amount. Or, acting in the opposite direction, shifting the bus to the right from elsewhere adds numbers into the memory location.

The CPU ("mill") is the active part of the machine. It requests data from and sends data to the RAM on the bus, and then processes it in various ways.

Programmer Interface

Unlike the Difference Engine, the Analytical Engine was designed as a general-purpose computer. This means we can ask it to perform different operations in different orders. To do this, we need a way to specify what these operations and orders are.

Let's clarify some terms I've been using loosely. An ordered list of *instructions* to perform operations is called a *program*. The act of carrying out a program is called *execution* or a *run*. The set of all available instructions is the *instruction set*.

Programs are stored as codes on punched cards, like those of the Jacquard loom seen previously in Figure 1-11. Each card contains one row of holes and non-holes, which together code for one instruction. Usually the instructions are executed in order, with the cards advancing in sequence, but some instructions make the cards rewind or fast-forward to jump around in the program. Let's look at what particular instructions are available.

Constants

One basic instruction is to set one of the RAM addresses to a given integer. For example, "Put the integer 534 into RAM address 27." This will move the gears in the 27th RAM location's column to the (decimal) digits 534, with zeros on the gears for the thousands place and higher. Let's first denote this using a human-readable notation:

```
N27 534
```

Here, `N` (for *number*) tells us that this is a RAM integer-setting instruction. The following number (`27`) tells us which RAM location is to be set, and the final number (`534`) is the value we're setting it to. A typical program begins by setting many RAM addresses to specific values in this manner. For example:

```
N27 534
N15 123
N99 58993254235
N0  10
N2  5387
```

Once we have some starting values, we can then use further instructions to compute with them, as in the next sections.

Load and Store

To process values from RAM, they must be moved into the CPU. To load a value from RAM into the CPU, we write `L` for *load*, followed by the RAM address where the value is stored. For example, this program sets the 27th RAM location to the value 534, then loads the value from this location into the CPU:

```
N27 534
L27
```

To store the CPU's latest result to RAM address 35, we write `S` for *store* followed by the desired address:

```
S35
```

Storing (`S`) is different from setting RAM to a constant (`N`) because it involves the CPU's accumulator. It transfers whatever value is in the accumulator to the RAM, rather than putting a fixed constant into RAM.

Now that we can move data around, we would like to perform calculations in the form of arithmetic on it.

Arithmetic

The Analytical Engine is able to perform elementary arithmetical operations: addition, subtraction, multiplication, and division, all on integers. These are denoted by the instructions `+`, `-`, `*`, and `/`.

To do arithmetic, you first have to set the *mode*, which tells the engine which of these operations you want to do. For example, to add two numbers, you put it into adding mode and then load the two arguments in order into the CPU. Consider the following program:

```
N0 7
N1 3
+
L0
L1
S2
```

This program first puts the integers `7` and `3` into addresses 0 and 1, respectively. It then puts the CPU into adding mode with the `+` instruction and loads the number from these addresses. It finally stores the result of the addition into address 2.

Now that we have arithmetic, we finally need to move from calculation to computation by adding jumps and branches.

Jumps

If you want part of a program to repeat forever, a simple method is to glue the end of the last punch card to the top of the first one to create a physical loop, as in Figure 1-15. However, this doesn't generalize well, so it's useful instead to have an instruction to rewind or fast-forward the cards to jump to any other line of the program when needed. Call this `C` for *control*. We'll then say whether we want to go backward (`B`) or forward (`F`) in the cards, and by how many. We'll also include the symbol `+` before the number (for reasons you'll see in the next section). Putting it all together, `CB+4`, for example, is a control instruction to go backward by four cards.

The following program uses `CB+4` to loop forever:

```
N46 0
N37 1
+
L46
L37
S46
CB+4
```

Here we use address 46 as a counter, adding 1 to its value every time we go around the loop.

Branches

Looping forever often isn't very useful; we usually want to loop *until* something has happened, then stop looping and move on to the next part of the program. This is done with a conditional *branch*, which asks whether a condition holds and jumps only if it does.

We'll use the same `CF` and `CB` notation we used for jumps, but with the symbol `?` replacing the `+` to denote that the jump is conditional. For example, `CB?4` is the control instruction to go backward by four cards only if some condition is true.

The following program uses a conditional branch and an unconditional jump together to compute the absolute value (always positive) of the sum of two numbers.

```
N1 -2
N2 -3
N99 0
+
L1
L2
S3
+
L99
L3
CF?1
CF+4
-
L99
L3
S3
```

This program uses the `+` instruction to add the two numbers in RAM locations 1 and 2, storing the result at location 3. It then adds zero (loaded from address 99) to that result, loaded back from location 3. Behind the scenes this addition operation also sets a special *status flag* to a 1 if the sign of the result differs from the sign of the first input (zero is considered positive). The conditional instruction (`CF?1`) then uses this status flag to decide what to do. If the flag is a 1, we skip over the next instruction, and so we arrive at the `-` instruction and perform a subtraction of the result from 0 to swap its sign. If the status flag is a 0, the conditional jump doesn't occur, so we simply move on to the next instruction (`CF+4`). This is an unconditional jump that skips over the four lines of subtraction code so as not to swap the sign. The final result is stored in address 3.

Branching completes the instruction set of the Analytical Engine and (assuming enough memory is always available) makes it into a Church computer. You can try tackling the end-of-chapter exercises and programming the Analytical Engine now—or, if you're interested to see how the machine works on the inside, read on.

Internal Subcomponents

Let's look at the subcomponents *within* the CPU that are needed to execute these programs. This section describes their static structure; we'll bring the

subcomponents to life in the next section when we cover how they move and interact with one another.

A CPU is formed from many independent simple machines, each made from several number representations and the machinery that acts upon them. The simple machines are grouped into three types: registers, an arithmetic logic unit, and a control unit.

As shown in Figure 3-4, all of these simple machines are arranged in a circle around a single large gear called the central wheel. Like the bus, the central wheel makes and breaks arbitrary data connections between components, in this case between the simple machines inside the CPU. These connections are made and removed by levers that put small additional gears into contact between the central wheel and the various machines.

Registers

Registers (what Babbage called "axes") are small units of memory location inside the CPU itself, rather than in the main RAM. There are only a few registers in the CPU, while there are many RAM addresses.

Recall from Chapter 2 that integers are represented in the Analytical Engine by digital, decimal gears. A digit d is read off a gear by rotating a shaft by a full circle, which results in the gear rotating by d tenths of a circle. To represent an N-digit integer, we simply stack N of these gears vertically, spanning the N cages of the machine. A register is one of these stacks.

The input register ("ingress axle") receives incoming data from RAM. The output register ("egress axle") temporarily stores (or *buffers*) results from the CPU's work, which are then transferred out to RAM. Other registers are used during computations for other purposes.

Arithmetic Logic Unit

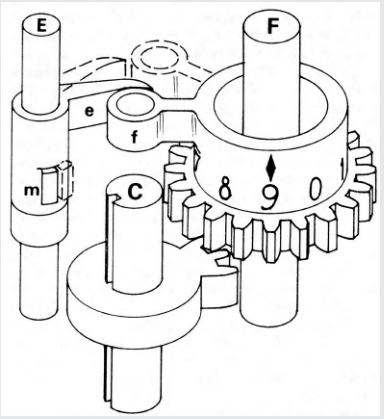
The *arithmetic logic unit (ALU)* is a collection of independent simple machines that each perform a single arithmetic operation. For example, a simple machine similar to Pascal's calculator is used to do addition. Multiplying by m can be done by a machine that triggers m repetitions of this adder. Multiplying or dividing by the nth power of 10 can be done by an especially simple machine that shifts all of its digits by n columns, the mechanical equivalent of "putting a zero on the end."

In addition to sending the result to an output register, some ALU operations can also set a single *status flag* as an extra, side-effect output. The status flag in the Analytical Engine is a single mechanical lever that is in either the up (1) or down (0) position. It might have had an actual red fabric flag on it to visually alert human as well as mechanical observers that "something interesting just happened" in the ALU.

ALU MECHANISMS

A digit *d* is given off from a gear *D* when it's read by physically rotating the gear by *d* tenths of a full circle. This digit can be added to another digit *a* stored on gear *A* by placing the gears next to one another so that their teeth mesh together, then giving off from *D*. As gear *D* rotates *d* tenths of a circle, gear *A* will be caused to rotate by the same amount, so gear *A* will end up storing the digit *a* + *d*. We say that *A* acts as an *accumulator* because we can go on adding many digits into it, and it accumulates their sum—that is, until the total goes above 9.

Integers larger than 9 are represented on stacks of gears, such as in registers. Adding them together is done similarly to adding in columns with pen and paper: the two digits in each column need to be added, but we also need to keep track of carrying when a digit goes above 9 by passing a 1 to the next column. Pascal had already developed a basic mechanical ripple carry system in his calculator, which allowed numbers to be added into an accumulator, and Babbage's carries are based on this. The following figure shows part of Babbage's design.



When a gear reaches the number 9 and is rotated by one more position in an addition, such as by an incoming carry (c), a tappet (f) connects to another tappet (e). The latter connects to a rod (m) that transfers the carry "upstairs" to the next cage, where it appears as (c) for the next column. Getting the timing right for long ripples of carries is very difficult, and this is where Babbage spent most of his design time.

Control Unit

The *control unit (CU)* reads instructions from the program in memory, decodes them, and passes control to the ALU or elsewhere to carry the instructions out. Then it updates the position in the program according to either normal sequential execution or a jump. The CU is like the conductor of an orchestra, coordinating the actions of all the other components at the right times. Babbage's CU is shown in Figure 3-6.

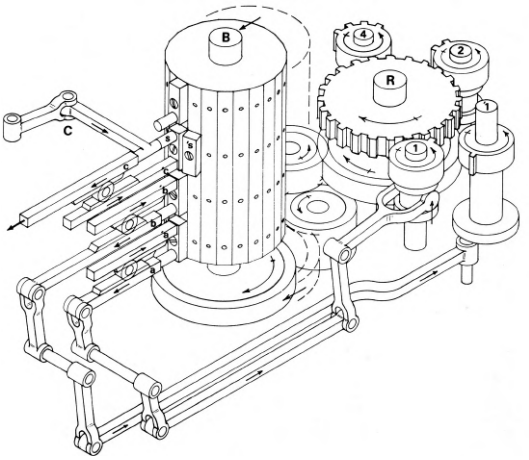


Figure 3-6: The Analytical Engine control unit

A mechanical barrel, just like that of a barrel organ, rotates over time, and each column of the barrel has several sockets for pins that may or may not be present. The pins trigger tappets that activate the other simple machines in the CPU through a complex system of mechanical levers. This enables each phase of the control unit's work to be triggered in sequence, much like a barrel organ playing a sequence of notes. The speed of rotation of the barrel can be controlled by feedback mechanisms, so the next step doesn't commence until the current step has been completed.

The configuration of the barrel's pins is *not* the user's program, but rather a lower-level *microprogram* that defines the sequencing of the CPU itself: the fetch-decode-execute cycle that we'll discuss next. As the microprogram runs, it causes individual commands from the user's higher-level program to be read into registers from punched cards, then causes those commands to be executed via the simple machines in the rest of the CPU.

Internal Operation

The CU—in Babbage's case the rotating barrel—triggers a regular cycle of activities. These are usually grouped into three main stages: fetch, decode, and execute. All of the CU's operations must be carefully timed to occur in the right order. Let's look at these three stages in turn.

Fetch

Fetching means reading the machine code for the next instruction into the CPU. Recall that the human-readable assembly language instructions such as `N37 1` and `CB+4` are actually represented as binary machine code on the punched cards. For the Analytical Engine, fetching could be done exactly as on the Jacquard loom, by attempting to insert a set of physical pins into the locations on the card. Where there's a punched hole, the pin can pass through, but where there isn't a hole the pin gets stuck on the card and

doesn't move as far. The physical positions of these pins can then be amplified and transmitted into the CPU by metal levers.

The card reader is a physical device, rather like a typewriter, in which there's a current line accessible to the pins. To read from any other line, it's necessary to pull the string of punch cards through this reader until the desired line is positioned in it. The current physical state of the punch cards—which one is currently in the reader—thus acts as a form of memory. We'll call this physical state the *program counter*.

The physical positions of the metal levers can also be considered as a form of memory that contains a copy of the current instruction inside the CPU. We'll call this the *instruction memory*.

Decode

It isn't immediately obvious what the binary encodings on the punch cards mean, either to a human or a machine: at this stage, they're just patterns of 0s and 1s. *Decoding* means figuring out what this code means. The card-reading levers traveling into the CPU can activate different pieces of machinery there, depending on what *combinations* of levers are up or down. For example, if the load instruction (`L`) is represented as binary 010, a machine could be set to respond only if three fetch levers are down, up, and down, respectively. Similarly, numerical addresses included in instructions need to be decoded, from decimal codes to mechanical activations of the addresses they represent. The decoder is a bank of machines that each look for a specific pattern in the fetched signal and activate something when they see it.

Execute: Load and Store

Execution means carrying out the decoded instruction. How this is done will depend on what type of instruction it is. Each form of execution is implemented by a different simple machine, and the decoder will select and activate the appropriate one.

Values can be *loaded* into the CPU registers from RAM when the CPU needs to use them—for example, as part of a calculation. The results of the CPU's work are also placed in registers, whose values can then be *stored* by copying them out to RAM addresses.

To load a value, the CU makes mechanical connections between the gears at the RAM address and the bus, and between the bus and input register at the CPU end. It then triggers a giving off at the RAM address, spinning the gears by a full circle so that they make the bus physically shift toward the CPU by n steps, where n is the digit represented. This occurs in parallel, with each column of the number having its own RAM gear, bus, and input register gear.

When a value is to be stored, the CU triggers the opposite set of steps. Storing assumes that the value to be stored is already in the output register. First, it clears the RAM at the target address by rotating all the digits to zero. Then it makes mechanical connections from the output register to the bus, and from the bus to the required address in RAM. Then it spins the output register by a full circle, which physically shifts the bus by n steps toward the

RAM, which in turn rotates the RAM gear by *n* steps so that the number is stored there.

Execute: Arithmetic Instructions

When an arithmetic instruction, such as an addition, is required, the appropriate simple machine, such as an adder, is brought into mechanical contact with the input and output registers and activated. In the Analytical Engine this is done mechanically by inserting gears (cogs) that physically link the registers to the simple machine, then transmitting power to the simple machine to make it run. Babbage's adder was similar to a Pascal calculator, loading in the first argument, adding the second argument to, and then transferring the result to the output register. When the calculation is done, these gears are pulled away to disable the simple machine.

In addition to affecting the output register, the ALU's simple machines may also raise or lower the status flag if something interesting happens during the arithmetic. The different simple machines in the ALU each have their own definition of "interesting" and can each set the flag according to these interests: `+` and `-` set the status flag to true if and only if the sign of their result differs from the sign of their first input, while `/` sets the status flag to true if a division by zero was attempted.

Execute: Program Flow

At the end of each instruction, the CU must complete the fetch-decode-execute cycle and prepare for the start of the next one. How this is done differs depending on whether we have a normal instruction (such as load and store or ALU instructions) or one whose purpose is to alter the program flow—that is, jumps and branches.

In *normal execution*, when an instruction completes, we want to advance to the next instruction in the program, which for Babbage is the one on the punch card whose top is attached by string to the bottom of the current instruction's punch card. This will prepare the system for the next fetch, which will be on the new instruction. To do this, the CU needs to trigger and increment the program counter. For the Analytical Engine, this is done by making mechanical connections that supply power to the punch card reader to perform a line feed, pulling the card deck through the reader by one card.

Jump instructions mean fast-forwarding or rewinding the program as requested. Consider the instruction `CF+4`, which means forward by four lines. When the CU sees this instruction, it will again modify the program counter, but rather than simply incrementing it, it will advance or rewind it by the number of lines requested. In the Analytical Engine, this is done by sending power to the line feeder for a longer time than a single line advancement, and also by mechanically switching the direction of line feed between forward and backward.

Branch instructions such as `CB?4` are executed differently, depending on the state of the status flag. This instruction, for example, tells the CU to jump, decreasing the program counter by four, if and only if the status flag is up. Otherwise, the instruction has no effect, and normal execution is used

to increment the program counter and move to the next instruction. This branching is the important difference that separates the Analytical Engine from previous barrel and punch card program machines such as music players and the Jacquard loom. Unless historians discover any previous machines that could do it, this engine marked the first time that a machine was designed to modify the execution of its own program rather than always follow it in the same sequence. This ability to look at the state of things and make decisions based on it is a key requirement of a Church computer.

Summary

We've studied Babbage's Analytical Engine in this chapter because it was and still is the blueprint for all computers that came after it, including modern PCs. Its high-level architecture includes a CPU, RAM, and a bus connecting them. Inside the CPU is an ALU, registers, and a CU that conducts a fetch-decode-execute cycle. The instruction set includes load and store, arithmetic, and jump and branch instructions. There's a program counter storing the current program line number, and a status flag that gets set if something interesting happened in the latest arithmetic operation. All of these features are found essentially unchanged in a modern PC.

As a mechanical system, the Analytical Engine can be much more concrete to visualize and understand than electronics. But electronic computers are based on simply translating each of Babbage's components into a faster and smaller implementation based on electronic switches grouped into logic gates. In the second part of this book, you'll see how this is done by building up the modern electronic hierarchy from switches to CPUs. Now that you've seen what a CPU needs to do, you should have a clearer picture of where this electronic hierarchy is heading.

Exercises

Programming the Analytical Engine

1. Install the Fourmilab Analytical Engine emulator from *https://www.fourmilab.ch/babbage*, or use its web interface.
2. Enter and run the Analytical Engine programs discussed in this chapter. If you run the programs using the `java aes -t test.card` command, then the `-t` option will print out a trace of changes to the machine state at each step.

Lovelace's Factorial Function

Write a factorial function for the Analytical Engine. Ada Lovelace wrote one of these, and it has since become the standard "Hello, world!" exercise to try whenever you meet a new architecture. (Actually printing "Hello, world!" tends to be more complicated, as it requires ASCII and screen output—you'll see how to do this in Chapter 11.)

Further Reading

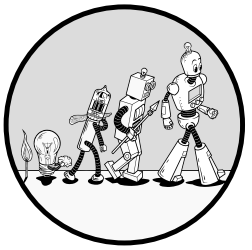
- For a more historically accurate description of the Analytical Engine, see A. Bromley, "Charles Babbage's Analytical Engine, 1838," *Annals of the History of Computing* 4, no. 3 (1982): 196–217.
- For a more fictional version, see William Gibson and Bruce Sterling, *The Difference Engine* (London: Victor Gollancz, 1990). This is the original steampunk novel, featuring Babbage, Lovelace, and a working Analytical Engine.

PART II

THE ELECTRONIC HIERARCHY

4

SWITCHES



Switching one signal on and off in response to the state of another signal is a fundamental ingredient of computation. It's what separates a limited machine like the Jacquard loom from a general-purpose machine such as Babbage's Analytical Engine. Whereas the Jacquard loom can only ever perform a predetermined sequence of operations and can't change that sequence in response to what's happening, the Analytical Engine has instructions that can evaluate the state of a register and jump around the program in response to that evaluation. Switching makes this possible.

From basic switches, we can build more sophisticated devices like logic gates, simple machines, and CPUs. As we saw in Chapter 1, the main type of switch used in today's computers is the transistor. Transistors work through a mixture of fundamental physics ideas around directionality and specific implementation of those ideas that hinge on the properties of substances

like silicon. If we jump straight to discussing transistors, however, these two threads can be hard to separate.

Accordingly, in this chapter we'll first consider the fundamental physics ideas in a simpler directional system: water flowing through a pipe. We'll see how a valve can start and stop the flow of water, then transfer what we've learned to electrical diodes made with vacuum tubes and semiconductors. We'll then build more complex switches, starting again with a water analogy and applying it to the transistors found on modern silicon. Finally, we'll explore how modern silicon chips are fabricated, so that you can understand computers right down to the grains of sand they're made from.

Directional Systems

A switch is a directional system: it takes an input and does something, causing an output. This feels fairly intuitive, but in physics you can take any equation describing a physical system, invert the direction of time, and it will still work. So why is it that when we drop a glass, its atoms don't usually jump back up into place to form a new glass? The answer is *entropy*, or the organization of energy; the chemical and potential energy in the glass dissipates into the atmosphere as a tiny amount of heat. There actually *is* a small chance the glass would reconstitute itself, but energy is much more likely to spread out as heat than to concentrate into organized structures. At the start of the Big Bang, all the energy in the universe was in one very organized place, and ever since then it's been spreading out more and becoming less organized.

It's entropy that enables us to experience the direction of time and the feeling of causation. The past was more organized than the future, so it's easier for our brains to store information about the past than about the future. Because we have memory of the past, we can link it to what we see in the present and describe past events as causing present ones.

Entropy also enables us to build machines, including computers, in which we experience being able to cause sequences of events to occur in a given direction. To give the machine a very high probability of running in a given direction, rather than randomly switching between running backward and forward, we create it in a highly organized energy state, and set up the desired sequence of states so that each step uses up some of the organized energy and gives it off as heat. This is why computers have to use up energy organization and emit heat—to make their programs run in the right direction over time.

Water Valve

A water valve found in everyday plumbing is a simple example of a directional system. For instance, there's typically a valve positioned in the pipe that feeds water from the local supply into your home. It ensures that the water can travel only *into* your home through the pipe, and not *out* of your home. This prevents you from, for example, poisoning the rest of your street

by pouring chemicals down your drain. The way this water valve works is shown in Figure 4-1.

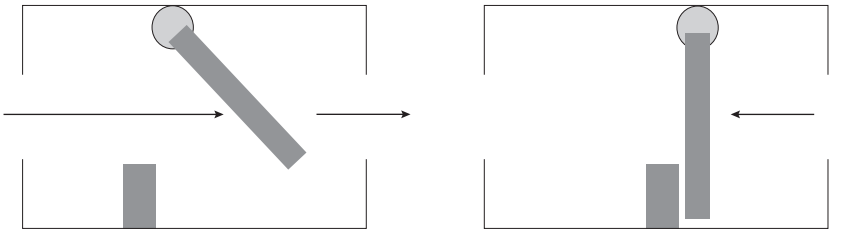


Figure 4-1: A one-way water valve. When the water flow is forward-biased (left), it pushes open the valve and passes through. When reverse-biased (right), the water pushes the valve into the block, sealing it shut so nothing can flow.

Here, a gate mounted on a hinge is free to swing to the right, but it can't swing to the left because its movement in this direction is blocked. If water pressure is applied from right to left, it simply pushes the gate shut even more firmly, so nothing can flow. We'll call this flow direction *reverse-bias*. When pressure is applied from left to right, it pushes open the gate and allows current to flow. We'll call this flow direction *forward-bias*.

The valve isn't as simple as it first appears. Imagine the water is made of individual mass particles, pushing their way through. As a particle strikes the gate, some kinetic energy is transferred from the particle to the gate. After the particle passes through, the gate still has this energy. In the absence of any damping, the gate will bounce off the top of the pipe, then off the blocker on the bottom, and it will keep oscillating open and closed forever. A more realistic model includes damping, where the gate may begin to oscillate but its kinetic energy is quickly absorbed by the block and emitted out of the system as lost heat. A system must emit heat as the price for being one-way.

Meanwhile, the particle that passed through lost some of its velocity when it transferred energy to the gate. Because the output particles have less energy than the input particles, we'll need to do work on them to add some energy back if we want to use them as input to a second valve with the same physics. Adding this energy compensates for the heat lost.

You could make current flow in the reverse-bias direction if you *really* work hard to push it. You'd need to push hard enough to smash the block. This would likely make a loud bang and permanently destroy the device. We'll see similar behaviors in electrical analogs of the water valve, which we turn to next.

Heat Diode

A *diode* is any electrical system that enables current to flow in one, forward-bias direction and not (easily) in the other, reverse-bias direction. The first diodes were vacuum tube heat diodes, built in the electromechanical ("diesel") age; they're easier to understand than modern ones, so we'll start there.

A vacuum tube diode (Figure 4-2) looks a bit like an old-fashioned filament light bulb. Two external wires connect to the components: the *cathode* (something that emits electrons) and the *anode* (something that absorbs electrons). Here, the cathode is a metal core and the anode is a cylindrical wrapper around it, separated from it by a vacuum. The cathode is heated up by an external energy source.

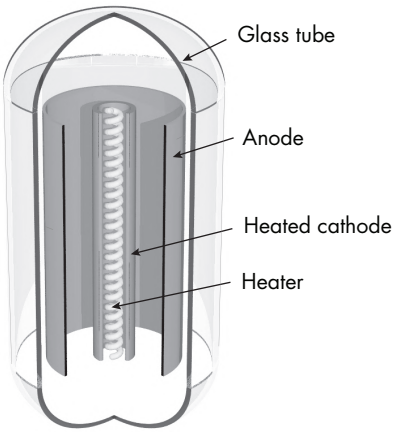


Figure 4-2: A vacuum tube diode

When a voltage is applied in the forward-bias direction, electrons flow into the cathode from outside. They are heated up by the heater, which gives them enough energy to fly out of the metal cathode and across the vacuum to be picked up by the anode. This is current flowing across the diode.

When a voltage is applied in the reverse-bias direction, electrons arrive at the anode from outside, but they don't have enough energy to fly out of the metal and across the vacuum because the anode isn't heated. Current doesn't flow in this direction.

To create the directionality in this system, we have to do work on it by inserting energy in the form of heat. This heat energy is then dissipated into the outside environment.

You could make current flow through the vacuum tube in the reverse-bias direction if you *really* work hard to push it. You would need a very high voltage to persuade electrons to jump off the anode. This is likely to make a loud bang and permanently destroy the device.

p-n Junction Diode

Most diodes used today aren't vacuum tubes, but are instead formed from *p-n junctions* on silicon, where *p* and *n* stand for positive and negative charge regions, respectively. To understand how p-n junctions work, we'll need a quick crash course on semiconductor chemistry and physics.

A Semiconductor Crash Course

Basic electronics divides materials into *insulators*, which don't conduct electricity, and *conductors*, which do. *Semiconductors* are materials that in their normal states are insulators, but that can be coaxed into becoming conductors through a very small change. *Silicon (Si)*, element 14 of the periodic table, is a semiconductor. You can see a representation of a silicon atom in Figure 4-3.

A silicon atom has 14 positive protons and 14 negative electrons. The electrons come in three concentric shells. The innermost is a full shell of two electrons, the middle is a full shell of eight, and the outermost is a half-full shell, having four of eight electrons.

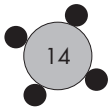


Figure 4-3: A silicon (Si) atom has 14 electrons, 4 of which are visible and available for interactions in the outer shell.

Quantum mechanics (a topic beyond the scope of this book) shows that atoms are in a low-energy state when their outermost shell is full. Informally, low-energy states are called *happy* and high-energy states *unhappy*. This anthropomorphism reflects the appearance of physical systems to "want" to move from unhappy to happy states. The "wanting" is a consequence of statistical physics, which shows that there are more ways to be happy than unhappy, so the system is more likely to find its way into a happy state.

Happy states are highly probable because moving into them is a directional system. When electrons move from an unhappy to a happy energy state, they give off the excess energy as a photon, usually lost as heat. To get the electron back to the high-energy state, you would need to find a similar or higher-energy photon and shoot it back at the atom, which is unlikely unless you work to make it happen. These probabilities function as a chemical force acting on the electrons, pushing them into configurations that have full outer shells.

The happiest state for a set of silicon atoms is thus for them to share electrons in their outer shells via covalent bonds. Each atom bonds to four neighbors by sharing an electron pair, ostensibly giving each atom a full outer shell of eight electrons. This can be drawn in 2D as a regular square grid of atoms, as in Figure 4-4.

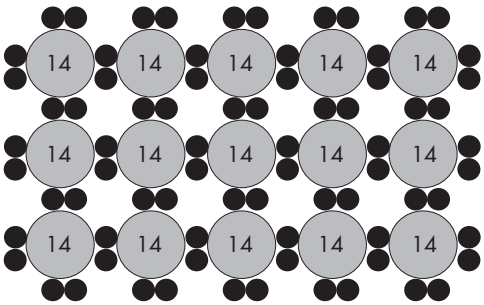


Figure 4-4: Silicon atoms form a crystal lattice, sharing electrons to fill their outer shells with eight electrons.

In the real, 3D world, however, the structure is tetrahedral, with the four neighbors positioned in different 3D directions, as in Figure 4-5. This structure is called a *crystal lattice*, and it's very strong and stable. (For carbon, the crystal form is called diamond. Silicon crystal has some similar properties, but it's easier to work with and substantially cheaper to obtain.)

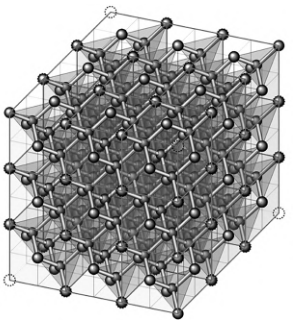


Figure 4-5: The silicon lattice is really 3D and tetrahedral.

Silicon crystal doesn't conduct because all the electrons are happy to be where they are and don't need to move around to lower their energy. However, we can make a silicon crystal conduct, like a metal, by adding just a very small number of different atoms into its lattice. This process is called *doping*. Consider doping with either of silicon's neighbors on the periodic table: aluminum (Al), element 13, having three electrons in its outer shell, and phosphorus (P), element 15, having five electrons in its outer shell. Doping with aluminum gives rise to a net shortage of electrons in the crystal, called *p-doping* (*p* for positive). Phosphorus gives rise to a net surplus of electrons in the crystal, called *n-doping* (*n* for negative). The doped crystals are still electrically neutral: they contain equal numbers of protons and electrons. The shortage and surplus have to do only with the chemical state of the atoms wanting to have full outer shells.

In p-doping, some atoms will have "holes" in their outer shells where electrons are missing. In n-doping, some will have excess electrons that result in a fourth non-full shell appearing with only one electron in it (on top of the three full inner shells with two, eight, and eight electrons). Both types of doped silicon behave as metals. In n-doped silicon the excess electrons aren't tightly bound into the stable structure, and circulate freely between different atoms. This means they can flow through the crystal, and that it has become a conductor. Similarly, holes may circulate through p-doped silicon, making it into a conductor. This works even if the number of doping atoms is tiny compared to the number of silicon atoms.

How a p-n Junction Works

A p-n junction consists of a p-doped and an n-doped region next to one another, as in Figure 4-6.

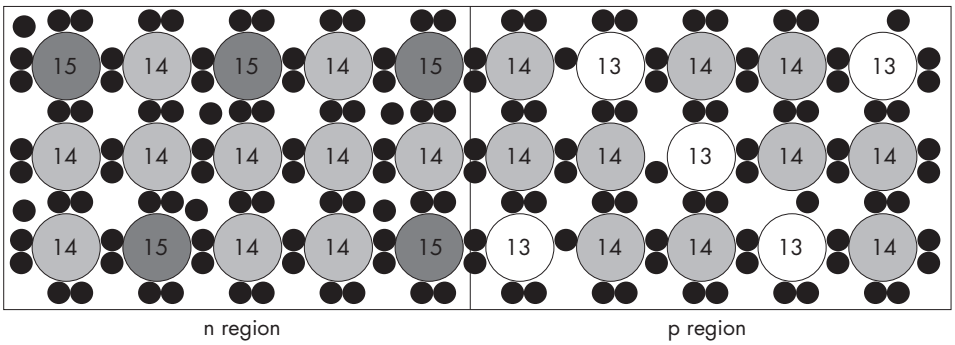


Figure 4-6: A p-n junction in a high-energy state, made by doping two regions of silicon

Here we see the surplus electrons around the phosphorous atoms (element 15) in the n region and the shortage of electrons around the aluminum atoms (element 13) in the p region. In this state, both regions are conductors, as they have either free electrons or holes, so current can flow across the junction.

When the junction is created, there's no effect on the parts of the crystals that are far from the boundary where they touch. But in the region close to the boundary—called the *depletion zone*—something interesting happens almost instantly. In this zone, the excess electrons on the n-doped side experience a chemical force that attracts them across the boundary to complete the outer shells on the p-doped side, as in Figure 4-7.

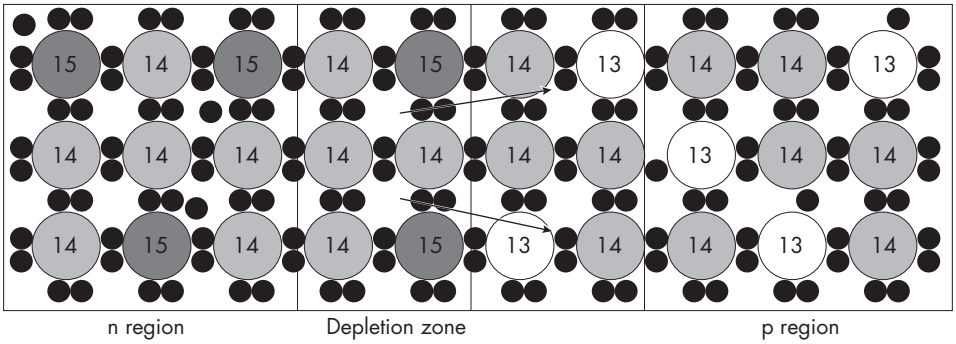


Figure 4-7: A p-n junction in a low-energy state, when electrons cross near the junction, giving off energy in the process

This chemical force is strong enough to overcome some of the electrical force, which usually keeps the electrons close to the protons to which they were originally matched. The chemical and electrical forces balance each other out at a point where the electrons cross only in the depletion zone. (If they crossed any further, the electrical forces would become stronger than the chemical ones and push the electrons back again.) The chemical force is strong enough to produce a stable, low-energy state in which the atoms in the depletion zone have full outer shells. The atoms are also ionized, since they have differing numbers of protons and electrons: there's a net positive charge in the n region and a net negative charge in the p region. Since this

is a lower-energy state than the starting state, photons are emitted and lost in the form of heat as the electrons enter their new positions. And because all the atoms in the depletion region have full outer shells, the region acts as an insulator (like pure silicon), so current can't flow across the junction.

The p-n junction functions like the water valve: in its high-energy state it's like an open valve, enabling current to flow; in its low-energy state it's like a closed valve, preventing current from flowing. Like the water currents pushing open the valve and lifting it against gravity, an electrical current flowing in the forward-bias does some work, adding energy back into the system and pushing it into its open, high-energy state. And like the water currents in reverse-bias pushing the valve firmly closed, an electrical current in reverse-bias pushes the system into the low-energy state, which doesn't conduct. This works as follows.

In forward-bias, extra electrons are pumped from outside into the n region. The depleted part of the n region is only *mildly unhappy* to receive them because they can bind to the shells of the element 15 ions, whose electrons were previously lost to the p region. They stop being ions and go back to being regular atoms. They're unhappy to not have a full outer shell now that a new electron has started a new shell, but this is almost compensated for by their becoming electrically neutral. As such, only a bit of work is needed to overcome this mild unhappiness and shove the electrons in.

Something similar happens in the p region as electrons are pulled out by the forward-bias; it's mildly unhappy because it loses its full shells, but it gains electric neutrality, which almost—but not quite—counters that loss.

To summarize, electrons have entered the n region and exited the p region, which means they've effectively flowed from the n side to the p side as current. We have now also returned the system to its original high-energy, conducting state, as in Figure 4-6, because we're back to having the original numbers of electrons on each side. We had to do some small work to overcome the atoms' unhappiness about being changed, and this work is equal to the photons that were lost as heat when we moved from the high- to low-energy state. Almost immediately after this has happened, the system will fall back to the low-energy state again, emitting new photons as heat as it does so. We'll need to continue to do work on the system to force further batches of electrons through it, and see that work come out as photon heat.

In reverse-bias, we try to pump electrons from outside into the p region side. In this case, the atoms in the p region depletion zone very strongly don't want the electrons. They aren't just mildly unhappy, but *very* unhappy to take them, because doing so would both spoil their full outer shells and also double-ionize them into 13^{2-} ions. So the incoming electrons don't go there. Instead, the *undepleted* part of the p region will take them, because this area contains element 13 atoms that are happy to be ionized into 13^{-} ions because this fills their outer shells. The effect of this is that the new electrons that arrive act to enlarge the depletion zone, as each newly ionized atom stops conducting because it has a full shell. This makes the p region even less conductive, the analog of the reverse-bias water current pressing on the valve to slam it shut even harder.

Something similar happens on the n side as we try to pump electrons out of this region. Here, the 15^{+} ions *really* don't want to give up electrons, as this would both destroy their nice full outer shells and also make them into more strongly charged 15^{2+} ions. So instead we end up pulling electrons out of the undepleted part of the N region. This makes the 15 atoms happy to have full shells, but turns them into insulators and again makes the depletion zone larger.

It's possible to force electrons to enter these very unhappy states and then to cross the junction in reverse-bias, but only if very large forces are applied to them. The system will resist the forces for a long time, entering a very high-energy state, then eventually break down, releasing all of that energy as the electrons cross. This is likely to make a loud bang and permanently destroy the device.

In a *light-emitting diode (LED)*, the photons emitted when electrons cross the junction and fall into lower-energy states have frequencies that are visible as light. Here it's especially clear that you have to do a small amount of work on the system by putting power in, in order to get the electrons to cross the junction and emit the photons. You may also see a larger emission of light and possibly sound and smoke if you try to force electricity backward through your LED.

Note that in this discussion we've deliberately considered the "flow of electrons" rather than "electric current," to make it as simple as possible to follow the analogy of the flow of water. Due to a very unfortunate historical accident, "electric current" was defined as the *negation* of the "flow of electrons," and is said to flow from anode to cathode rather than from cathode to anode. This is reflected in the diode symbol (Figure 4-8), where the arrow shows the direction in which current—not electrons—can flow. The bar at the tip of the arrow suggests that current flow in the opposite direction is blocked. Swapping the entrenched definition of current to reflect the flow of electrons would be about as hard as getting everyone in the UK to drive on the more sensible right side of the road.

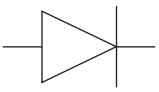


Figure 4-8: The diode symbol

Switching

Directional systems are the building block for our next level of architectural structure: switches. A switch allows us to turn a flow on and off automatically using another flow. Once again, we'll consider this general principle with a simple water example before transferring it to electronics.

Water Current Switch

Consider two water valves placed in sequence, with both of their blocks replaced by a spring-loaded moving platform, as in Figure 4-9.

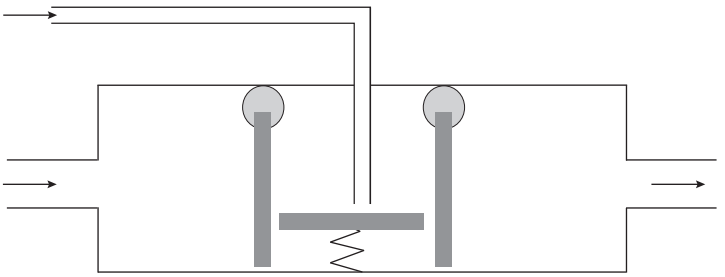


Figure 4-9: A water pressure switch

When the platform is up, the left valve can't open to the right, nor can the right valve open to the left, so water can't flow in either direction. We'll call the region between the two valves the *base*, the water pipe entering from the left the *emitter*, and the water pipe exiting on the right the *collector*.

If we were to connect a small additional pipe to the base (as shown in Figure 4-9), we could then force a current of water through this pipe and into the base. This would push down the spring-loaded platform, allowing both valves to open, which in turn would allow water to flow in either direction along the main pipe. This creates a switch: by turning the current in the small pipe connected to the base on and off, we can control when current can flow along the main pipe.

Consider the energy used in this system. We have to put energy into our switching current, which must come out somewhere. In this case, energy has gone into the spring, and when we stop piping water into the base this spring will bounce back up and give off heat as it damps down. Also note that the water we've piped into the base has to go somewhere: it joins the main water current from the emitter pipe and leaves through the collector pipe.

Electrical Tube Switch

As the water switch extends the water valve, electrical tube switches extend electrical heat diodes by using one electrical current to control the flow of another electrical current. This works by inserting a metal grid in the middle of the vacuum between the cathode and anode in the heat diode, as shown in Figure 4-10.

Like the heat diode, the tube switch also looks like a filament light bulb. The added metal grid is connected to a third "base" wire. If you push electrons down this wire into the grid, they make the grid negatively charged. This prevents other electrons from jumping from the cathode to the anode, as negative charge repels negative charge. If you release the electrons from the base, then the tube behaves exactly as a heat diode and enables current to flow from the cathode to the anode.

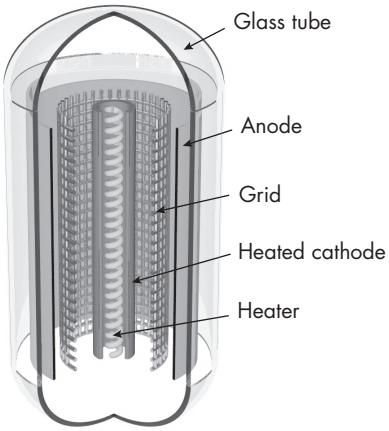


Figure 4-10: A vacuum tube switch

Electrical tube switches are confusingly known as *valves* in some contexts, though they're analogous to water switches rather than to water valves. These are the switches used in early electrical computers such as ENIAC. They aren't ideal for practical computing, however, because they require vacuums inside highly breakable glass bulbs, and also require messing around with heat, which can make them overheat and explode; as you can imagine, they need to be replaced frequently. Electrical tube switches are found in tube amps (or valve amps) for electric guitars, where they're used for their analog qualities rather than the digital ones relevant to computing. (So you could build a computer out of old Marshall amp tubes—a nice project!)

p-n-p Transistor

The *p-n-p transistor* is a better way to make electrical switches; it avoids many of the practical problems of vacuum tubes. Its design is based on the p-n junction diode. As the water current switch can be viewed as two mirror-image water valves stuck together with a base pipe attached to their center region, a p-n-p transistor can be viewed as two mirror-image p-n diodes stuck together to form a p-n-p sequence, with a base wire attached to the central n region. The transistor is shown in its high-energy state in Figure 4-11.

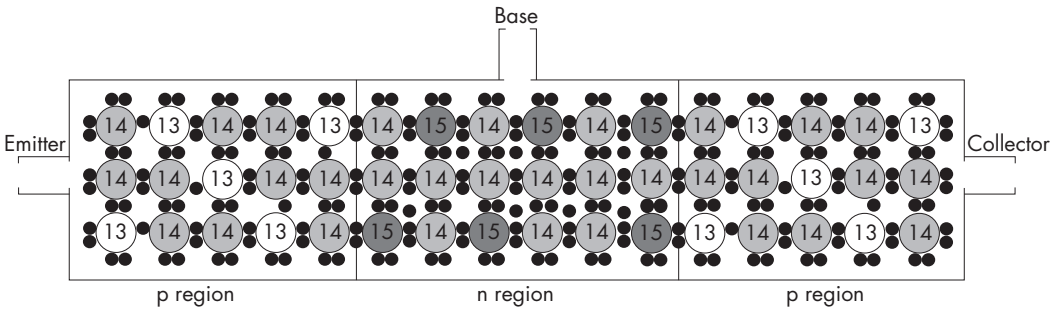


Figure 4-11: A p-n-p transistor in its high-energy state

The two junctions between p and n regions are analogous to the valve gates in the water switch. A switching electrical wire into the base (n) region between the boundaries is analogous to the switching water pipe into the base region between the water valves. The wire containing current entering from the left is the emitter, and the wire carrying it out on the right is the collector.

As water injected into the water valves' base region pushes open the two valves, electrons injected into the base region open up both p-n junctions. The work done to push them into the base lifts the system into its higher-energy, conducting state, enabling electrons to flow across the transistor from the emitter to the collector. The transistor thus acts as an electrical switch, with the electrons pumped into the base switching on the flow of electrons from the emitter to the collector.

NOTE *Like vacuum tubes, transistors have analog properties that can be used in audio amplifiers, such as transistor radios and more modern guitar amps. As with vacuum tubes, we're interested only in their digital qualities here.*

As with the water switch, there's a cost to this process. It requires energy to inject current into the base and kick both junctions into their high-energy, conducting states. This energy is later given off as photons (heat) when both diodes fall back into their low-energy states, shown in Figure 4-12.

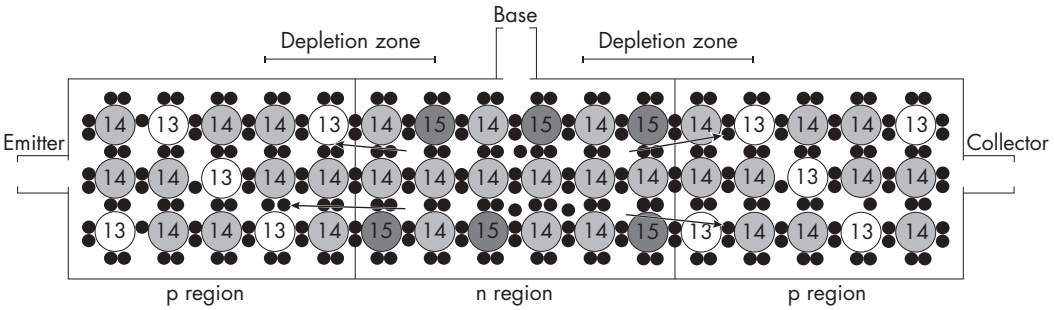


Figure 4-12: A p-n-p transistor in its low-energy state

Like the water switch, the injected base current also has to go somewhere, and its electrons flow out of the collector along with the main current from the emitter.

The standard symbol for a transistor, with E, C, and B meaning emitter, collector, and base, respectively, is shown in Figure 4-13.

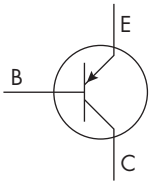


Figure 4-13: The p-n-p transistor symbol

As with diodes, we've discussed transistors in terms of the "flow of electrons" rather than "electric current," to preserve the water analogy. In fact, electric current flows from collector to emitter in our p-n-p transistor, whereas the electrons flow the other way around.

NOTE *It's also possible to make n-p-n transistors that use the regions the other way around–that is, they work by pulling electrons out of the base to open the gates. Again, in this case, electrical current is said to flow into the base region to open the n-p-n transistor.*

Early silicon chips used p-n-p transistors, but they're inefficient because of the loss of electrons from the base into the collector. Modern chips use a modified device, a field-effect transistor, to improve this. We'll again introduce this idea using its water analogy, then translate it to semiconductors.

Water Pressure Effect Switch

In the water switch, water from the base is lost to the collector because it's pushed into the switch and joins the main flow of water from the emitter. We can address this inefficiency by covering the end of the base pipe with a rubber membrane where it joins the base region, as in Figure 4-14.

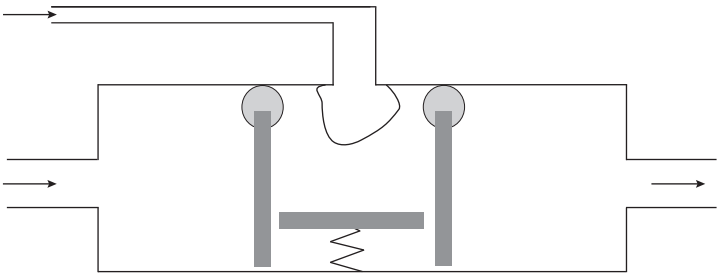


Figure 4-14: A water pressure switch

The rubber can stretch, but it doesn't allow water to flow through it. When work is done to pump water into the base pipe, the pressure created will stretch the membrane so it expands into the base region. This will displace the water in the base region and force it through both valves, again enabling a main water current to flow from the emitter to the collector. When the pressure on the base is released, the membrane will shrink back, releasing the pressure on the valves so they close and turn off the main water current.

The advantage of adding the membrane is that the water pumped into the base is no longer lost to the collector. No water leaves the base. The water in the base acts only to temporarily apply pressure on the valves. Less moving stuff means that less energy is wasted, so the system can run smoother and faster.

Work is needed to push water into the base. This is converted into potential energy to lift up the valves against gravity. Then it's lost as heat as the valves close, bounce around, and damp down.

Field-Effect Transistors

Field-effect transistors (FETs) are the exact analog of water pressure switches, as p-n-p transistors are to water current switches. The FET improves on the p-n-p transistor by covering the end of the base wire with an electrical insulator (such as silicon oxide, SiO_2) where it joins the n region, as in Figure 4-15.

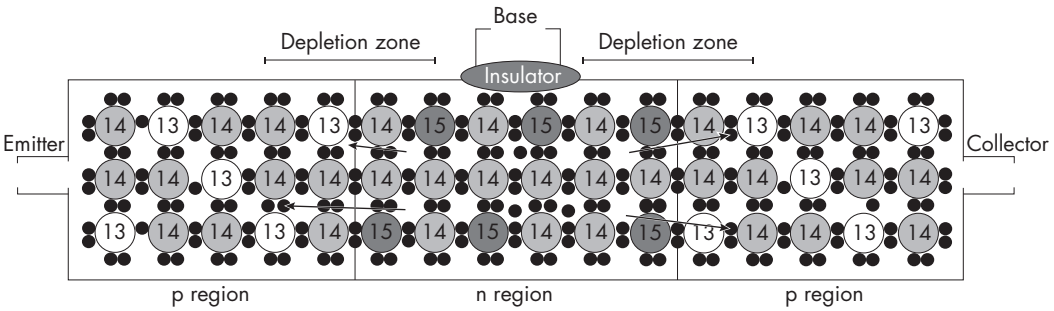


Figure 4-15: A FET transistor in its low-energy state

The insulator allows an electrical *field* to transmit through it, but it doesn't allow electrons to flow through it. This means an electron on one side can push away an electron on the other side, without the electron itself crossing over. When work is done to push electrons into the base wire, the negative charge accumulated in the base will push away electrons in the n region, forcing them through both p-n junctions, again enabling a main electron flow from the emitter to the collector. When the voltage on the base is released, the electrical field across the insulator will shrink back, releasing the voltage on the p-n junctions so they close and turn off the main electron flow.

The advantage of adding the insulator is that the electrons pumped into the base are no longer lost to the collector. No electrons leave the base. The electrons in the base act only to temporarily apply voltage rather than electron flow on the p-n junctions. Less moving stuff means that less energy is wasted and that the system can run smoother and faster.

Work is needed to push electrons into the base. This is converted into potential energy to raise the p-n junctions from their low- to high-energy states. Then it's lost as heat as the junctions fall back to their low-energy states, giving off photons.

Clocks

Often we want to automatically switch a signal on and off regularly over time. Such a signal is called a *clock*, and it can take the form of a binary input that oscillates in a square wave over time, as in Figure 4-16.

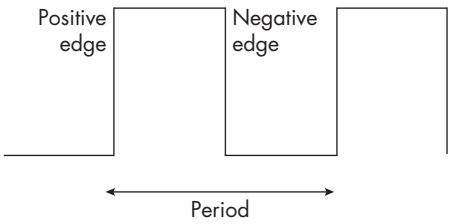


Figure 4-16: A square wave clock

Fast electrical clocks can be made from materials having *piezoelectric* properties, meaning that they mechanically oscillate in response to a voltage placed across them. These oscillations in turn change the material's electrical resistance and create an oscillating voltage as a result. Quartz crystals and some ceramics have this property, with oscillations in the megahertz (MHz) to gigahertz (GHz) range depending on their exact structure and the voltage applied to them. These can be made into clock units by adding hardware to apply the required voltage across them and to rectify their signals into the square waves needed for clock signals.

We'll depend on clocks to drive the "sequential logic" structures in Chapter 6. These are structures whose state can update at regular time intervals. Sequential logic structures, in turn, form subcomponents of CPUs. Physical clocks are thus very important to computation and can be found on motherboards in modern electrical computers, as Figure 4-17 shows.

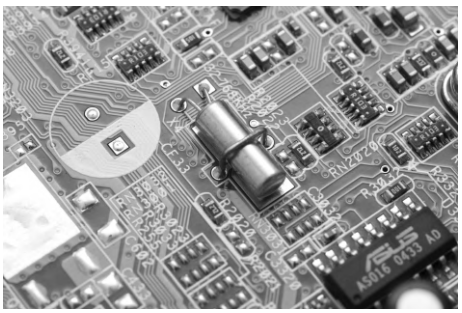


Figure 4-17: A quartz crystal oscillator

These clocks can be bought for a few dollars on eBay to mount on a breadboard in your own projects (search for terms like "quartz crystal oscillator").

Fabricating Transistors

Modern integrated circuits use FET transistors, created on silicon chips. A "chip" of silicon is a small, very thin slice, much like a potato chip (known as a "crisp" in the UK). Silicon is an abundant element that can be obtained as sand from the beach. Once purified, it can be formed into sausage-shaped lumps called ingots. Ingots are sliced like salami into large, very thin slices called wafers. Each wafer is later cut into many small, thin, square chips.

The process of creating transistors and wires and connecting them together on wafers is called *fabrication*. A wafer presents a two-dimensional surface on which transistors are laid out. Tiny metal wires are added to connect them together.

The same masking concept used for printing T-shirts and PCBs that we discussed in the introduction is used, albeit in miniature, to fabricate *application-specific integrated circuit (ASIC)* silicon chips. Unlike with PCBs, the components themselves—transistors—are fabricated along with the wiring. You design circuit layouts in a CAD program, using a fixed number of doping chemicals to form the different regions of each transistor, and copper to form wires connecting the transistors. You then print out one binary image for each chemical, onto a transparency, to show where its atoms will go onto the 2D silicon surface. This transparency is used to create a physical mask, which allows particles to pass through in the desired areas and blocks them in undesired areas.

You lay your mask on top of a blank wafer of silicon and pour atoms all over it. The atoms will pass through onto the wafer only in the allowed areas of your design. You allow this chemical layer to dry, and repeat the whole process for each chemical to build up the design. Usually the transistors are laid down first via masks that create the doped regions in the silicon surface itself. Further masks are used to build up metal wires connecting the transistors above the silicon surface. Figure 4-18 shows a single FET transistor and its wires formed on a silicon chip.

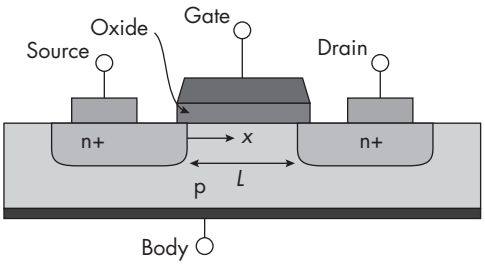


Figure 4-18: A single n-p-n FET transistor formed on a chip, shown as a cross section of the silicon surface, with chemical layers made in and on the surface

Fabrication is difficult and expensive. Rather than just "pouring" atoms onto the mask, they need more energy to smash their way into the silicon lattice, which may involve a particle accelerator. In addition to, or instead of, aluminum and phosphorus, many other chemicals are used for doping, such as germanium, boron, arsenic, gallium, lithium, indium, and the heavy metals antimony and bismuth. These other chemicals have similar properties to aluminum and phosphorus, but they're easier to work with. Unlike T-shirt printing, fabrication can also make heavy use of subtractive processes, which use similar masks to apply chemicals that remove rather than add layers.

Traditional fabrication required the wires to not cross one another; they had to be laid out in 2D circuits, with the wires going around one another. This was a major driver for network theory algorithms research to find optimal layouts. Modern fabrication allows for limited crossing of wires, such as via 20 alternating layers of copper and insulators laid down by masks in a kind of 3D printing, as in Figure 4-19.

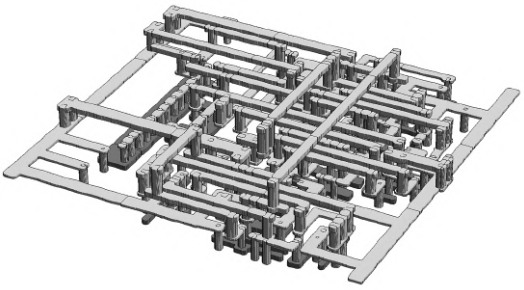


Figure 4-19: Some 3D copper wiring laid down above transistors in silicon

Most systems create FET transistors from a particular selection of chemicals, and so the devices are known as *MOSFETs (metal-oxide-semiconductor FETs)*. They typically use a particular style of masking sequence known as *CMOS (complementary metal-oxide semiconductor)*. A modern CMOS process might have around 300 masks applied in a specific sequence of additive and subtractive layers. In 2018, a fabrication plant cost around $5 billion to build, and producing one mask set cost around $5 million. You *really* don't want to have any bugs in your circuit design by the time you send it to a fabrication plant, or you'll need to pay another $5 million to redo the mask set.

Moore's Law

Fabrication technologies have advanced rapidly throughout the transistor age, roughly doubling the amount of transistors that can be created per unit area of silicon every two years. As you saw earlier, this empirical observation is known as *Moore's law*, after Intel's Gordon Moore, who first noticed it. Early chips had a few thousand MOSFETs, with around 2,250 in the 4-bit Intel 4004 shown in Figure 4-20, connected into circuits by non-overlapping

copper wires. Modern chips have billions or even trillions of MOSFETs, connected by overlapping, 3D copper wires.

As transistors got smaller, they also got faster, being clocked at higher speeds, so until the 64-bit era "Moore's law for clock speed" was often stated as a doubling of clock speed every two years.

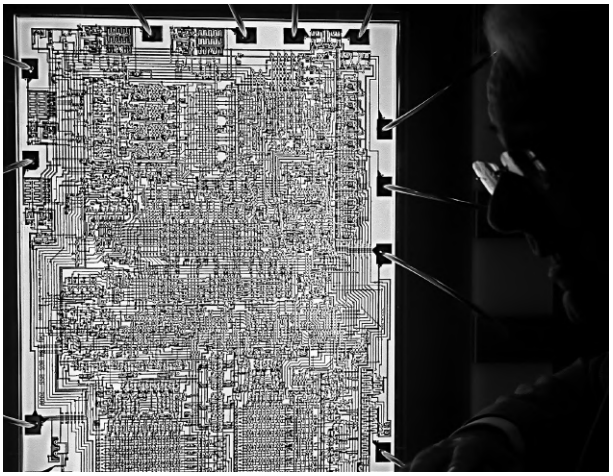


Figure 4-20: The layout of the 4-bit Intel 4004 processor chip, photographed with its designer, Federico Faggin

Some thought Moore's law would last forever in both forms, but as we've seen in our study of directional systems, there are fundamental links between switching, computation, power consumption, and heat. The faster we switch transistors on and off, the more heat will be generated, as there is a relationship between clock frequency f, capacitance C, voltage V, and power usage P:

$$P = CV^2f$$

As a result, the two forms of Moore's law have diverged since the start of the 64-bit era. This was known as "hitting the power wall," and it has been a major driver of recent changes to architecture that you'll meet in Chapter 15. The original law, of doubling the density of transistors, has continued to hold; while there are more transistors needing to be powered, they're also smaller, so each one uses less power and the total power use remains similar. Meanwhile, clock rates flattened out at around 3.5 GHz. You can already fry an egg on a 3.5 GHz CPU. But if Moore's law for clock speed had continued through the 64-bit era, CPU temperature would have reached that of the surface of the sun by 2010.

Summary

Switches are directional systems used to cause desired computations to happen. Directional systems must use up organized energy and emit heat. Modern electronic computers are built from FETs as switches, connected together by copper wires. The transistors and the wires are fabricated onto

silicon chips by expensive and complex masking processes. Throughout the transistor age, Moore's law has observed a doubling of transistor density every two years, made possible by advances in fabrication. This continues to hold, though increases in clock speeds ended in the 64-bit era due to energy and heat limits.

To make sense of large, complex networks of transistors, architects chunk them into higher-level structures, beginning with logic gates, which we'll study in the next chapter.

Exercises

Poor Man's AND Gate

Why can't you use a single transistor as an AND gate? (Hint: Consider the energy, electrons, and heat going in and out. Also consider the water switch equivalent as a simpler case.)

Challenging

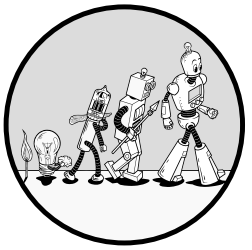
Try to visit a fabrication plant. Some may have visitor tours, especially if you ask as an organized group such as a student society. They are not just in Silicon Valley; for example, there are several in the UK. For an international list, see *https://en.wikipedia.org/wiki/List_of_semiconductor_fabrication_plants*.

Further Reading

- For a beautiful book on the fundamental physics of computation, see Richard Feynman, *The Feynman Lectures on Computation* (Boca Raton: CRC Press, 2018). The book includes detailed but intuitive discussions of the links between energy, heat, information, and computation. It's also the original source of the concept of quantum computing, which relies on these ideas.
- Another beautiful read is Richard Feynman, "Lecture 46 on Rachet and Pawl," in *The Feynman Lectures on Physics* (Boston: Addison–Wesley, 1964), *https://www.feynmanlectures.caltech.edu/I_46.html*. This lecture explores the general physics of directional systems.
- For an animated look at how chips are manufactured, see Intel, "The Making of a Chip," YouTube video, 2:41, May 25, 2012, *https://www.youtube.com/watch?v=d9SWNLZvA8g*.
- If you're interested in the analog properties of tubes and transistors, including their use in audio amplifiers and digital switches, see Paul Horowitz and Winfield Hill, *The Art of Electronics* (Cambridge: Cambridge University Press, 1980).

5

DIGITAL LOGIC



Switches such as the semiconductor transistors seen in the previous chapter are the most basic building block of modern electronic computation. But architects don't usually think in terms of switches. Instead, they build up a hierarchy of more complex structures from them, eventually forming a CPU. The next layer of this hierarchy consists of logic gates: devices formed from a few switches in standard circuits representing basic Boolean functions, such as AND and OR. Logic gates, in turn, can be used to build up larger structures such as simple machines for arithmetic and memory.

In this chapter, we'll examine some common types of logic gates and see how they're constructed from switches. We'll discuss how universal gates such as NAND can replace all the others and how Boolean logic can model and simplify circuits made from logic gates. But first, a little history.

Claude Shannon and Logic Gates

By 1936, complex electronic switching circuits were in heavy use in telephone exchanges. These circuits automated the work previously performed by human telephone operators, making and breaking connections between users' telephone wires to enable their calls. For example, a circuit might calculate functions such as, "If the caller has sent us a sequence of pulses encoding number 024 680 2468, and there is an available wire from here to the exchange managing 024 numbers, then connect the caller to the available wire and transmit 680 2468 in binary on it until the exchange replies with a connection code, and start billing them. Otherwise, connect the caller to a line-busy signal." These call-routing functions grew in complexity as more telephones, wires, exchanges, and companies connected to the network. There was an urgent economic need to reduce their wiring and complexity if at all possible. Many hacks existed for replacing complex groups of switches with simpler ones that seemed to have the same function, but how to do this reliably or optimally wasn't understood.

As we saw in the last chapter, switching devices use energy, so the energy of their outputs is less than that of their inputs; this made it difficult to reuse an output of one switch as an input to the next. For example, an electrical implementation using 0 V and 5 V to represent binary 0 and 1 as inputs will produce something like 0 V and 4.9999 V as outputs, because the switching mechanism loses some of the energy and voltage. If you build a large system from many switches, these voltage drops will accumulate until the output is no longer recognizable as representing the binary code.

All this changed in the great computing year of 1936, when Claude Shannon began his master's degree at MIT, which produced arguably the greatest master's thesis of all time. Shannon's thesis introduced two innovations to computer architecture that solved the switch simplification problem.

First, it defined a method to organize groups of switches into a new higher-level abstraction, the *logic gate*. Logic gates are defined as devices that take a representation of one or more binary variables as inputs, and produce one or more binary outputs using the *same representation*. Simple switches are *not* logic gates because they lose energy, so the output representation has lower energy and is different from the input representation. In contrast, a logic gate must top up any energy lost from switching so that its output coding is exactly the same as its input coding. This property enables the output of one gate to be cleanly used as the input to the next gate, and thus for arbitrarily long sequences of gates to be connected together without having to worry about noise introduced by the energy loss at each step.

It's much easier for a circuit designer to think in terms of logic gates because they no longer have to keep track of the lower-level energy considerations. Shannon showed how to implement logic gates from the switch technology of his day (electromechanical relays), but they can be implemented using many technologies, including water switches and modern metal-oxide-semiconductor field-effect transistors (MOSFETs).

Second, Shannon showed that any computation could be performed by combining instances of small sets of standard logic gates, such as AND, OR,

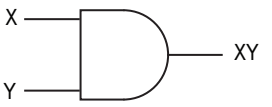
and NOT. He showed how to map these gates and their connections onto the mathematical logic of George Boole, *Boolean algebra*, which had been discovered 100 years earlier, around 1836. Boole's theory can be used to find equivalent expressions to complex circuits to simplify them and retain the same functionality using fewer gates and switches.

NOTE *As if this work weren't enough for one lifetime, Shannon also went on to invent communication theory, an entirely separate and equally brilliant contribution to computer science. Smart guy.*

Logic Gates

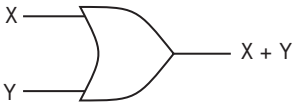
In modern terms, a logic gate is any device that has some binary inputs and some binary outputs and doesn't contain any memory, where the inputs and outputs use exactly the same physical representations for two symbols, 0 and 1. A logic gate's function can be completely and deterministically described by a *truth table*, which lists the resulting outputs for each configuration of the inputs. You'll see some examples soon.

It's possible to invent infinitely many different logic gates, but the most common ones today, and those studied by Shannon, are those with only one or two inputs and only one output. These standard gates include AND, OR, NOT, XOR, NOR, and NAND gates. Figures 5-1 to 5-6 show these gates and their truth tables.



X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5-1: An AND gate and its truth table



X	Y	X + Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 5-2: An OR gate and its truth table



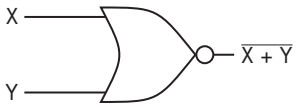
X	$\overline{X}$
0	1
1	0

Figure 5-3: A NOT (inverter) gate and its truth table



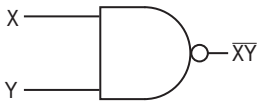
X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5-4: An XOR (exclusive OR) gate and its truth table



X	Y	$\overline{X + Y}$
0	0	1
0	1	0
1	0	0
1	1	0

Figure 5-5: A NOR gate and its truth table



X	Y	$\overline{XY}$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 5-6: A NAND gate and its truth table

Each gate's truth table lists every possible combination of inputs in the left-hand columns, and shows the corresponding output in the rightmost column. For example, the AND gate's output is 1 if and only if both of its inputs X and Y are 1. For any other combination of inputs its output is 0.

The names and functions of these gates are intended to mimic our human sense of logical combinations, with 1 corresponding to truth and 0 to falsehood. For example, the AND gate says that X AND Y is true if and only if X is true and Y is true. An XOR gate, short for "exclusive OR," requires *exactly* one of its two inputs to be true; the output is false if both inputs are true. This is distinct from regular OR, which is true if *either or both* of its inputs are true. (Students of digital logic have been known to reply "yes" to questions such as "Would you like beer or wine?") NOR stands for "neither X nor Y," and the output is true only when both inputs are false. NAND can be read as "not X and Y," and its truth table is the opposite of AND.

Logic gates can be connected together into networks to represent more complex expressions. For example, Figure 5-7 represents X OR (Z AND NOT Y), and will set the output to 1 if X is 1, or if Z is 1 and Y is 0. Note that "or" here is inclusive.

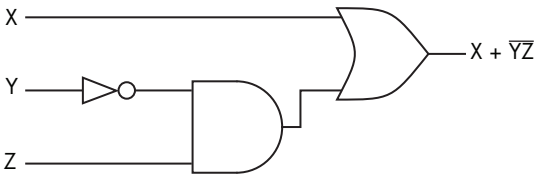


Figure 5-7: The logic gates for F(X, Y, Z) = X + $\overline{YZ}$

The network of gates in Figure 5-7 may be used, for example, in Shannon's telephone switching application, where it might be a circuit that disconnects a call if the receiver hasn't picked up after 30 seconds (X), or if the call previously began (Z) and the caller has no remaining credit (Y).

Identifying Universal Gates

In his research, Shannon wanted to identify a set of *universal gates*, a group of different types of logic gates that could be reconfigured to build any machine at the hardware level. He showed that several universal sets exist. For example, if you have a drawer containing an infinite number of AND and NOT gates, you can build anything from them. You could also do this with an infinite number of OR and NOT gates, but you can't build arbitrary functions from only a drawer of AND and OR gates. Most interestingly, a drawer containing only NAND gates, or only NOR gates, is universal. For example, Figure 5-8 shows how to build the standard NOT, AND, and OR gates from only NANDs. You'll get a chance to explore this figure more in an exercise at the end of the chapter.

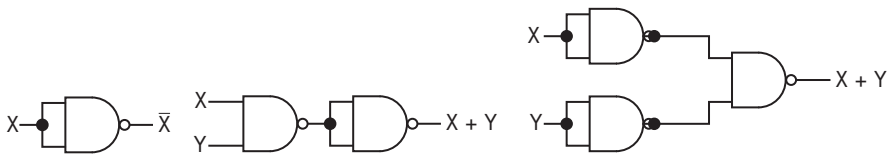


Figure 5-8: Building NOT, AND, and OR gates from universal NAND gates

Universal gates are important because they allow us to reduce the number of types of physical gates we need to manufacture down to just one. This is what we do in modern integrated circuits (ICs).

Making Logic Gates from Transistors

You might initially think that we could just use a single electrical switch, such as a transistor, as an AND gate. After all, a switch takes two inputs, its emitter and a base, and turns on one output, the collector, if and only if both

inputs are on, which is the definition of logical AND. However, we've seen that a switch must convert some of the incoming energy to heat, so the output is not quite in the same form as the inputs and can't be used directly as an input to the next logic gate. To keep the output in the same form as the inputs, we instead combine several switches, while using an external power source to keep topping up the energy that they lose as heat.

There are many different ways to do this. Shannon's original designs were optimized for electromagnetic relay switches rather than transistors. Modern chips use so-called *CMOS (complementary metal-oxide semiconductor)* style, which forms NAND gates from two positive-type and two negative-type transistors, as shown in Figure 5-9. With NAND as a universal gate, you can make all the other gates out of these CMOS NAND gates.

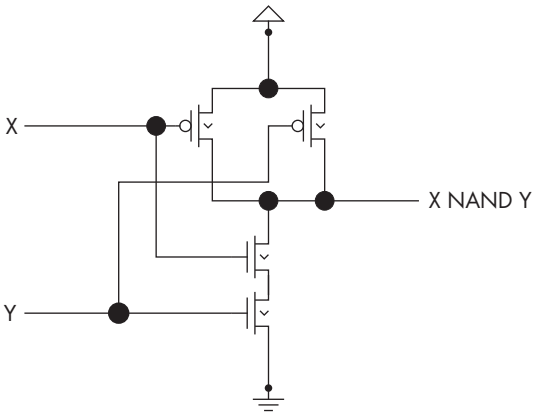
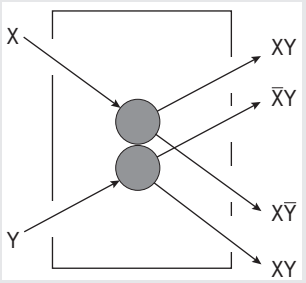


Figure 5-9: A NAND gate made from p-type and n-type transistors

An electrical *circuit* is a concept that exists at the transistor level, where electrons flow from the power source to ground, then are pumped back from ground to power by a power source, creating a closed loop. While it's common to informally refer to networks of logic gates as "digital logic circuits," this is technically incorrect because at that higher level of abstraction, the networks don't usually form closed circuits and instead can have arbitrary network topologies. If we were to implement the same networks using non-electronic implementations of logic gates, there might not be any circuit even at lower abstraction levels. When we draw diagrams and build systems made from logic gates, we should therefore more properly call them "digital logic networks" rather than "digital logic circuits."

MAKING LOGIC GATES FROM BILLIARD BALLS

Logic gates don't have to be made from transistors or even electricity. For example, billiard-ball computers are a theoretical invention where computation is done with balls in a complex geometric maze environment, in which versions of logic gates such as AND and OR are implemented through geometric structures and mechanics. The gates are arranged so that, for example, an AND gate tests for the collisions of two balls and directs one of them to the positive output only if the collision occurred, as in the following figure.



Due to the mechanical laws of conservation of energy, the billiard-ball computer models can be used to show that computation needs the same amount of energy and therefore the same number of bits of information to exit and enter. This isn't the case for a normal AND gate, which has two inputs and one output. The model shows that we should add a second "garbage" output for the exhausted second bit. This has the interesting property that it makes the computation *reversible* in the sense that the input could be reconstructed if we know the output. This allows us to run the machine backward. If that sounds strange, consider that there are many programming situations where it would be nice to have a reverse debugger that could undo the effects of recent lines of code.

The billiard-ball computer was intended to enable us to think clearly about the role of energy usage and heat in computation. This has become a big topic recently as the rise of portable computers dependent on small batteries has increased the need to conserve computational energy, and as concerns have mounted about the environment, fuel resources and costs, carbon emissions, and heat pollution. A conventional AND gate has two inputs and only one output, so one billiard ball's worth of energy is lost as physical heat every time we do an AND operation. The billiard-ball model suggests that we could build electric AND gates that don't waste energy if we were to keep track of a second output bit from the gate—the same bit needed to make it reversible. Heat is actually energy that we've lost track of, in this case by throwing information away. This is why your phone gets hot when doing heavy computation, and it's why your processor needs a large fan. The fan is pumping waste information in bits out of your computer's vents. (In this sense, the world running out of fuel isn't an energy crisis, but rather an information crisis. Energy can't be created or destroyed, but we can lose the information about where the energy is—information we'd need to make the energy do useful work for us.)

Putting Logic Gates on Chips

When you first look at a chip through a microscope, or at any computer built from any kind of logic gates, you won't be able to point to a single component and say, "That's a logic gate." What you'll actually see is a whole load of transistors, organized into gates. For example, take a look at Figure 5-10, which shows a microscope photo ("die shot") of a very simple silicon chip containing only four CMOS NAND gates, each formed from four transistors (as you saw in Figure 5-9).

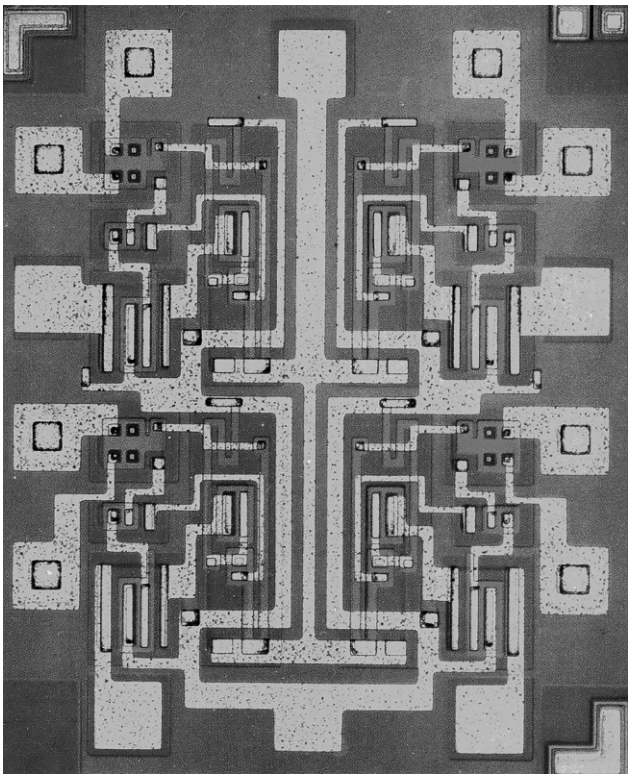


Figure 5-10: A die shot of a simple silicon chip containing four CMOS NAND gates

Figure 5-11 is a mask set for a single CMOS NAND gate, showing how to physically lay out the p- and n-doping regions along with the copper wiring.

Modern processors may have billions of transistors grouped into logic gates. But older-style ICs containing just a few logic gates are still manufactured and are very useful for building your own circuits. The 7400 TTL series is a famous example of such simple chips. Originally produced by Texas Instruments in the 1960s, they're now widely manufactured as generic products. Most chips in this series contain just a handful of logic gates of a single type, such as four AND gates, four NAND gates, or four NOR gates, as shown in Figure 5-12.

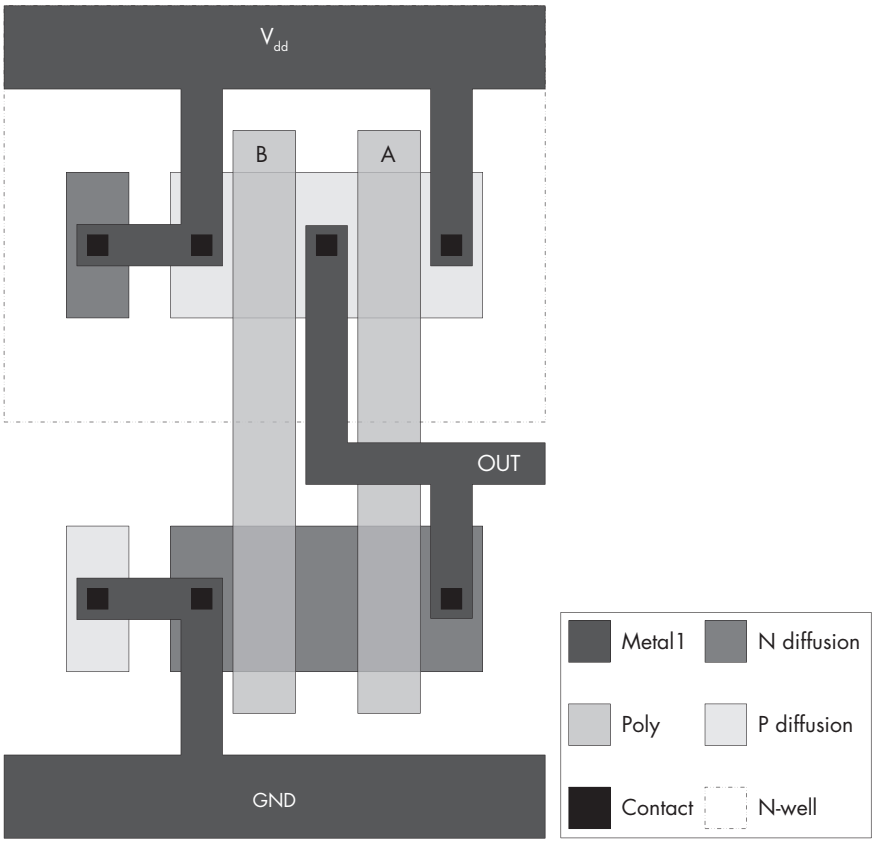


Figure 5-11: A CMOS NAND gate made from transistors and copper wire as a chip layout

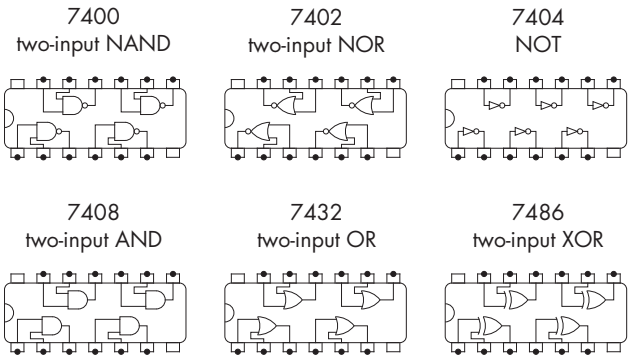


Figure 5-12: Some 7400 TTL series chips, each containing a few logic gates

These diagrams show the physical layouts and pinouts of the chips; to connect the logic gates you attach physical wires to the appropriate pins. You can buy bags of these chips for a few dollars on eBay and wire them up on a breadboard with a 5 V power source and ground, as in Figure 5-13, to make your own physical digital logic networks.

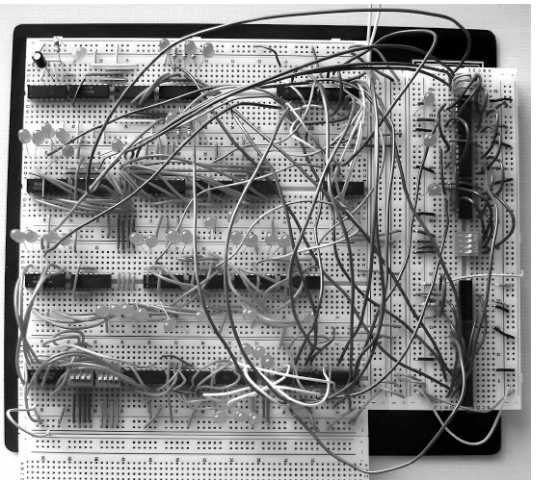


Figure 5-13: Building a digital logic network (forming a 4-bit CPU) using the logic gates on TTL 7400 series chips on a breadboard

You can see from Figure 5-13 that the wiring for digital logic networks can get quite complicated. Wouldn't it be nice if we had a way to simplify networks to use fewer gates and wires, while still performing the same functionality? This is where the next part of Shannon's innovation comes in: how to use George Boole's logic to perform such simplifications.

Boolean Logic

Logic allows us to formalize statements and inferences about truth and falsehood. It was developed by the ancient Greeks and remained largely unchanged until George Boole's work around 1836. Boole's work was picked up by Shannon in 1936, who realized that it could be used to model, simplify, and verify circuits built from his logic gates.

Boolean logic uses variable *names* to represent conceptual statements whose *values* are either true or false. It then provides connective symbols for AND, OR, and NOT, and rules that give truth values to expressions built from variables and these connectives.

Consider the following example. We have two variables: X represents the statement "God exists" and Y represents "snow is white." We can then

represent "God exists and snow is white" with X AND Y. Or we can represent "Either God doesn't exist and snow is white, or God does exist and snow is white" with (NOT X AND Y) OR (X AND Y).

Now let's see how to work with these statements.

Logic as Arithmetic

Boole discovered a structural similarity between logic and arithmetic. Previously, these had been two entirely different fields of study. Logic was an "arts subject" performed using natural language text and by studying rules to analyze arguments. Arithmetic was a "STEM subject" made of numbers and equations. As mathematicians had managed to unify geometry and algebra, so Boole managed to unify logic and arithmetic.

He did this by noticing that truth, represented by the symbol T, behaves like the number 1, and falsehood, represented by the symbol F, behaves like the number 0, if we replace AND with multiplication, OR with addition, and NOT with inversion about 1.

As we write $x + y$ for addition and xy for multiplication in arithmetic, we can thus use these same notations for OR and AND. When using this notation, it's common to also write $\bar{x}$ for NOT x, which corresponds to the arithmetic operation $(1 - x)$.

The similarity isn't quite perfect, because $1 + 1 = 2$ in arithmetic but we need $1 + 1 = 1$ in logic. Boole worked around this by choosing to work in a number system with only two numbers, 0 and 1, and by defining 1 plus anything to equal 1 within this system.

Using Boole's system, logical arguments can be converted into simple arithmetic. The advantage of doing this is that we know a lot about arithmetic already, in particular how to use laws such as associativity, commutativity, and others listed in Table 5-1 to simplify expressions.

Table 5-1: Useful Arithmetic Theorems for Simplifying Boolean Logic

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{(A + B)} = \overline{AB}$

For example, say we want to calculate the truth value of:

(F AND (T OR F)) OR ((F OR NOT T) AND T)

We can do it by converting the logic to arithmetic, then using standard arithmetic rules to simplify the expression:

$$
\begin{aligned}
&(0(1) + 0(0)) + ((0)1 + (1 - 1)1) \\
&= (0 + 0) + (0 + (0)1) \\
&= (0) + (0 + 0) \\
&= 0 + 0 \\
&= 0
\end{aligned}
$$

Finally, we convert the resulting number, 0, back to the logical value false.

This also works using variables rather than particular values; for example, the previous statement about God and snow can be written and then manipulated arithmetically as:

$$
\begin{aligned}
&((1 - x)y) + (xy) \\
&= (y - xy) + (xy) \\
&= y - xy + xy \\
&= y
\end{aligned}
$$

This can then be converted from the arithmetical number y back to the logical value Y. This shows that the truth of the statement is actually independent of the existence of God (X) and depends only on whether snow is white (Y). So assuming that snow is indeed white, the statement is true.

NOTE *The ability to move back and forth between logical truth values and integer 0s and 1s is now often used (or arguably misused) in languages such as C that play fast and loose with these types.*

Model Checking vs. Proof

We often want to know whether two Boolean expressions are equal. There are two main ways to go about determining this.

The first is called *model checking*. Given a potential equality, we simply compute truth tables for both the left side and the right side of the potential equation. If the truth tables match completely, then the expressions are equal. As an example, let's check that the AND form of the distributive law from Table 5-1 always holds. First, we calculate and compute the table for the left side of the equality, $A + BC$. We begin with three columns for our variables: A, B, and C. We then add a column for our intermediate term, BC, and use this to compute the value of the whole expression in the rightmost column, as in Table 5-2.

Table 5-2: The Truth Table for $A + BC$

A	B	C	BC	A + BC
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Next, we do the same for the right side of the equality, $(A + B)(A + C)$, in Table 5-3.

Table 5-3: The Truth Table for $(A + B)(A + C)$

A	B	C	(A + B)	(A + C)	(A + B)(A + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Finally, we compare the tables. Here we can see that for every possible assignment of values to the variables, the resulting value is the same in both tables. Therefore, by model checking, the left side equals the right side.

Model checking makes use of the *values* of the terms. If an equality has been shown by model checking, we say that it has been *entailed*, and that it is *true*, and we write $\models A + BC = (A + B)(A + C)$.

The second way to show equalities is by *proof*. If some equalities have already been established, such as the laws of Table 5-1, we can make use of their results symbolically without having to grind through the truth tables of everything. A proof is a list of transformations from the first to the second expression, where each transformation is justified by stating which law has been applied. If an equality has been shown by proof, we say it is *proved* and write $\vdash A + BC = (A + B)(A + C)$.

For example, here's one way to prove that $A + BC = (A + B)(A + C)$:

$$
\begin{aligned}
A + BC &= (1A) + (BC) && \text{: by AND identity law} \\
&= (A(1 + B)) + (BC) && \text{: by OR null law} \\
&= (A1) + (AB) + (BC) && \text{: by OR distributive law} \\
&= (A(1 + C)) + (AB) + (BC) && \text{: by OR null law} \\
&= (A(A + C)) + B(A + C) && \text{: by OR distributive law} \\
&= (A + C)(A + B) && \text{: by OR distributive law}
\end{aligned}
$$

NOTE *For Boolean logic it can be shown that any equality established through model checking can also be proved, and vice versa, so you can use either method according to taste. It may seem obvious that model checking and proof give the same answers in Boolean logic, but this isn't always the case for other logics, as found later by Gödel.*

The ability to check whether two expressions are equal isn't purely academic. Shannon recognized its value in simplifying his digital logic networks.

GEORGE BOOLE

George Boole published his books *The Mathematical Analysis of Logic* (1847) and *The Laws of Thought* (1854) a few years after Babbage's Engines. Boole grew up and formed his ideas in Lincoln, England. Unlike most historical academic heroes from rich families able to buy their way into Cambridge—such as Babbage and Turing—Boole came from an ordinary, poor family. His father was a shoemaker. Boole had no formal education, going instead to the public library and reading books to teach himself, *like you can do today*.

Boole created many new ideas outside the academic system, without that system's constraints on his thinking. In particular, no one told him that arts and sciences were supposed to be kept separate, so he would physically wander between both sections of the library, making comparisons between them. While his name is strongly associated with Boolean logic and the `boolean` or `bool` data types in modern programming languages, he also worked on probabilistic and other forms of reasoning, and was motivated by trying to understand and model human intelligence, as in modern AI and cognitive science. His real motivation for studying logic and other forms of reasoning was to formalize, analyze, and check the many arguments from classical philosophy, especially concerning the existence of God. He wanted to find out if these arguments were valid, breaking them down into their parts and testing each step so he could find out which of their conclusions were true and what to believe.

For example, here's part of Boole's logic for the existence of God (from *The Laws of Thought*, Chapter 13):

> Let x = Something has always existed.
> y = There has existed some one unchangeable and independent being.
> z = There has existed a succession of changeable and dependent beings.
> p = That series has had a cause from without.
> q = That series has had a cause from within.
> Then we have the following system of equations, viz.:
> 1st. $x = 1$;
> 2nd. $x = v\{y(1 - z) + z(1 - y)\}$;
> 3rd. $z = v\{p(1 - q) + (1 - p)q\}$;
> 4th. $p = 0$;
> 5th. $q = 0$:

Boole's short life—the founder of modern logic was killed by his homeopathic wife's theory of wrapping him in ice-cold blankets to cure pneumonia—was a subset of Babbage's, so they would likely have read each other's work. Boole wasn't interested in computer science, however. His ultimate interests were philosophical, and his work understanding and modeling intelligence was primarily intended as a contribution to the philosophical method. Still, he would have been aware that creating such formalisms would also enable them to be mechanized as AI, as discussed by Lovelace. It's a great shame the two never got together to develop this idea.

Simplifying Logic Circuits Using Boolean Logic

Shannon discarded Boole's conceptual interpretations of the variables, and instead showed that Boole's algebra could be used to simplify physical digital logic networks composed of logic gates. Simplification can include reducing both the number of gates and also the number of *types* of gates, such as reduction to all NAND gates.

This is done by translating a logic gate network into a Boolean expression, simplifying the expression using the laws of arithmetic, then translating the result back into a smaller logic gate network. Simplifying networks is useful because it reduces the number of transistors needed, which in turn reduces manufacturing costs and energy usage. Nowadays, CAD software is available that performs simplifications automatically: you input your digital logic network, click an icon, and get back a smaller and more efficient version.

For example, suppose we've designed the digital logic network on the left-hand side of Figure 5-14. Using Boole's theory this can be converted to an arithmetic expression and simplified to obtain $(A + B)(A + C) = A + BC$. This corresponds to the smaller network on the right side of Figure 5-14.

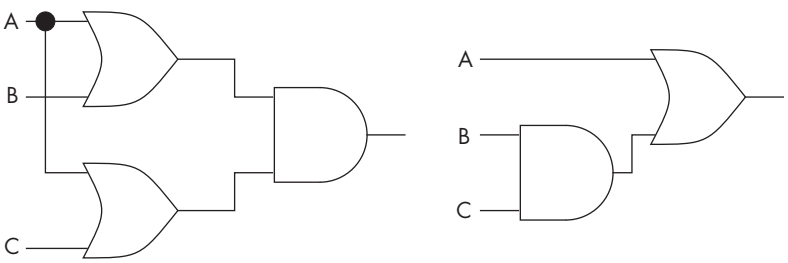


Figure 5-14: A logic network for (A + B)(A + C) (left) and reduced A + BC form (right)

We can use Boolean logic to further simplify the logic network to use only universal NAND gates, then reduce the number of NAND gates, as in Figure 5-15.

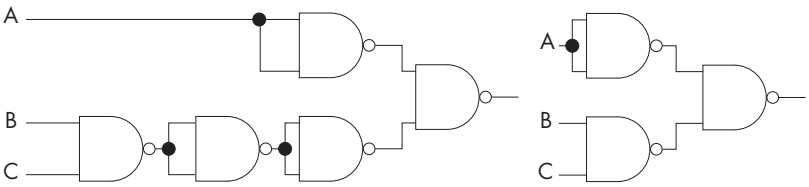


Figure 5-15: The Boolean logic converting A + BC to NAND gates (left) and then reducing them (right)

This allows us to convert any network into one that can be built more easily, using just one type of gate and as few of them as possible.

Laying Out Digital Logic

Once you've designed and simplified a digital logic network, you'll usually want to transfer it to real hardware. There are several ways to do this, which we'll examine here.

7400 Chips

One way to implement a simple logic network has remained unchanged since the 1960s: lay it out across a bunch of 7400 chips and connect them together with a rat's nest of wires.

As you saw earlier, each 7400 series chip contains a number of gates, usually all of the same type. Unfortunately, a single chip doesn't usually correspond to any particular topological region of your circuit. You need to consider each gate in your circuit and choose a specific gate on a specific chip to instantiate it. You can choose what goes where completely arbitrarily and your circuit will still work, but if you apply a bit of cleverness to the layout you'll be able to considerably reduce the length of wire needed to connect it.

For example, suppose you want to build the network shown on the upper left of Figure 5-16 and you have two TTL chips available in your electronics drawer: one containing four XORs and one containing four NANDs. The upper right of Figure 5-16 shows the result of using Boolean logic to convert the network to use the available gates, and the lower part of the figure shows one possible way to lay this out across the two TTL chips.

You can buy the TTL chips, plus a breadboard, switches, LEDs, 9 V battery, and resistors to drop the battery down to the 5 V used by the TTL chips (plus a resistor for each LED to prevent them exploding), and wire them up as in Figure 5-17 to implement your design.

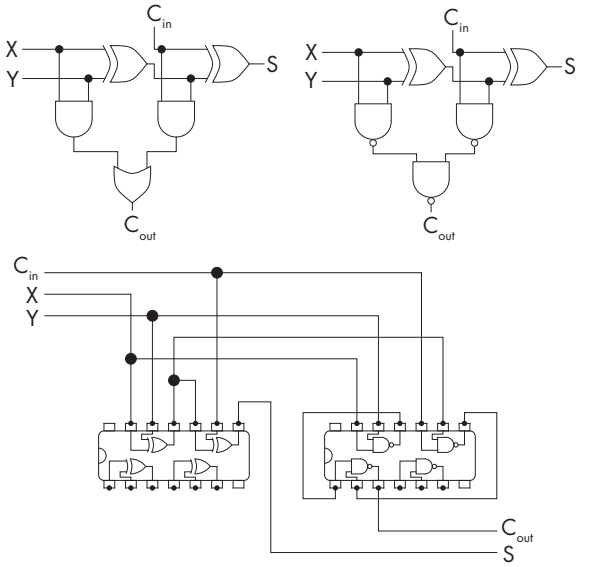


Figure 5-16: A plan for converting a network to NANDs and laying it out using TTL chips

It's possible to build a whole CPU from TTL chips in this way, and indeed this is how many early CPUs were built.

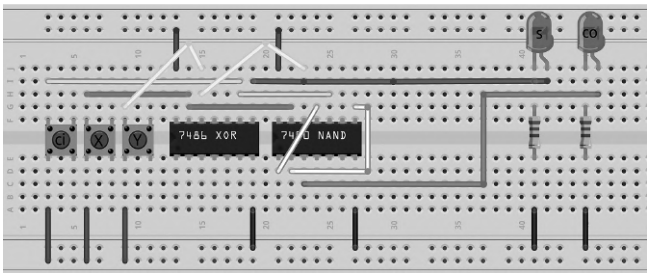


Figure 5-17: The TTL plan laid out on a breadboard (using Fritzing)

NOTE

The "bit of cleverness" required to optimize the wiring will very quickly grow in size as you try to do it for larger circuits. Similar cleverness is needed to optimize the physical layouts for the other hardware methods we'll discuss next. Designing algorithms to do this automatically and at scale is a major area of computer science and is heavily used, researched, and developed by chip companies.

Photolithography

The ASIC process described in Chapter 4 is the most heavyweight method for implementing digital logic networks, costing $5 million to make a mask set. Here, masks are prepared containing the transistor layouts needed to form the logic gates. This process gives the smallest, fastest hardware, but it's economical only at large scales to justify the setup costs.

Programmable Logic Arrays

A *programmable logic array (PLA)* is a chip with many inputs and many outputs, made with photolithography, such that every input and every input's negation are connected to a series of AND and OR gates by fuses. Figure 5-18 shows a small example of a PLA structure. The plane in the figure is stacked multiple times, with each layer sharing the same AND and OR gates. The circles are fuses.

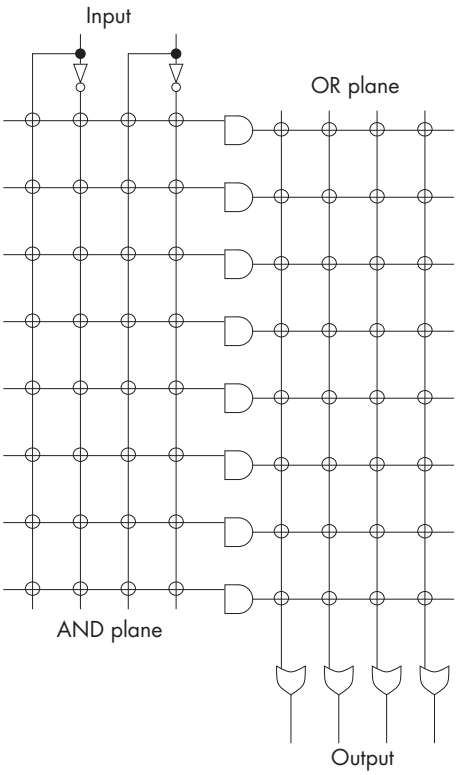


Figure 5-18: A PLA schematic showing the interconnectivity of inputs and outputs

Beginning with this structure, you can make any Boolean logic function by blowing out some subset of the fuses to effectively remove those wires. If you have a big enough PLA, you can take any digital logic design, perform some Boolean logic transformations to get it into the best form, then "burn" it into the PLA by blowing the fuses. This is nice because instead of having to custom-design your chip and spend $5 million on making a set of photolithography masks, only one set of masks is ever needed—the one to make the generic PLAs. You can then buy generic PLAs from a mass-producer and turn them into your own chips.

Field Programmable Gate Arrays

A *field programmable gate array (FPGA)* is similar to a PLA, but you can rewrite it whenever you like with new digital logic rather than only being able to burn it once. This is because rather than physically blowing fuses, FPGAs operate by electronically switching on and off connections between blocks of standard logic. Each of these blocks can be configured to act as some small simple machine. Figure 5-19 shows an example of this design.

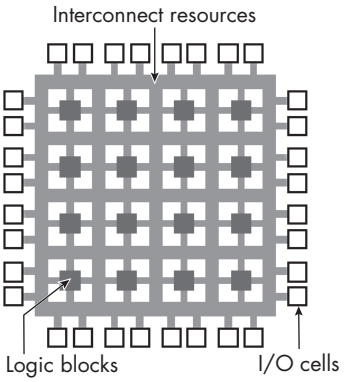


Figure 5-19: The structure of an FPGA chip, made from configurable blocks and connections between them

Boolean logic is again used to transform any initial digital logic design into a collection of such simple machines and connections between them. This is almost a software approach, with a list of the connections to enable and disable sent to some firmware memory on the FPGA board, then used to make the electronic configurations.

FPGAs are often sold on a development board with extra hardware around the FPGA chip to help connect it to a PC and program it. You can buy cheap, maker-friendly consumer FPGA boards starting at around $30. There are two main manufacturers of FPGAs: Xilinx and Altera (the former is now part of AMD; the latter is now part of Intel). Alternatively, FPGAs intended for use in production can be obtained without any supporting structure, in which case an external programmer machine is needed. FPGAs come in a variety of sizes; the larger of these chips are used for prototyping CPU designs before more expensive ASIC photolithography, while the smaller ones are intended for embedded systems.

Figure 5-20 shows a typical example layout of some digital logic on a physical FPGA surface, and the development board used to place it there.

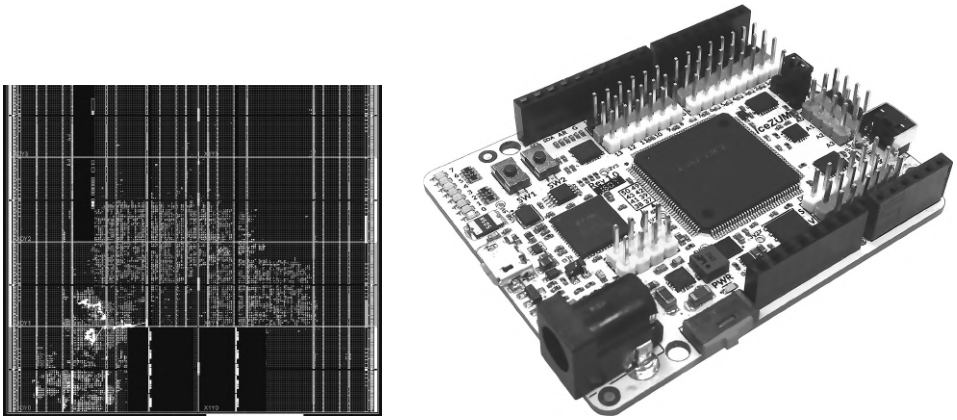


Figure 5-20: A display of the logic configuration inside an FPGA (left) and an FPGA (the large, central chip) on its development board (right)

When humans lay out digital logic manually, they tend to organize it spatially so that different regions correspond to different structures. Automated layouts, as seen inside the FPGA, tend to be visually unstructured and thus hard or impossible for humans to understand.

Summary

Logic gates are abstractions: they're one way to organize small groups of switches, such as transistors, into functional units. Human designers like to think at this level rather than at the level of switches, so they design circuits using logic gates. Each logic gate is then "compiled" into small groups of switches. (A few professional chip designers really can "see" the logic gates on the silicon. They get so used to looking at the standard patterns of transistors created by the gates that these patterns jump out in their perception. But for the rest of us, we see only the transistors.)

Unlike simple switches, logic gates have the key property that their output preserves the same representation as their input. For example, transistor-based logic gates don't produce lower voltages on their outputs than they receive as input. This means they can be combined into complex logic networks.

Claude Shannon showed us that we can use George Boole's algebra to simplify circuits of logic gates, often reducing the number of gates needed, and replacing all other types of gates with only NAND gates. This reduces the number of transistors that we need to fit onto silicon and simplifies the design.

Exercises

Universal Gates

Work out the truth tables for each of the NAND gate–based circuits in Figure 5-8, or otherwise convince yourself that they are equivalent to the standard NOT, AND, and OR gates.

Setting up LogiSim Evolution

LogiSim Evolution is a graphical digital logic simulator. It was used to create the digital logic circuit figures in this book. It can simulate circuits that you design, and later also transfer them onto real chips.

1. Install and run LogiSim Evolution from *https://github.com/logisim-evolution/logisim-evolution*.
2. Create a project and play around to create some gates and wires connecting them. Components are connected by clicking the output of one and then the input of another. Activate a component or wire by clicking it. Delete components with the DEL key and the latest wire with ESC. Press the Simulation button to run the simulation. Voltages on the wires are shown as black for 0 and red for 1. Some components can be right-clicked to edit their properties.
3. Use constant inputs and LED outputs to build and test the circuits in Figure 5-14.

Simplifying Circuits

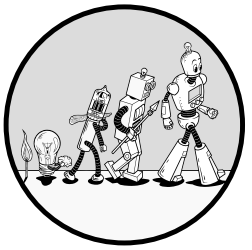
1. In LogiSim, use only NAND gates to build each of the other gate types.
2. Use model checking or proof to show why the circuits in the section "Simplifying Logic Circuits Using Boolean Logic" are all equivalent. How would you find the forms on the right of the figures from those on the left? Is there an algorithm guaranteed to give the minimal NAND form?
3. Calculate the truth table for a Boolean function such as $W(Y\bar{Z} + \bar{X}Y)$ and check it by building and simulating an equivalent circuit in LogiSim.
4. Use Boolean identities to simplify the function from the previous problem, and build a new LogiSim circuit of the simplified version. Simulate it to check that the truth table remains the same.

Further Reading

- To learn about Boole's logic (and theology) straight from the source, see George Boole, *The Laws of Thought* (1854), *https://www.gutenberg.org/ebooks/15114*.
- For arguably the greatest master's thesis of all time, see Claude Shannon, "A Symbolic Analysis of Relay and Switching Circuits" (master's thesis, MIT, 1940), *https://dspace.mit.edu/handle/1721.1/11173#files-area*.

6

SIMPLE MACHINES



In mechanical engineering, *simple machines* are a well-known set of standard designs including levers, axles, screws, and pulleys that each perform one function and can be put together to make larger machines. Analogously, computational simple machines are standard designs that are often used as subcomponents of computers. For example, the arithmetic logic unit in a modern CPU—exactly as in Babbage's Analytical Engine—is made of many such simple machines that each perform one kind of arithmetic, such as addition, multiplication, or shifting.

This chapter introduces a range of simple machines as the next architecture level above logic gates. Then, in the next chapter, we'll make use of these simple machines as components of a CPU. The simple machines we'll discuss come in two main groups: *combinatorial machines*, which can be written as Boolean expressions, and *sequential machines*, which require feedback and sequential logic, extending Boolean logic with a temporal element. Feedback and sequential logic are needed to create memory.

Combinatorial Logic

Combinatorial logic refers to those digital logic networks that can be described by regular Boolean logic, without considering the role of time. In this section, we'll see examples of several combinatorial simple machines, which we'll later rely on as we build up CPU structures.

Bitwise Logical Operations

The individual logic gates of the previous chapter act on single bits of data: they usually take one or two single-bit inputs and yield a single-bit output. It's simple to arrange multiple copies of a single gate, in parallel, thus creating an *array operator*, a simple machine that simultaneously performs the same operation on each bit of an input array to give an output array, as in Figure 6-1.

NOT:

x	0	1	0	1	0	1	1	1
out	1	0	1	0	1	0	0	0

AND:

x	0	1	0	1	0	1	1	1
y	1	1	0	0	1	0	1	0
out	0	1	0	0	0	0	1	0

OR:

x	0	1	0	1	0	1	1	1
y	1	1	0	0	1	0	1	0
out	1	1	0	1	1	1	1	1

XOR:

x	0	1	0	1	0	1	1	1
y	1	1	0	0	1	0	1	0
out	1	0	0	1	1	1	0	1

Figure 6-1: Some bitwise logical operations

Here, the input arrays x and y (or just x in the case of the NOT operation) pass through an array of identical gates, producing z as the output. These array operations are well known to low-level C programmers, as the C language includes them and assigns symbols to them. C compilers will ultimately execute these instructions using exactly this simple machine, if it's present in the target CPU.

Multi-input Logical Operations

We can create multi-input versions of AND gates from hierarchies of their two-input versions, as in the eight-input AND gate shown in Figure 6-2.

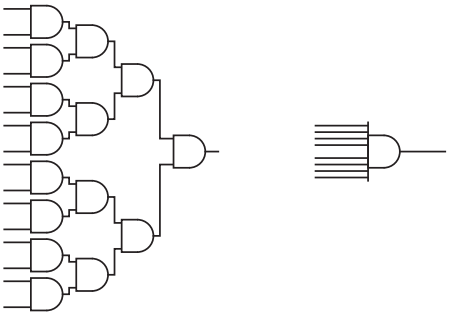


Figure 6-2: An eight-input AND gate made from two-input AND gates (left) and its symbol (right)

This structure will output 1 if and only if all of its inputs are 1. The same structure works to create multi-input OR gates, which will output 1 if one or more of its inputs are 1.

Shifters

In base 10, there's a fast trick for multiplying integers by 10: just append a zero to the end. We can also multiply by a higher natural power of 10, 10^n, by appending n zeros. Rather than thinking of it as appending a zero, think of it as shifting each digit one place to the left. Then the trick also works for multiplying non-integer numbers by powers of 10. We can similarly do easy and fast divides by powers of 10 by shifting the digits to the right. These tricks remove the need for the usual slower work of human pen-and-paper multiplication involving repeated single-digital multiplications, additions, and carries.

The same tricks work in binary for fast multiplication and division by integer powers of 2. To multiply or divide a number by 2^n, shift the number's bits n places to the left or right.

Figure 6-3 shows a simple machine that, when enabled, performs a left shift, thereby multiplying an input number by 2. The machine is enabled by setting the *S* (shift) input switch to true. If the *S* input isn't enabled, the machine outputs the original input unchanged.

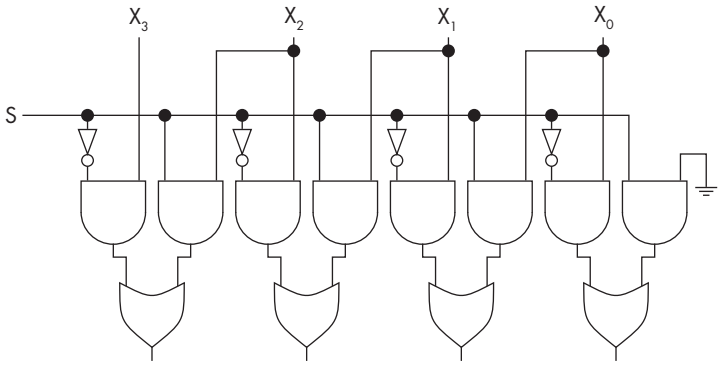


Figure 6-3: A left-shifter made from logic gates

The shifter design is based on a sub-machine consisting of two ANDs, one NOT, and one OR. Each column (digit) of the number has a copy of this sub-machine, which either allows the column's own bit to pass through unchanged, or takes the bit from the column to its right.

When you do multiplication by powers of two with an operation like `x>>2` using a high-level language like C, your CPU may contain a dedicated shifter that gets activated rather than its usual multiplication digital logic. This makes the multiplication operation go faster than one that isn't by a power of two. This is an example of how knowing architecture enables you to write fast programs. You'll often see speed-critical code designed to exploit this trick, such as games and media codecs enforcing values to be powers of two.

NOTE *Shifting by more than one place can be done in several ways. You could reuse the same shifter network several times, which would save on transistors but take longer to run. Or you could use more transistors to implement many different switches that request different kinds of shift, and implement them immediately. Deciding whether to trade off transistors for speed in this way is a common architectural dilemma.*

Decoders and Encoders

Suppose you have a positive integer x represented as an M-bit binary number. Computers often need to convert this binary representation into an alternative 1-of-N representation, which has $N = 2^M$ bits, all 0 except for a 1 in the xth bit. For example, an $M = 3$ bit input such as 101 (coding the number 5_{10}) would be converted to 00000100, which has $2^3 = 8$ bits with only the fifth one high (counting the bits from left to right, starting from 0). A simple machine called a *decoder* can perform this conversion. Figure 6-4 shows a digital logic circuit for a 3-bit decoder.

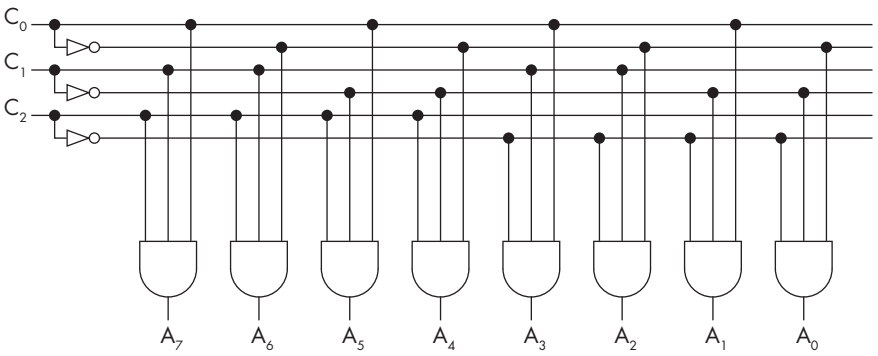


Figure 6-4: A 3-bit decoder

Each input is first copied and inverted. Then a set of AND gates are connected to either the uninverted or inverted versions of each input bit in connection patterns that model the patterns of binary number codings.

An *encoder* performs the inverse operation: it takes a 1-of-N representation as an input and transforms it into a binary number encoding.

Multiplexers and Demultiplexers

We've seen that the Analytical Engine consisted of many subcomponents that were dynamically connected and disconnected as needed to perform computations. Making and breaking these connections in the Analytical Engine was done mechanically. For example, when we wanted to do some adding, the mechanisms physically brought the gears into contact between a register and the arithmetic logic unit (ALU). Or when we loaded data from RAM, a mechanism physically connected the desired RAM location to the bus. The digital logic version of this idea is multiplexing and demultiplexing.

A *multiplexer* enables us to select which one of multiple possible sources we wish to connect to a single output. For example, we might have eight registers and want to select one of them to connect to an ALU input. Figure 6-5 shows an eight-source multiplexer. It consists of a decoder together with eight data inputs, D_0 through D_7, and additional AND and OR gates.

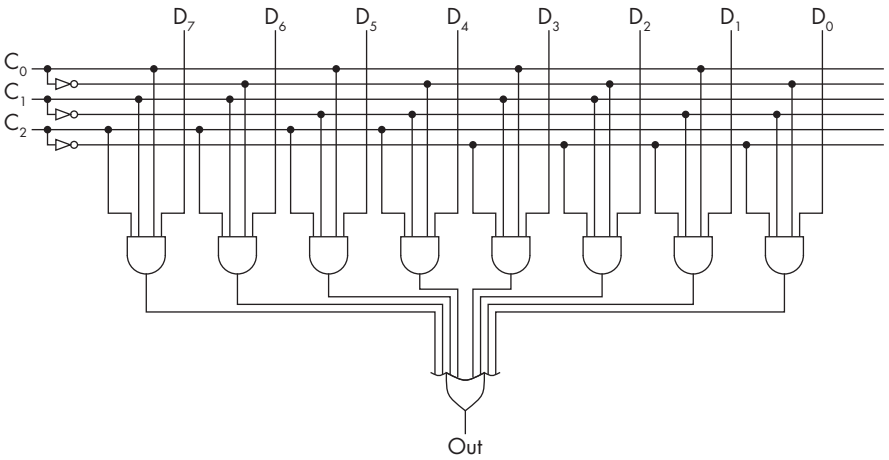


Figure 6-5: A multiplexer

If we wish to connect a particular source, such as D_3, to the output wire, we place its code, 011_2 for 3_{10}, onto the decoder inputs C_0 to C_2. The decoder sets the third line only to true, which is AND gated together with D_3 as a switch. The OR gates then copy D_3 onto the output wire, as all their other inputs are false.

A *demultiplexer* performs the opposite function to a multiplexer. It takes a single input wire and a code *n*, and sends a copy of the input signal to the *n*th of multiple output wires.

Multiplexers and demultiplexers are often used together, so we can choose which one of several possible sources to connect to which one of several possible destinations. In these cases, the shared wire is known as a *bus*.

Adders

You saw in Chapter 2 how to represent integers in binary. We can construct simple machines that use this representation to perform arithmetic operations, such as *adders* for performing addition.

Here's an example of adding two binary numbers, 001100 and 011010:

```
    0  0  1  1  0  0
+   0  1  1  0  1  0
-------------------
    1  0  0  1  1  0
    1  1
```

You can perform this addition by hand using the same algorithm as taught to children for decimal addition: starting from the rightmost column, compute the column sum by adding the digits from the input numbers for that column, writing the result underneath as the output sum for that column. If this creates a carry, for example from 1 + 1 = 10, write the lower-power column of the result (the 0 of 10) as the sum and carry the higher-power column of the result (the 1 of 10) to the next column, where it needs to be added as a third input. In the example, the first three columns (counting from the right) don't produce carries, but the fourth and fifth columns do. (The carries are shown below the final sum.)

If you look back at the truth tables for AND and XOR in Figures 5-1 and 5-4 and compare them to the work done during binary addition, you'll see that as long as there's no input carry (as is the case for the first four columns in the example), the results of XOR are identical to column-wise addition, while the results of AND are identical to the carry operation. We could thus use one XOR and one AND to form the simple machine known as a *half adder*, shown in Figure 6-6, to compute the sums for columns when there's no carry coming in.

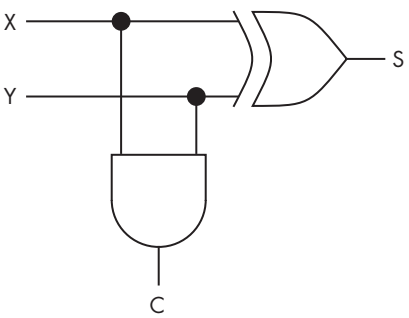


Figure 6-6: A half adder

By itself, the half adder isn't very useful, as we don't usually know if an input carry will also be present. However, if we combine two half adders together with an OR gate, as in Figure 6-7, we obtain a more useful network called a *full adder*.

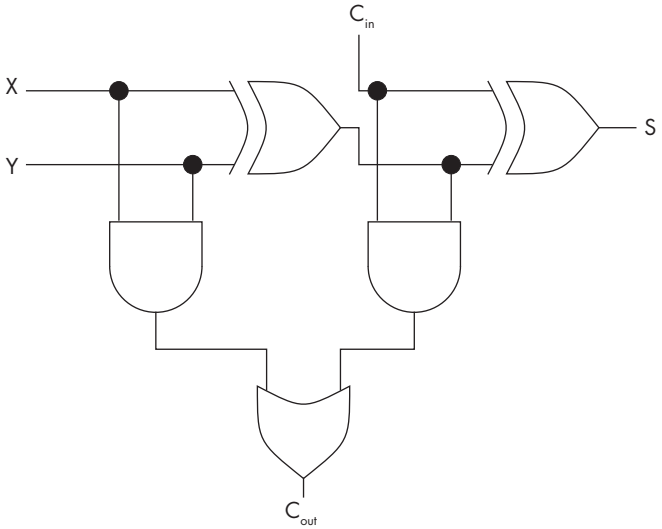


Figure 6-7: A full adder made from two half adders and an OR gate

The truth table for a full adder is shown in Table 6-1.

Table 6-1: Full Adder Truth Table

X	Y	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The full adder performs two single-bit additions in a row, the first for the main inputs (X and Y) and the second for the sum of the main inputs plus the incoming carry (C_{in}). The net result is a single-column sum, as shown here:

$$\begin{array}{ll} & C_{in} \\ + & X \\ + & Y \\ \hline C_{out} & S \end{array}$$

This is the full process needed to correctly find the binary digit sum for each column of binary addition. As well as adding the two binary digits from that column of the two input numbers, it also adds an incoming carried digit whenever it's present. The full adder's two outputs are the sum for the column (S) and the carry out for the column (C_{out}).

The full adder network is often represented by the single symbol shown in Figure 6-8.

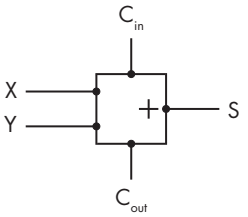


Figure 6-8: The adder symbol

A full adder performs addition of a single column, but to actually add integers together we need to add many columns. One way to do this is to create one full adder for each column and connect the carry out from each column to the carry in of the next. This is known as a *ripple-carry adder*. Figure 6-9 shows a 3-bit example that calculates Z = X + Y.

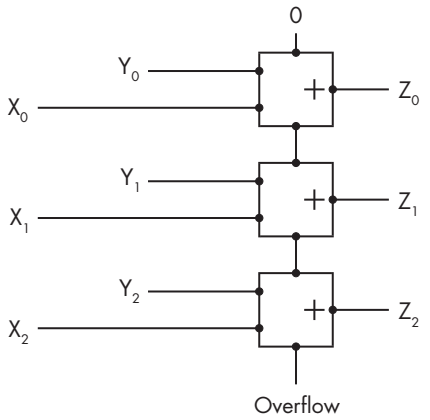


Figure 6-9: A ripple-carry adder computing the 3 bits of Z = X + Y

The subscripts say which power of 2 the column represents; for example, here X_0 is ones (as $2^0 = 1$), X_1 is twos (as $2^1 = 2$), and X_2 is fours (as $2^2 = 4$). There's an additional output from the final carry to indicate if an overflow has occurred. In some cases this would be interpreted as an error. In others it might be connected to further systems that together are able to handle larger numbers.

The adder symbol of Figure 6-8 can also be used to denote a multi-bit adder such as a ripple-carry adder, where the input and output lines are assumed to denote groups of wires rather than single wires.

RIPPLE-CARRY VS. CARRY-SAVE ADDERS

When you're taught to do addition at school, you're taught a serial adding algorithm, starting at the right side and moving across, with carry digits moving to the next step. The ripple-carry adder is a straight base 2 translation of this idea into digital logic.

Think about the efficiency of this process; assuming that both of the inputs are n digits long, we see that this method of addition will scale linearly with n as the length of digits increases the addition runs in roughly $O(n)$ time.

But addition doesn't have to be done or taught like this. Imagine that instead of teaching kids to add numbers together individually, they're taught to work from the start as a team, each performing a smaller part of the addition. How would you get the numbers added together as quickly as possible in parallel? You'd probably give each kid one pair of column digits from the addition, have them each do their addition at the same time, then have them send their carry along to the person on their left. Then they each take the carry from their right and add it into their result to update it if needed, and sometimes update their carry output and pass it again to their left, until everyone is happy. This is called a *carry-save adder*.

Estimating the number of carry steps that need to be done in this kind of parallel addition is quite a challenge. Naively, around one-quarter of initial additions will produce a carry. But then you need to think about the probability of a second or third subsequent carry step as you later receive incoming carries.

To do this properly as a probabilistic estimate, you should take into account the distribution of digits involved in the addition. Most natural quantities have a lower probability of higher-value digits (5+ in decimal; 1 in binary) than low-value digits (up to 4 in decimal; 0 in binary). This is found both in physical and pure mathematics quantities (for example, digits of Planck's constant, π, and e), though the reason why is quite complex.

Carry-save adders can do addition in $O(\log n)$ time. They're still doing the same $O(n)$ amount of total work as the ripple-carry adder, but performing more of the work in parallel, using more silicon. More silicon consumes more space and money, but in this case delivers faster performance. Again, trading silicon for time is a common architectural dilemma.

Carry-save adders are found in modern ALUs. They aren't a new idea and in fact were featured in one of the Analytical Engine designs. This is one of the main reasons the machine was never built: Babbage kept going back to improve the efficiency of the carry mechanism, to the point of obsession. Had he stuck to one design, it may have been completed.

Negators and Subtractors

If we use two's complement data representation for integers, then negating a number (that is, multiplying it by −1) can be performed by flipping its bits and then adding 1 to the result. A machine that performs this operation is called a *negator*. Figure 6-10 shows a 3-bit negator.

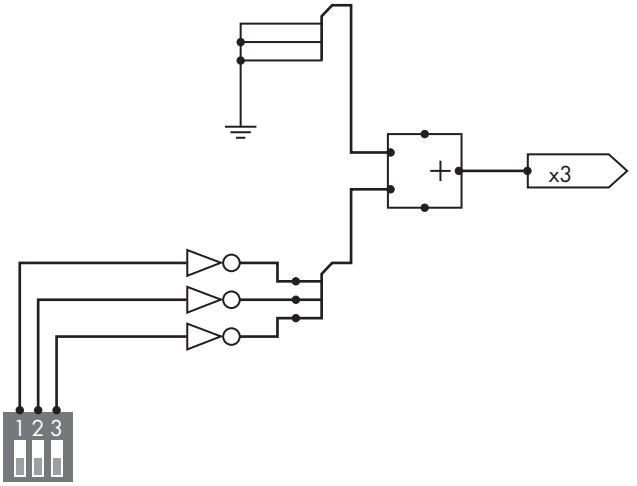


Figure 6-10: A 3-bit negator

The thick wires in this figure are standard notation for bundles of multiple individual wires, in this case bundles of three wires. (Another common notation for bundles is to draw a diagonal slash through and write the number of wires next to it.) The switches in the bottom-left specify the input number, with the least significant bit first. The number 1 to add is encoded by power and ground inputs, again with the least significant bit first. The adder symbol here indicates not just a full adder but a 3-bit adder, such as a ripple-carry adder.

Once we have a negator, we can make a *subtractor*, a machine that subtracts one number from another. Single- and multi-bit subtractors are indicated with the symbol in Figure 6-11.



Figure 6-11: The subtractor symbol

We could make a two's complement subtractor to calculate $c = a - b$ by passing b through a negator and then using an adder to add the result to a.

From Combinatorial to Sequential Logic

The combinatorial circuits we've seen so far may be viewed as computing instantly. Each circuit corresponds exactly to a Boolean logic expression, which has a definite, mathematical truth value that corresponds to the output of the circuit. This output depends only on the input values, and the input-output pairs can be listed in a truth table.

We've seen that Shannon's combinatorial logic circuits can be used to build many simple machines, like multiplexers and adders. Shannon proposed his logic gate theory in 1936, the same year as Church's and Turing's definitions of computation, and you might want to view Shannon's logic gates as an additional competing model of computation from this year, if

you're happy for a "program" to be a set of instructions for how to physically connect a bunch of logic gates, in a similar manner to programming the pre–virtual machine ENIAC.

However, Church computers need to be able to simulate any other machine (given enough memory), and we know that some other machines have *memory* for data storage. There's no concept of memory in combinatorial logic circuits because memory means storage over time, and there's no concept of time because these circuits can be viewed as acting instantly. Church computers need to have time and memory and be able to compute outputs that are functions not only of their current input but also of their state as derived from previous inputs.

We can extend Shannon's logic gates with these additional concepts if we allow logic gate networks whose outputs are fed back into their inputs. Such networks weren't allowed in Shannon's original combinatorial logic, as they would have resulted in paradoxical Boolean expressions. For example, the circuit in Figure 6-12 appears to instantiate the Boolean statement X = NOT X. This Boolean statement says that if X is true, then X is false, but if X is false then X is true. What do you think this circuit would do in practice if you connected it? Perhaps it would oscillate or explode?

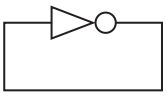


Figure 6-12: A paradoxical circuit

In computer science, feedback is often thought of as evil or paradoxical, something to be avoided: many of the theorems in logic and computability theory are about how to destroy programs, proofs, and machines by feeding their output or descriptions of themselves into their inputs. But feedback is a big idea in computer science in general, and learning to control it and use it for good has been a major part of our success and our culture. Creating memory is one such positive and controlled use of feedback.

Let's illustrate this idea using the example of a guitarist. Guitarists have a more practical worry about feedback, as their guitar strings can vibrate in sympathy with the sounds coming from their amplifiers. These vibrations, in turn, are amplified, and so on, leading to a terrible (or beautiful, depending on your musical point of view) single-frequency screeching sound. Consider exactly *when* this happens. It's possible to put the same guitar in exactly the same place in front of the amp, and yet have the system remain completely silent if there's no initial sound. The feedback emerges only if there's some sound—even a small one—to make it begin. We could thus use this guitar-amp system to store 1 bit of information. We bring the guitar next to the amp very carefully so no sound is made and the system stays silent, representing a 0. If we later want to store a 1, we stroke the strings to begin the feedback, which continues forever, representing the 1. To change it back to 0, we could turn the amp off and on again.

The circuit in Figure 6-13 is an attempt to make a digital logic version of the same idea. If we try to map it to Boolean logic, it seems less paradoxical than the circuit from Figure 6-12, appearing to instantiate the Boolean statement Q = G OR Q. (G is for *guitar*, and Q is a traditional symbol for

quiescence, or system state.) You can just about convince yourself that this is stable for G = Q = 0 or for G = Q = 1.

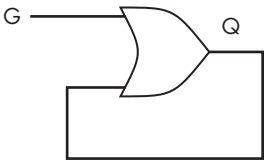


Figure 6-13: A guitar-like feedback circuit

However, this still doesn't give us the concepts of time or memory, because Boolean logic is inherently static. To fully capture these concepts, we need to go beyond Boolean logic and Shannon gates, and consider a new type of logic gate having different states at different *times*. We need to distinguish states at times using *sequential logic*, such as writing $Q_t \neq Q_{t-1}$ for states at time t and just before time t. This would be foreign to Boole and Shannon, and indeed it's an extension of their theories. It can be used to give meaning to digital logic circuits that their theories can't handle, such as mapping Figure 6-12 to X_t = NOT X_{t-1} and Figure 6-13 to Q_t = G OR Q_{t-1}. The latter is now an exact analog of the guitar feedback memory, with Q able to sustain a value of 1 copied from G even if G is later lowered to 0.

This still isn't a very useful memory, because once Q has been set high there's no way to reset it to low again. We need to add the equivalent of the amplifier power switch, A, as in Figure 6-14.

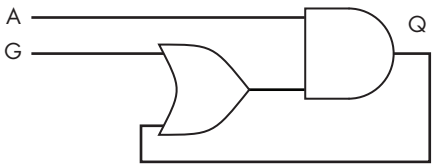


Figure 6-14: A guitar-and-amp-like feedback circuit

The *SR flip-flop* of Figure 6-15 is a variation on this idea made from two NAND gates, the most common universal gate.

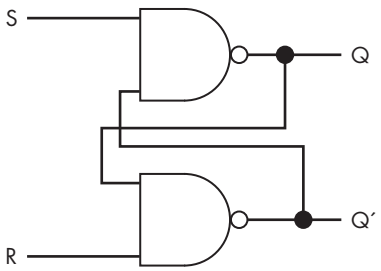


Figure 6-15: An SR flip-flop

S and R stand for *set* and *reset*. When S is high, it sets the output Q to 1. When R is high, it resets the output Q to 0. (This also has the advantage of making NOT Q available on the Q′ output as a free by-product, which is sometimes useful.)

Clocked Logic

Sequential logic behavior can be unpredictable if we don't have a clearly defined, discrete signal telling us when *t* has changed to *t* + 1. This can be done with a clock signal, traditionally called *clk*, that steadily oscillates between 0 and 1, as discussed in Chapter 4.

By tradition, the instant of the rising edge of clk is used as the instant that *t* increases by one; this is called a *tick*. We then design the temporal parts of our circuits to update their state at each tick. Copies of clk can be wired into many points across the system to make them all update simultaneously on each tick.

As with the combinatorial logic section, we'll now walk through a series of clocked logic machines.

Clocked Flip-Flops

Most sequential simple machines can be converted to clocked form by adding gates that AND their inputs with a clock signal. Figure 6-16 shows how to extend an SR flip-flop in this way.

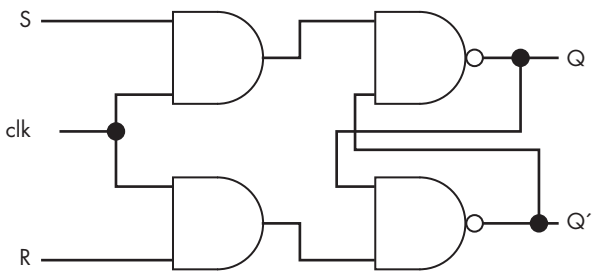


Figure 6-16: A clocked SR flip-flop

Only a single tick of high signal is needed in S or R to flip the state of the memory, which is then retained over time until a new S or R signal is received. Changes occur only during a clock tick, as the AND gates on the clock act to disable the S and R inputs at other times.

Clocked versions of simple machines are drawn with the clock input marked with a triangle, as in Figure 6-17.

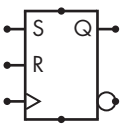


Figure 6-17: The symbol for a clocked SR flip-flop

SR is the simplest type of flip-flop to understand, and for that reason it's generally used to introduce the concept, but SR flip-flops aren't typically used in practice. This is because they have undesirable, undefined behavior in cases where both inputs are 1. The *D-type flip-flop* has a modified design that fixes this issue; it's widely used in practice. Unlike SR, it uses an inherently clock-based approach.

A D-type flip-flop (D for *data*) has only one data input and a clock input. At one point of the clock cycle, such as the rising edge, it captures the data on the D input. For the rest of the clock cycle, it outputs that value on its output Q. This stores the data for only one clock cycle—if you want to keep it for longer, you need to arrange external connections so that $D_{t+1} = Q_t$. One of many possible implementations of a D-type flip-flop is shown in Figure 6-18.

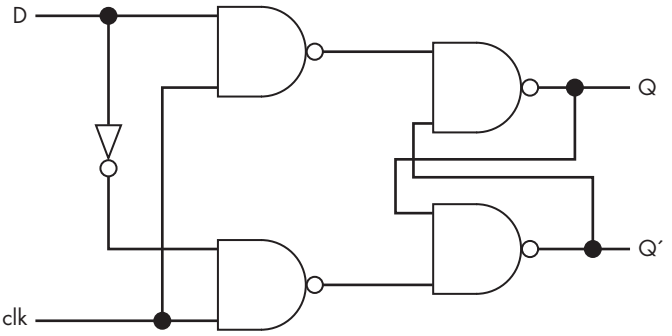


Figure 6-18: A D-type flip-flop

The standard D-type flip-flop symbol is shown in Figure 6-19.

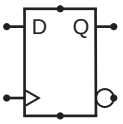


Figure 6-19: The D-type flip-flop symbol

Here, the standard triangle symbol is used for the clock input, and the negated output is shown by a circle, as used in NAND and NOR gate symbols.

Counters

A *counter* is a digital logic version of Pascal's calculator. We use a D-type flip-flop to store the value in each column, and wire its output to both its own data input (to refresh the storage) and also to the clock input of the *next* column's flip-flop as a carry. This is shown in Figure 6-20.

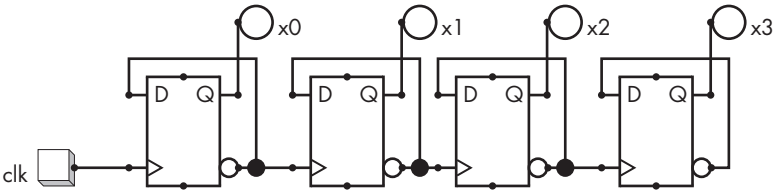


Figure 6-20: A 4-bit binary counter

If the input to the first column is a clock, then the counter will count the number of ticks that have taken place. If you take an output wire from one of the columns of the counter, you get a clock divider, which drops the clock frequency by a power of two. This is useful when you have a fast clock and want to create a slower clock from it, for example to use as a clock for slower pieces of hardware.

Alternatively, the input to the first column can be any arbitrary signal, such as a wire from a manual-controlled switch or some other event in a digital circuit, in which case the counter will count the number of these events that have taken place.

Sequencers

A *sequencer* is a device that triggers a bunch of other devices at particular times. For example, a traffic light sequencer will turn on and off the different colored lights in a particular, repeating order. A sequencer can be made from a counter and a decoder, as in Figure 6-21, which simply uses the counter's output as an input to the decoder.

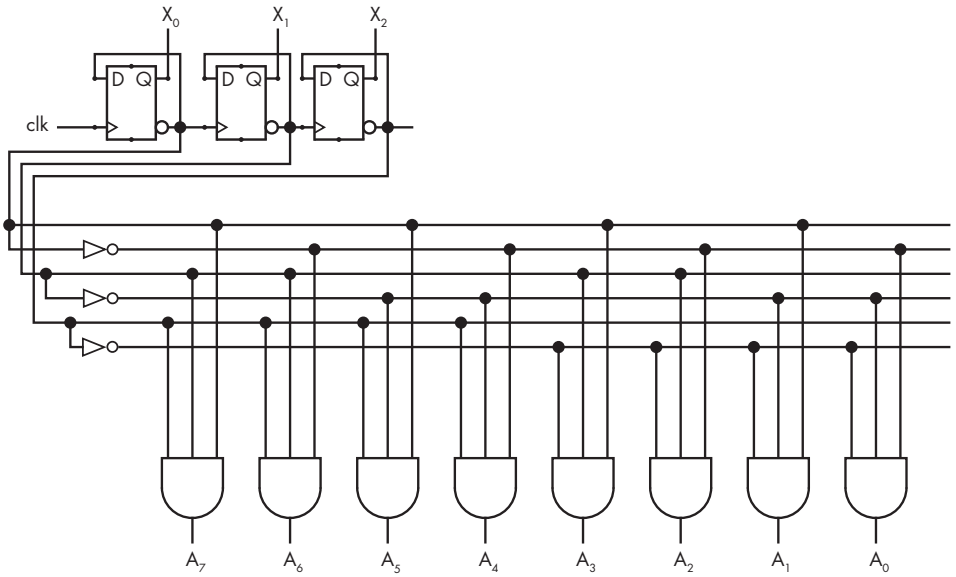


Figure 6-21: An eight-state sequencer using a 3-bit counter and decoder

Random-Access Memory

Random-access memory (RAM) is memory that consists of addresses, each containing a group of bits of data known as a *word*, and in which any address can be read and written at equal time cost. Babbage's Analytical Engine features a mechanical RAM; let's see how to build the same structure from digital logic as a simple machine.

Basic RAM has three groups of wires as its interface. First, N address wires carry a binary natural number representation specifying which of 2^N addresses is of interest. Each address stores a word of length M so, second, a group of M data wires carry copies of words to or from the specified address of the RAM. Finally, a single control wire, called *write*, carries a single bit that controls whether the specified address is to be read or written.

Figure 6-22 shows a (toy-sized) RAM with $N = 2$ and $M = 2$. The address wires are labeled A0 and A1, and the data wires D0 and D1.

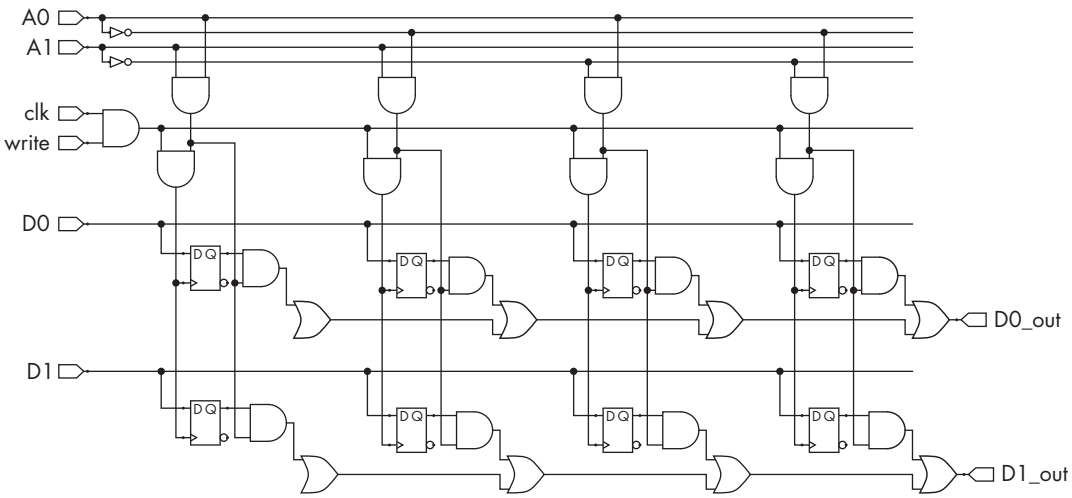


Figure 6-22: A simple RAM, with addressed words implemented as flip-flops. This toy example has a 2-bit address space of 2-bit words.

Each of the $2^2 = 4$ addresses stores a 2-bit word. Each bit of each word is stored by a D-type flip-flop. The selection of address from the address wires is performed using a decoder.

HARDWARE DESCRIPTION LANGUAGES

The tools used in this book are focused on LogiSim and simulation. Large-scale architecture is, however, usually done via a stack of text-based languages such as netlists, Verilog, and Chisel. Let's take a brief look at these formats in case you max out LogiSim and want to explore larger and more complex designs in your own projects.

Mask Files

Mask files are the very lowest level of chip description, containing the physical locations, sizes, and shapes of components such as transistors and wires. These are used to produce the masks needed for fabrication.

Netlist Files

Netlist files contain descriptions of connectivity between physical components and wires, but as abstract connectivity rather than a physical layout. You use a layout engine program to *place and route* the connections—that is, to transform a netlist file into a mask file. (This is an NP-hard problem, so layout programs use complex heuristics that were until recently closely guarded commercial secrets.)

Verilog and VHDL Files

Verilog and *VHDL* are text-based hardware description languages for designing electronic systems. In their most basic forms, they have a similar function to LogiSim, allowing you to instantiate and connect various electronic components. But instead of using a GUI, they use text files with a syntax similar to software programming languages. Unlike a language like C, however, which is imperative, Verilog and VHDL fundamentally describe static objects and relationships between them. In this sense, their structure is more like XML or a database, containing lists of facts rather than instructions to *do* things. For example, here's a Verilog module representing a full adder:

```
module FullAdder( input io_a,
                  input io_b,
                  input io_cin,
                  output io_sum,
                  output io_cout
                );

  assign io_sum = io_a ^ io_b ^ io_cin;
  assign io_cout = io_a & io_b | io_a & io_cin | io_b & io_cin;
endmodule
```

Once you write a Verilog or VHDL description, a compiler turns it into a netlist. This compilation process is called *synthesis* because the logic expressed in the source code is synthesized from gates. Software simulators also exist that can be used to test Verilog or VHDL hardware designs without actually manufacturing the hardware.

While some people still write Verilog or VHDL by hand to design digital logic, it's becoming more common to use higher-level tools such as LogiSim or Chisel (discussed next) that compile into Verilog or VHDL. Verilog also adds higher-level language constructions that enable some C-like imperative programming and get compiled to digital logic structures. LogiSim Evolution is able to export your designs as Verilog or VHDL, which enables you to compile them to netlists and use them to make real chips.

(continued)

Chisel

Chisel is a high-level hardware language that was developed for general architecture design use. Chisel describes classes of hardware with object orientation; for example, you could create a `FullAdder` class to represent the class of full adders, which could be abstracted and inherited in the usual high-level object-oriented ways:

```
class FullAdder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val cin = Input(UInt(2.W))
    val sum = Output(UInt(2.W))
    val cout = Output(UInt(2.W))
  })
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout := a_and_b | b_and_cin | a_and_cin
}
```

Chisel classes may have parameters for numbers of input and output wires, for example, to enable loops to generate *N* full adders to make a ripple adder.

Chisel is a hardware language, but it's based closely on the very high-level Scala software language. Scala, in turn, is influenced heavily by lambda calculus, functional programming, and Java; these kinds of languages are not usually associated with hardware design, so bringing them in has enabled Chisel to operate at much higher levels than the old days of having to do hardware design in Verilog. You may benefit from taking regular Scala tutorials before attempting to work with Chisel.

Summary

Logic gates can be combined into networks to perform more complex functions. Simple machines are certain well-known types of networks that tend to appear again and again in architecture. Combinatorial logic machines—including shifters, encoders, multiplexers, and adders—use Shannon's original theory, without relying on feedback or time. When feedback and clocks are also allowed, additional sequential and clocked logic simple machines can be created as well. These are able to retain data in memory over time. Flip-flops are simple machines storing 1 bit of memory. They can function as subcomponents of counters, sequencers, and RAM.

Now that we have a collection of simple machines, we can combine them in the next chapter to build a digital logic CPU.

Exercises

Building Simple Machines in LogiSim Evolution

As you work on the following exercises, keep in mind that you can create hierarchies of subcircuits in LogiSim. You might do this, for example, so that your shifter becomes available as a single component to use in higher-level networks. To create a subcircuit, click the **+** button. Then, to use the new component, go back to the main circuit and add it like any other component. Use pins for input and output inside the subcircuit if you want them to show in the external interface in the main circuit.

1. Build the left-shifter (Figure 6-3), decoder (Figure 6-4), and multiplexer (Figure 6-5) shown earlier in this chapter.
2. Design and build a right-shifter, encoder, and demultiplexer. These perform the inverse functions of the left-shifter, decoder, and multiplexer, respectively.
3. Build and test an 8-bit ripple-carry adder. Use it to perform subtraction and addition, using two's complement.
4. Build and test unclocked and clocked SR flip-flops, and a D-type flip-flop.
5. Build and test a counter, using a clock as its input.
6. Build a traffic light sequencer from a 2-bit counter and decoder. Use it to light red, amber, and green bulbs in the UK's standard sequence, which goes: (1) red (stop); (2) red and amber together (get ready to go); (3) green (go); (4) amber (get ready to stop). This is roughly how the control unit of a CPU works.

Prebuilt LogiSim Modules

LogiSim has prebuilt modules for many simple machines. For example, there's a prebuilt RAM module found under Memory in the menu, as in Figure 6-23.

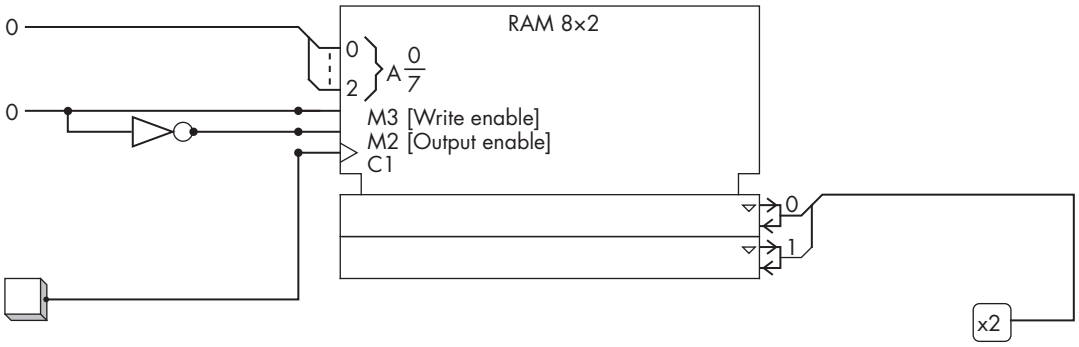


Figure 6-23: An eight-address, 2-bit word RAM

This version has two control inputs, one for write enable and one for read enable. A NOT gate is used in the figure to create both from a single control line.

1. Explore LogiSim's prebuilt modules that correspond to the machines you implemented in the previous exercises. Check that they give the same results as your own implementations.
2. Explore the RAM module shown in Figure 6-23. Use the module options to specify the RAM's word and address lengths. You can manually edit the RAM's contents with a built-in hex editor by right-clicking and then clicking Edit Contents. A splitter, found in the Wiring menu, is used to bundle and unbundle groups of wires for data and address. A probe or LEDs can be used for output; constants, DIP switches, or pins can be used for input.

Challenging

1. Design and build a natural number multiplier in LogiSim. This can be done by following the usual multiplication algorithm that you were taught at school, but in binary. You can use shifters to multiply one of the inputs by all the different powers of two, then adders to add together those powers that are present in the second number. Use AND gates to enable and disable the relevant powers. As is often the case in architecture, you can choose whether to use multiple silicon copies of the required structures, or use a single copy plus timing logic to run it many times.
2. Extend your multiplier to work with negative integers, using two's complement data representation.

More Challenging

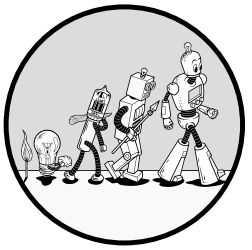
Design, build, and test an 8-bit carry-save adder in LogiSim. How much more efficient is it than a ripple-carry adder?

Further Reading

For full details on how to use LogiSim, including advanced features, see George Self, *LogiSim Evolution Lab Manual* (July 2019), *https://www.icochise.com/docs/logisim.pdf*.

7

DIGITAL CPU DESIGN



We've been building up larger and larger components of digital electronic computers, from transistors, to logic gates, to simple machines such as decoders and adders. Now it's time to put everything together at the top level to make a digital electronic CPU. At least until very recently, the CPU has been the heart of digital electronic computers.

The aim of this chapter is to overcome your fear of digital electronic CPUs. Modern CPUs are probably the second most complex device known to humanity, after the human brain. If you look at a CPU circuit under a microscope and see all the wiring without first preparing your mind, you'll likely go crazy. As with driving, you don't study a modern, state-of-the-art machine immediately; you start with a bicycle, then an old beat-up car, and then, as you get a feel for the systems, you progress to more powerful and modern machines. Likewise, we'll use one of the first and simplest digital electronic CPU systems as our example for this chapter: the Manchester Baby. Contemporary CPUs are much more complicated and may vary or break many or all of the design principles covered here, but they're still based on classical ideas. Seeing how they play out in a historical machine like the Baby will help us understand the basics.

We've already seen and understood the basic structure of a CPU from Babbage's Analytical Engine, so here we'll focus on the digital electronic implementations of the same overall design. As with Babbage, we'll play a bit fast and loose with the actual details and history of the Manchester Baby; the point is to use it to get a flavor of digital CPUs in general. The original Baby's digital electronics were built from vacuum tubes rather than transistors, and didn't necessarily use the same structures that we'd use today to re-implement its functionality. Nonetheless, the simple machines that we've studied can be used to build a modernized implementation of the Baby. We'll build such an implementation here, capable of running real programs, using LogiSim. But first, we'll learn what the Baby needs to be capable of by programming it as a user.

The Baby's Programmer Interface

Unlike the Analytical Engine, von Neumann architectures such as the Baby store their instructions and data in the same RAM space. A program is thus a list of lines that are all copied into RAM. Each line has a number, and it gets copied to the RAM address of that same number. Like the Analytical Engine, a program is made up of binary *machine code* corresponding to a series of instructions from its instruction set.

Here's the Baby's complete instruction set. We'll discuss each instruction in more detail in the following sections.

- `HLT` Halt the Baby and light the stop lamp
- `LDN` Load the negated content of the specified address
- `STO` Store the latest result to the specified address
- `SUB` Subtract the content of the specified address from the result
- `JMP` Jump to the line number stored in the given address
- `JRP` Jump forward by the number of lines stored in the given address
- `SKN` Compare the result: if less than 0, skip the next instruction

NOTE *The Baby's designers really wanted to have a regular load instruction, as in the Analytical Engine and most modern computers, to load a copy of the data stored at the address. But due to technical limitations of the time, they were forced to replace this with `LDN`, "load negated," which also inverts each of the bits from the data as they are loaded. This is a famous quirk of the Baby that gives its programming a distinctive problem-solving flavor.*

Halting

The `HLT` instruction stops the machine. This prevents execution of any further instructions and turns on a light bulb to tell the user that the work has finished, so they know when to inspect the results. The simplest Baby program is thus:

```
01: HLT
```

The line number 01 on the left of the instruction is also the RAM address where the instruction will be stored. When this program is loaded into the Baby's RAM, address 1 of RAM will contain the binary machine code for `HLT`. After loading the program, it can be run. The Baby begins execution from address 1 (not address 0, because the program counter is incremented just before each fetch) so the `HLT` will be executed, causing the Baby to halt.

Constants

Lines with `NUM` aren't true instructions but are used to place *data* at their address when the code is first loaded into RAM. For example, consider the following:

```
01: HLT
02: NUM 10
03: NUM 5
04: NUM 0
```

When this is loaded into RAM, the constants 10, 5, and 0 will be placed into addresses 2, 3, and 4, respectively, as well as the `HLT` instruction being placed into address 1.

If you actually run the program it will begin at line 1, execute the `HLT` instruction, and immediately halt. The `HLT` instruction here is very important; the CPU tries to read and execute the instructions in order, starting from address 1, but the values that we've placed into addresses 2, 3, and 4 are intended to be used as data, not instructions. The `HLT` instruction stops the CPU from getting to address 2 and beyond, preventing the data values from being treated as instructions.

This method of writing the program and data together, and storing them in the same RAM together, is the defining characteristic of von Neumann architectures. When programming von Neumann machines, it's very important that we only ever execute instruction lines, and that we don't try to execute data lines. It's a programming error—that is, a bug—to write code that ends up trying to execute data as if it were part of the program.

NOTE *Execution of data can have unpredictable and dangerous behavior. That's why it's often used as a security attack technique: if you want to break into someone else's program and have it execute your own code, you can sometimes do this by entering your code as data to that program, then somehow trick the program into executing it.*

Load and Store

The constants in the above code never enter the CPU; rather, the whole code is loaded into the RAM locations given as the line numbers by some other mechanism before the CPU is even turned on. To make use of data

from RAM in actual computations, we need to execute load and store instructions, which copy data from RAM into the CPU, and from the CPU to RAM.

For example, the following Baby program loads the (negation of the) number in address 20, then stores a copy of it into address 21:

```
01: LDN 20
02: STO 21
03: HLT
20: NUM -10
21: NUM 0
```

In this example, the number −10 is initially placed at address 20, but it gets loaded into the CPU as +10, its inverse, due to the Baby's automatic negation of loaded data. This number 10 is then stored to address 21, overwriting the 0 initially placed there. Note that the executable program is stored in addresses 01 through 03, and is terminated with `HLT`; higher addresses are used for data storage to avoid the risk of executing data.

Arithmetic

The Baby has one arithmetic instruction: subtraction. It works like Pascal's calculator: you first load one number into the CPU with a load instruction, then you give a `SUB` instruction that subtracts a second number from it. For example, the following program computes 10 − 3 = 7:

```
01: LDN 20
02: SUB 21
03: STO 22
04: LDN 22
05: HLT
20: NUM -10
21: NUM 3
22: NUM 0
```

The integers −10 and 3 are placed in addresses 20 and 21 by lines 20 and 21 when the whole program is first loaded into memory. Line 1 loads the (negated) integer from address 20 into the CPU. Line 2 subtracts the integer from address 21 from it. Line 3 stores the result in address 22, overwriting the 0 initially placed there.

Jumps

The `JMP` instruction makes program execution jump to the line whose address is one plus the number stored at the address given in the instruction. This operation is called an *indirect jump*, as opposed to a *direct jump*, which would encode the target address itself as part of the instruction, instead of, as in this case, the *location* of the target address. For example, consider the following program:

```
01: LDN 20
02: SUB 21
03: SUB 22
04: STO 20
05: JMP 23
06: HLT
20: NUM 0
21: NUM 0
22: NUM -1
23: NUM 0
```

Here, the `JMP 23` instruction at line 05 will cause a jump to line 01, because the integer 0 is stored at address 23, and 1 is the number after 0. This program loops and runs forever as a result of the `JMP` instruction.

Branches

Branching in the Baby is performed by the `SKN` instruction (skip next), which has no operand. `SKN` tests whether the current result is negative. If so, it skips the next instruction, moving to the one after it. `SKN` is usually paired with a jump (`JMP`) in the next instruction to create something similar to an `if` statement. If the result is negative, then `SKN` skips over the `JMP` instruction in the next line, and the program continues running from the line after it. If the result is positive, the jump is made and we continue running somewhere else in the program. For example, consider the following Baby program:

```
01: LDN 20
02: STO 23
03: SKN
04: JMP 21
05: LDN 22
06: SUB 23
07: STO 23
08: HLT
20: NUM -10
21: NUM 6
22: NUM 0
23: NUM 0
```

This program computes the absolute value of an integer input from address 20 and stores the result in address 23. That is, if the input is either −10 or 10, then the output will be 10; any negative sign is removed. Lines 03 and 04 are the `SKN-JMP` pair.

Assemblers

The programs we've been looking at for the Baby—and those seen previously for the Analytical Engine—are written using human-readable ASCII

symbols spelling out mnemonics such as `LDN` for "load negated," and decimal or hex numerals. Such notations are known as *assembly languages* or just *assembly*. CPUs don't understand such symbols; they require binary encodings of them, the *machine code*.

For the Analytical Engine, machine code takes the form of punches on punch cards, and the human programmer needs to manually translate their human-readable mnemonics into these binary punches before running their program. Similarly, for von Neumann machines such as the Baby, programs need to be translated into binary machine code and then placed into RAM before the CPU can execute them. The original Baby programmers had to do this by hand, using a pencil to work out the machine code, then a system of electronic switches to copy the machine code into RAM before turning on the CPU.

If you're programming a modern implementation of the Baby—or any other computer—in assembly today, you don't need to do the translation manually; there are other programs, called *assemblers*, that automate the process, translating human-readable assembly programs into machine code for you. A file of 0s and 1s corresponding to machine code is called an *executable*, as it can be executed directly by the CPU once copied into RAM. Multiple assembly languages are possible for the same target machine. For example, they could use different mnemonics for the instructions (as we have in this book compared to other implementations of the Baby).

The Baby's machine code uses one 32-bit word per instruction. The lowest 13 bits are called the *operand* and encode the numerical value used by the instruction (or are ignored for instructions that don't come with a number). The next 3 bits are known as the *opcode* and encode the type of instruction, obtained by direct translation of the assembler mnemonics, as in Table 7-1. The remaining 16 bits are ignored.

Table 7-1: The Manchester Baby Opcodes

Opcode	Mnemonic
0	`JMP`
1	`JRP`
2	`LDN`
3	`STO`
4	`SUB`
5	`SUB`
6	`SKN`
7	`HLT`

The following listing is an assembler for the Baby written in Python. If you know Python, you'll see how a dictionary is used to translate the instructions to opcodes, and how conversions between decimal, hex, and binary are used on the operands.

```
import re
f = open("TuringLongDivision.asm")
```

```
for_logisim = False #change to True to output hex for logisim RAM
dct_inst = dict()
dct_inst['JMP'] = 0
dct_inst['JRP'] = 1
dct_inst['LDN'] = 2
dct_inst['STO'] = 3
dct_inst['SUB'] = 4
dct_inst['SKN'] = 6
dct_inst['HLT'] = 7
loc = 0
if for_logisim:
  print("v2.0 raw")    #header for logisim RAM image format
def sext(num, width):
    if num < 0:
        return bin((1 << (width + 1)) + num)[3:]
    return bin(num)[2:].zfill(width)
def out(binary):
  if for_logisim:
    print(hex(int(binary,2))[2:].zfill(8))
  else:
    print(binary[::-1])  #Baby convention: show bit 0 on the left
for line in f:
    asm = re.split('\s*--\s*', line.strip())[0]
    parts = asm.split()
    thisloc = int(parts[0][:-1])
    if parts[1] == 'NUM':      #data line
        code2 = sext(int(parts[2], 10), 32)
    else:                      #instruction line
        inst2 = bin(dct_inst[parts[1]]).zfill(3)[2:]
        if len(parts) < 3:
            parts.append('0')
        operand2 = sext(int(parts[2], 10), 13)
        code2 = (inst2 + operand2).zfill(32)
    for addr in range(loc, thisloc):
      out('0'.zfill(32))  #fill in zeros where lines not given
    out(code2)
    loc = thisloc + 1
```

The following is a Baby program for long division, written by Alan Turing during his work testing and documenting the Baby at Manchester:

```
00: NUM 19   -- jump address
01: LDN 31   -- Accumulator := -A
02: STO 31   -- Store as -A
03: LDN 31   -- Accumulator := -(-A) i.e., +A
04: SUB 30   -- Subtract B*2^n ; Accumulator = A - B*2^n
05: SKN      -- Skip if (A-B*2^n) is Negative
06: JMP 0    --   otherwise go to line 20 ( A-B*2^n >= 0 )
```

```
07: LDN 31   -- Accumulator := -(-A)
08: STO 31   -- Store as +A
09: LDN 28   -- Accumulator := -Quotient
10: SUB 28   -- Accumulator := -Quotient - Quotient (up-shift)
11: STO 28   -- Store -2*Quotient as Quotient (up-shifted)
12: LDN 31   -- Accumulator := -A
13: SUB 31   -- Accumulator := -A-A (up-shift A)
14: STO 31   -- Store -2*A (up-shifted A)
15: LDN 28   -- Accumulator := -Quotient
16: STO 28   -- Store as +Quotient (restore shifted Quotient)
17: SKN      -- Skip if MSB of Quotient is 1 (at end)
18: JMP 26   --   otherwise go to line 3 (repeat)
19: HLT      -- Stop ; Quotient in line 28
20: STO 31   -- From line 6 - Store A-B*2^n as A
21: LDN 29   -- Routine to set bit d of Quotient
22: SUB 28   --   and up-shift
23: SUB 28   --   Quotient
24: STO 28   -- Store -(2*Quotient)-1 as Quotient
25: JMP 27   -- Go to line 12
26: NUM 2    -- jump address
27: NUM 11   -- jump address
28: NUM 0    -- (Answer appears here, shifted up by d bits)
29: NUM 536870912 -- 2^d where d=31-n, see line 30 for n
30: NUM 20   -- B (Divisor*2^n) (example: 5*2^2=20)
31: NUM 36   -- A (initial Dividend) (example: 36/5=7)
```

The following shows the machine code for Turing's program, as generated by the Python assembler:

```
11001000000000000000000000000000
11111000000000100000000000000000
11111000000001100000000000000000
11111000000000100000000000000000
01111000000000010000000000000000
00000000000000110000000000000000
00000000000000000000000000000000
11111000000000100000000000000000
11111000000001100000000000000000
00111000000000100000000000000000
00111000000000010000000000000000
00111000000001100000000000000000
11111000000000100000000000000000
11111000000000010000000000000000
11111000000001100000000000000000
00111000000000100000000000000000
00111000000001100000000000000000
00000000000000110000000000000000
01011000000000000000000000000000
```

```
00000000000001110000000000000000
11111000000001100000000000000000
10111000000000100000000000000000
00111000000000100000000000000000
00111000000000100000000000000000
00111000000001100000000000000000
11011000000000000000000000000000
01000000000000000000000000000000
11010000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000100
00101000000000000000000000000000
00100100000000000000000000000000
```

In this display of the binary machine code, the bit positions are printed with the zeroth bit on the left (contrary to modern convention), so the opcodes appear in the three bits just to the left of the middle of each word, with the operands to their left. This was the format used by the historical Baby, so our machine code could be entered using this convention.

The test program and the machine code each have 32 lines so that they exactly and unambiguously fill the Baby's 32 addresses of memory. The programmer needs to put something in all 32 addresses so those that aren't in use are explicitly filled with zeros. Note that the line numbers aren't encoded in the machine code; rather, they specify which address the machine code will be placed at. Also note that the data lines are translated as single 32-bit numbers, as `NUM` isn't an instruction but rather just a comment to tell the assembler that the line contains raw data. The Baby uses two's complement, so hex values such as `ffff0000` represent negative integers.

NOTE *Until the mid-1990s, many large applications and games were written by human programmers in assembly language, including* Street Fighter II *and the RISC OS operating system. Most modern programming isn't done in assembly language, but rather in a higher-level language such as C, C++, or Python. Programs written in these languages are first converted to assembly code by a compiler, before being assembled by an assembler.*

The Baby's Internal Structures

Now we'll turn to the Baby's internal structures. As we did for the Analytical Engine, we'll first introduce the subcomponents within the digital CPU, and then consider the dynamics of how they behave and interact to execute programs. The main digital CPU substructures are exactly the same as for the Analytical Engine: registers, an arithmetic logic unit (ALU), and a control unit (CU). They have the same functions as in the Analytical Engine but are built from the digital logic simple machines we studied in the previous chapter, rather than from Babbage's mechanical simple machines.

We won't follow the exact implementation of the original Manchester Baby here; rather, I'll show general digital logic implementations that *could*

be used to implement the Baby's programmer interface in a more modern style. These implementations are built from simple machines of digital logic, which in turn are built from logic gates that could be implemented equally well using modern transistors or the Baby's original vacuum tubes.

Registers

Registers are fast word-length memory, usually made today as arrays of D-type flip-flops, which live inside the CPU and are readable and writable by the CU and ALU. Most CPUs include several types of register used for different purposes.

The sizes of the registers in a CPU are usually taken to define the CPU's word length; for example, an 8-bit machine uses 8-bit words that are stored in 8-bit registers, and a 32-bit machine uses 32-bit words that are stored in 32-bit registers. The Baby is a 32-bit machine in this sense. The words use the data representations seen in Chapter 2, which require an array of bits to store numbers, text, and other data.

Like the individual flip-flops that compose them, registers must be timed to enable correct synchronization of reads and writes. An update signal can be sent to the clock inputs of all the flip-flops making up the register. Usually writes to the register are performed on the rising edge of this signal. Each register also continually outputs its latest stored value for reading as a set of parallel wires, regardless of the updates. The register structure is shown in Figure 7-1. The write is triggered when you press the button.

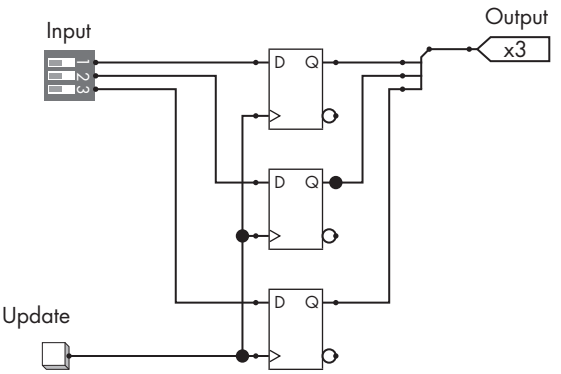


Figure 7-1: A 3-bit register made from flip-flops

Registers can also be notated using a single symbol suggesting a pile of flip-flops, as in Figure 7-2.

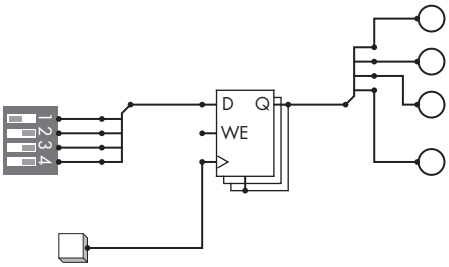


Figure 7-2: A 4-bit register as a single symbol, connected to input switches, write update button, and output LEDs

Here, the D input and Q output are each groups of wires, shown as thick lines, then split into single wires.

User Registers

User registers are usually the only registers that are visible to the assembly language programmer, who can give instructions to act on their contents.

An *accumulator* is a user register that functions both as an input and to store the results of calculations in the same place. As we've seen, Pascal's calculator is one big accumulator because it stores both one of the inputs to an addition and also its result, destroying the original representation of the input in the process. Your desktop calculator is also an accumulator: it only ever stores one number, the current result seen on the screen, which you can add to or multiply by, and which is updated to store the result. For example, if you enter 2, this is stored in the accumulator. If you then enter +3 the accumulator stores and shows the result 5. The original value 2 and the operation +3 are lost, and only the accumulated result is available.

Accumulator architectures are those that have only a single user register that acts as an accumulator. The Baby uses this simple style of architecture. This forces all computation to be done in the accumulator style because there are no other registers in which to keep inputs separate from outputs. By contrast, more complex CPUs may have other user registers, in addition to or instead of an accumulator.

Internal Registers

In addition to the user registers, most CPUs require further registers for their own internal operations. These *internal registers* may be invisible to the user, so you can't write assembly programs that access or modify them. Rather, they're used to make the CPU itself work, and to enable it to read and execute user programs. Let's look at the two most important internal registers.

A CPU needs to keep track of where it currently is in the execution of its program. In the Analytical Engine, the current line of the program was stored using the mechanical state of the program card reader. Like typewriter paper, the program was mechanically fast-forwarded and rewound so that the current line was positioned on the reader. In an electronic CPU, there is no mechanically moving state, so we must instead keep track of where we are in the program by storing the current line number in a register, called the *program counter* (*PC* in the listings in this chapter). As we've seen, von Neumann architectures—such as the Baby and most modern computers—store the program in main memory, along with other data, so these "line numbers" are actually memory addresses that store the instructions of the program.

The *instruction register (IR)* stores a copy of the current instruction, copied in from its address (as kept in the program counter) in memory.

Arithmetic Logic Unit

Just like the Analytical Engine's ALU, a digital logic–based ALU consists of a collection of simple machines, each performing one kind of arithmetic operation. Due to a quirk in its hardware, the original Baby had only a subtractor, but here we'll build a more general and powerful ALU that also includes addition, multiplication, and division. Figure 7-3 shows a 32-bit ALU with these operations.

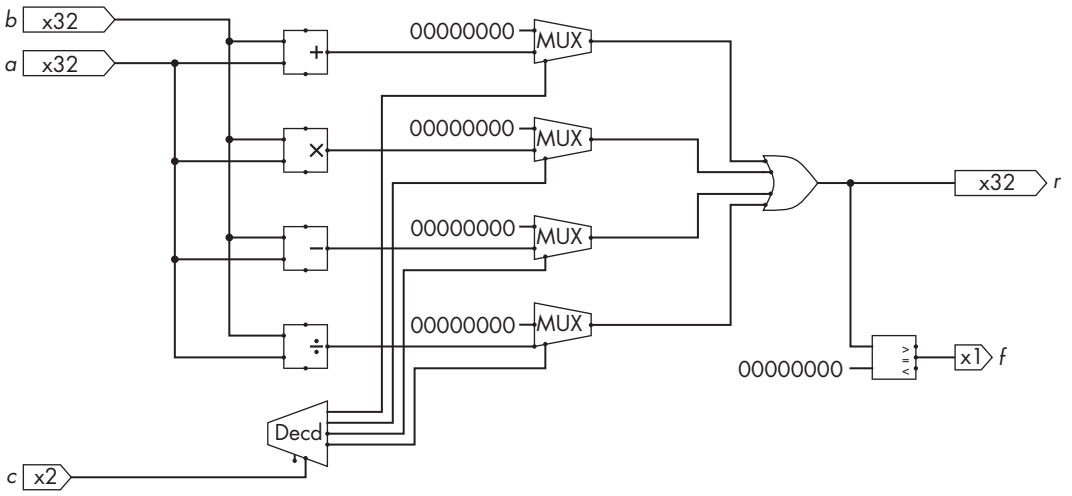


Figure 7-3: A 32-bit, four-operation ALU

Here there are two data inputs, *a* and *b*, each containing a 32-bit, two's complement integer. They're both sent to all four simple arithmetic machines, a subtractor, adder, multiplier, and divider, each of which

calculates an output. Just one of these outputs is then selected and passed to the ALU's output, *r*. To choose which arithmetic operation you want to do, place its 2-bit code on the *c* input. The decoder then activates one of the four 32-bit multiplexers, enabling the desired operation's output to be passed to and through the 32-bit OR gate array. A copy of the final output from the OR gate array is passed into a *comparator*, which tests if the number is zero and outputs a single status flag with this Boolean.

The comparator can be implemented simply by NOR gating together all of the bits in the number. More advanced ALUs often test for other interesting properties of the result, such as being positive or negative, having created an overflow (which can be seen on the carry out lines of the simple machines), or division by zero; they then output a set of flags rather than just this zero test.

Note that we could build this ALU with less silicon by using a single multiplexer with 2-bit input to select directly between the four arithmetic machines. Duplication of digital logic could be reduced by sharing structures between operations—for example, using two's complement enables adders to be reused as subtractors. You might also be concerned about the waste of energy from running all of the arithmetic options on each set of inputs but throwing all but one result away. You could find ways to redesign the network to reduce this energy usage. However, I've chosen the present structure for educational reasons, as it will help you to more easily understand CUs in the next section.

Control Unit

Digital logic CUs implement the same concept as Babbage's timing barrel, acting like a musical conductor to trigger all of the other CPU components at the right times. There are many ways to do this, so CUs vary far more than registers and ALUs. They're usually considered to be the hardest and most central part of CPU design. We'll choose a particular style here for ease of understanding rather than for computational or energy efficiency.

This style is based on two structures: first, a counter that, like Babbage's barrel, rotates regularly, and whose value is used to time the required events; second, a switching mechanism that determines the type of event to be triggered and makes temporary connections between components—such as registers, the ALU, and RAM—in response. In Babbage's machine, these connections were made and broken using mechanical levers. For our digital logic version, we'll use multiplexers, as we did in the ALU. These multiplexers have two data inputs, each of word length—32 bits for the Baby. One is hardwired to zeros and the other is from the temporary input source. They have a single-bit switch that switches between relaying the temporary input onto the output and sending all zeros to the output. Figure 7-4 shows how this works.

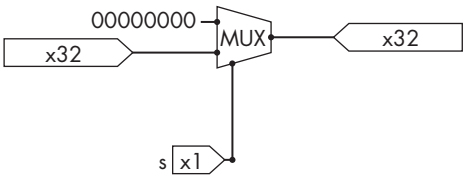


Figure 7-4: A multiplexer used to enable or disable a connection of 32 wires

As in the ALU, where multiple sources can potentially connect to a destination, they each get their own multiplexer. Then an OR array combines the multiplexer outputs, allowing the non-zero output to pass through.

We'll create a sequence of temporary connections between components. Some of these connections can be triggered simply by the time shown on the counter. This can be done with a decoder, taking the time as input and activating a particular trigger wire as output. Other connections need to be triggered by a combination of a time and some other value, such as the identity of the current instruction. These can be done by AND gating the appropriate trigger wire to signals representing the other required conditions. Figure 7-5 shows an example of this structure.

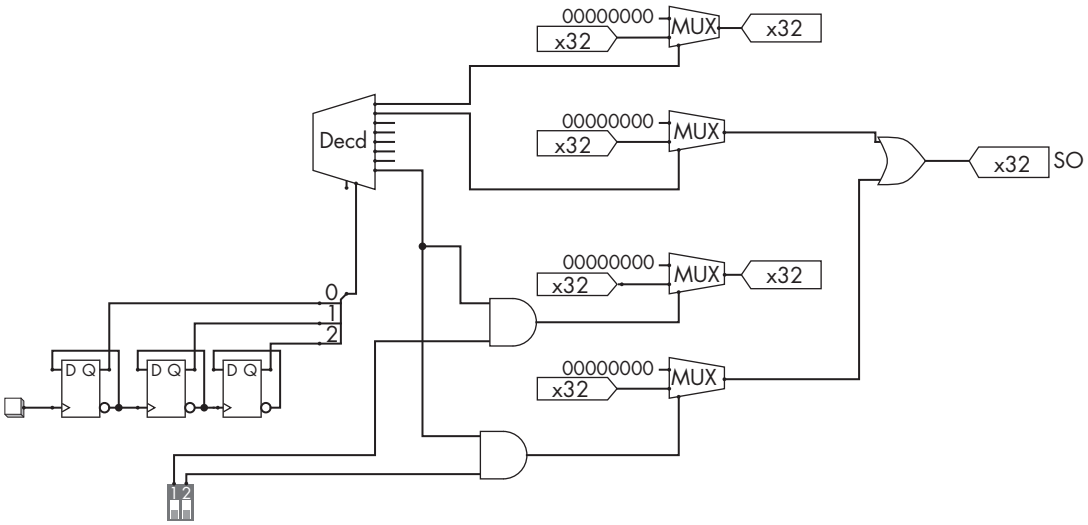


Figure 7-5: A minimal CU based on a 3-bit counter, a decoder, and multiplexer switches

The 3-bit counter and decoder on the left form the sequencer. Using 3 bits gives us $2^3 = 8$ times, from 0 to 7, looping back to 0 after each 7. The figure shows triggers only at ticks 0, 1, and 7. The triggers at ticks 0 and 1 (the upper two outputs from the decoder) depend only on the time and make connections between 32-bit wire groups. At tick 7, there are two possible triggers, which depend on conditions cond1 and cond2 being met, respectively, represented by the two switches near the bottom-left of the figure. Note that either, neither, or both of these conditions (and thus

triggers) could be active at this time. The OR symbol here represents an array of 32 OR gates. It allows two different inputs to be connected to the same shared output (SO) on different triggers. (Triggers from times 2 through 6 inclusive are omitted in this figure, but you can imagine those wires from the decoder connecting to similar triggers.)

Let's introduce a little extra notation at this point to help make our diagrams more readable. Figure 7-6 shows exactly the same minimal CU, but introduces *tunnels*, which are named points (t0 to t7 and c0 to c1) taking the place of wires. All tunnels having the same name are assumed to be connected to one another. For example, the t0 tunnel coming out of the decoder connects to the t0 tunnel going into the multiplexer near the top right of the figure.

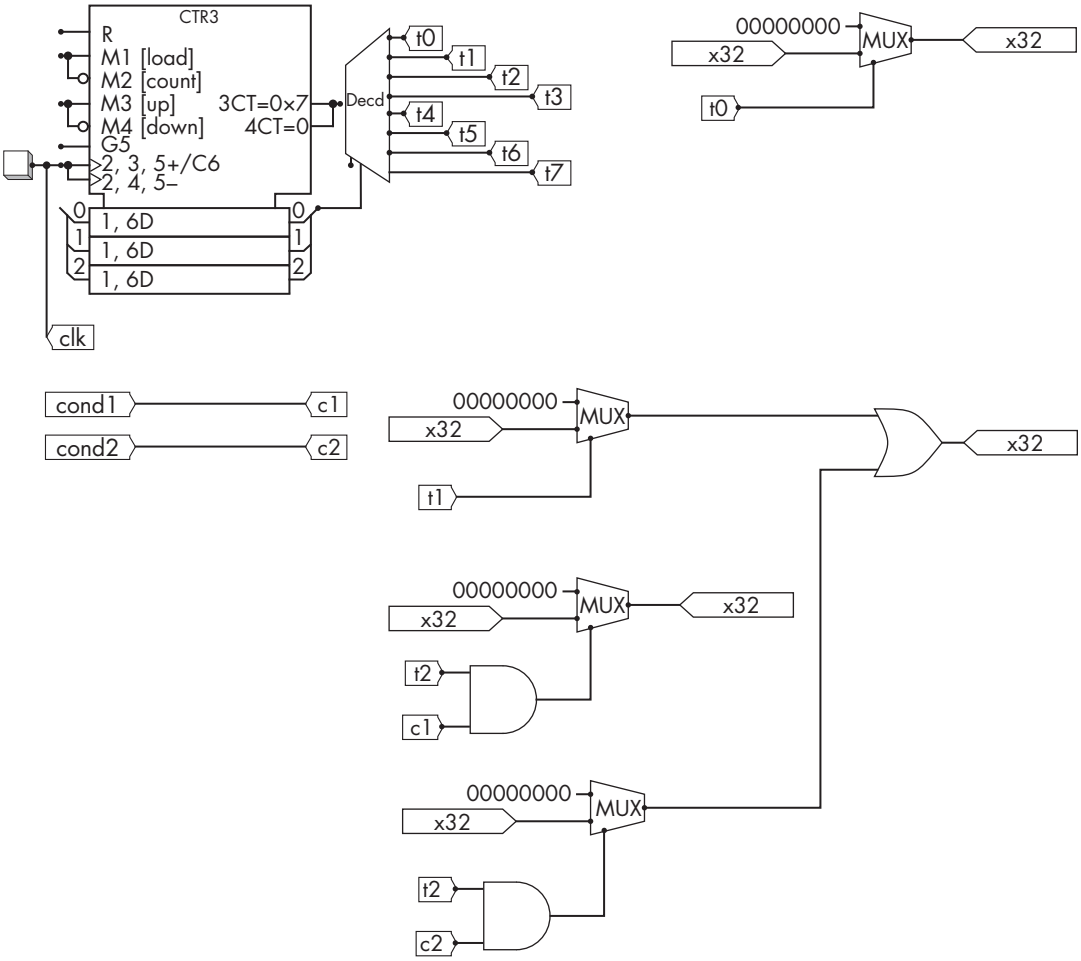


Figure 7-6: The same minimal CU as in Figure 7-5, redrawn using tunnel notation

This tunnel notation avoids the need to draw the rat's nest of wires formed as the CU sends its triggers all over the rest of the CPU. We've also encapsulated the 3-bit counter into a single block, CTR3, as provided off-the-shelf by LogiSim. (This block has some additional inputs and features that we don't use here.)

Putting It All Together

Now that we've seen each of the basic ingredients of a Baby implementation, let's put them all together—along with suitably timed CU dynamics—to build a complete, functioning Baby. We'll do this by considering the three main stages of operation—fetch, decode, and execution—in turn, just as we did when we discussed the Analytical Engine in Chapter 3.

Fetch

The aim of the fetch stage is to bring a copy of the next instruction from RAM into the IR in the CPU. Fetching assumes that the address of the next instruction is already in the program counter. When the CPU is first turned on, the program counter—like all registers—is initialized to 0, but is immediately incremented to 1, so the first instruction must be stored at address 1 and will be fetched.

To perform a fetch, the program counter is temporarily connected to the address lines of RAM, on tick 1. The data out lines of RAM can be permanently connected to the IR data in, but the IR takes only a copy of the word from these lines when write-enabled and clocked at tick 2. The network in Figure 7-7 is set to perform fetch for the Baby's 32×32 RAM ($32 = 2^5$ addresses, each containing one 32-bit word) by making these connections on ticks 1 and 2 of its eight-count control cycle, and breaking them on the other steps. In our Baby, the program counter is a 5-bit register and the IR is a 32-bit register.

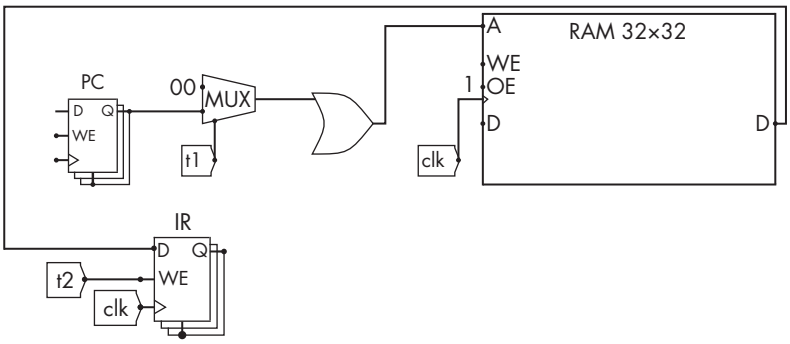


Figure 7-7: A fetch, triggered on ticks 1 and 2

We can write the fetch sequence as:

```
t1: RAM_A <- PC
t2: IR <- RAM_Dout
```

This style of notation is a form of *register transfer language (RTL)*. The symbols before the colon on each line are the triggers, which in this case are ticks 1 and 2. The arrows denote that a temporary connection from a source to a destination is made only when the trigger is active. The arrows thus correspond to the multiplexers used in our implementation style, with the triggers corresponding to the switching inputs of these multiplexers.

NOTE *RTL is not assembly language or machine code. It's a lower-level description of how the CPU works, whose function is ultimately to execute the machine code program written by the user and stored in RAM.*

Decode

We now have a copy of the next instruction sitting in the IR. It consists of a word of machine code, with some bits specifying the opcode and the other bits possibly containing zero, one, or more operands. In the Baby, bits 13 to 15 are the opcode, bits 0 to 12 are a single operand for some instructions, and the remaining 16 bits aren't used. This encoding now needs to be decoded. We need to split up the opcode and operand, then convert the opcode into an activation signal. Figure 7-8 shows our implementation.

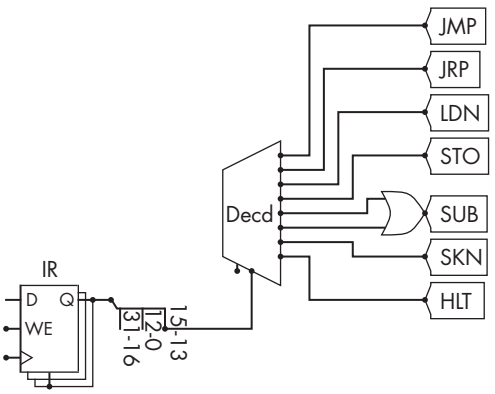


Figure 7-8: Decoding, triggered at tick 2

The IR output is first split into three sets of wires, for the first 13, next 3, and remaining 16 bits. The middle 3 bits, containing the opcode, are connected to a 3-bit decoder. The decoder activates one of its $2^3 = 8$ output lines, which is connected to a tunnel and will be used as a condition to trigger multiplexers in other steps. These tunnels are named as their corresponding assembly mnemonics. The 13 operand bits of the IR are further split into the 5 lower-order bits, which are used for address selection and will

have wires connected to them later, and the 8 higher-order bits, which have nothing to address and are ignored.

There's no sequential logic used here, so decoding happens roughly instantaneously once the IR content is updated on tick 2.

Execute

Unlike fetching and decoding, what happens during the execute stage depends on the instruction that has been fetched and decoded. Different instructions specify the activation of different structures that do different things: load, store, arithmetic, and program flow control. We'll look at how to execute each of these possible actions in turn.

Load

To execute a load, we temporarily connect the operand to the RAM's address input at tick 3, and then temporarily connect the RAM's data output to the accumulator (Acc) at tick 4. These tick numbers are chosen to take place after the previous fetch and decode. We can write this in RTL style as:

```
t3, LDN: RAM_A <- IR[operand]
t4, LDN: Acc <- -RAM_Dout
```

Note that the triggers before the colon now include both a tick and the `LDN` condition. The square brackets in `IR[operand]` indicate that only the operand bits of IR are to be used, rather than the entire register content.

Figure 7-9 shows the digital logic for our Baby implementation's load. (As the Baby's load operation also negates the loaded values, we pass the RAM data out through a negator on its way to the accumulator. This would not usually be done on a modern machine.)

In an accumulator architecture such as the Baby, the load always places the data from RAM into the accumulator register. In more complex architectures with more user registers, an additional operand is needed to specify the target, and more digital logic is needed to connect the right register to the data line.

Store

Storing a value from the CPU into memory is similar but opposite to loading. In the Baby, the value to be stored is always taken from the accumulator.

At tick 3, we temporarily connect the `STO` instruction's operand (the address to store at) to the RAM's address lines. The accumulator output can be permanently connected to the RAM data input, but only write-enabled at tick 3. The RTL for this is:

```
t3, STO: RAM_A <- IR[operand]
t3, STO: RAM_Din <- Acc
```

Figure 7-10 shows the digital logic implementing this for our Baby.

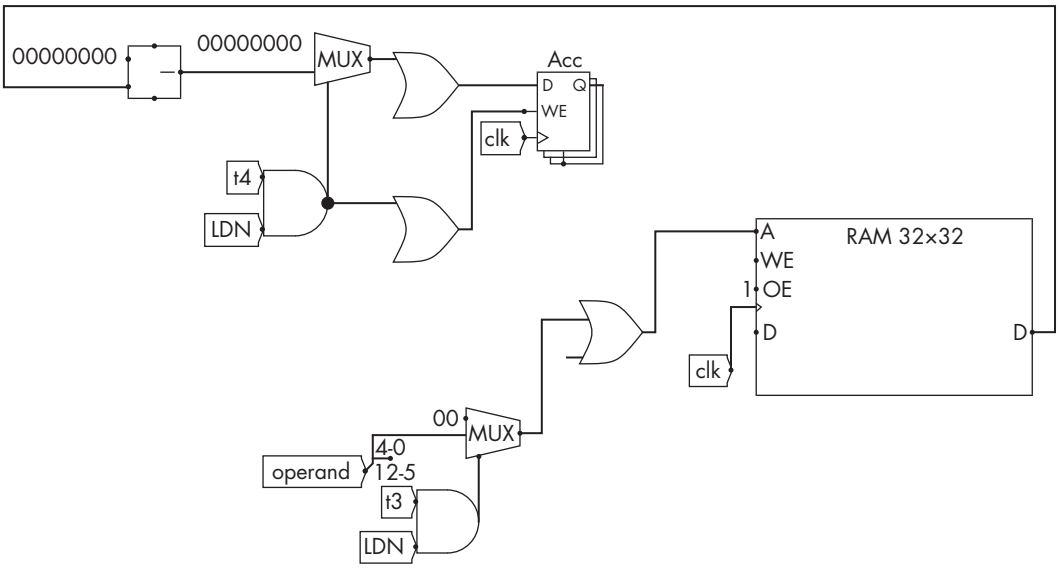


Figure 7-9: Executing a load, triggered at ticks 3 and 4

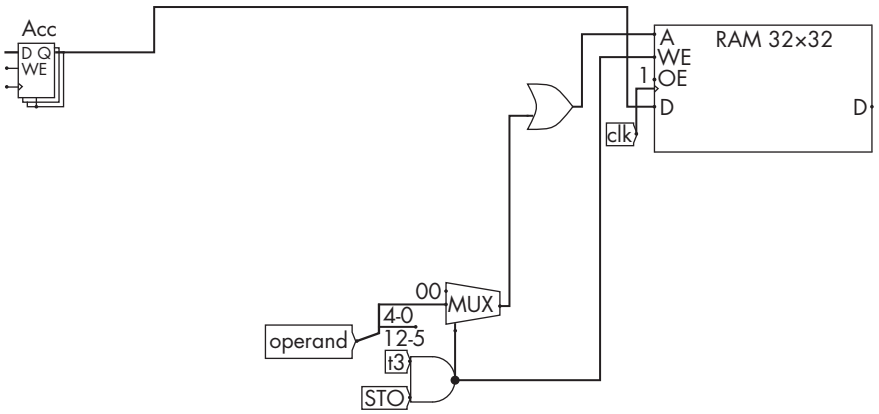


Figure 7-10: Executing a store, triggered at tick 3

In architectures with more user registers, another operand can be used to specify which register's contents are to be stored, and more switching logic is then needed to connect the right register to the data line.

Arithmetic

To execute an ALU operation, the CU makes temporary connections to the ALU's inputs from CPU registers, and creates and sends an ALU command to the ALU's command inputs. The ALU output is then temporarily connected to a destination register.

The Baby's ALU is especially simple, as it contains only a subtractor. The SUB instruction triggers a read from RAM, similar to a load instruction,

but the RAM data is sent to the subtractor rather than to the accumulator. The subtractor takes its other input from the accumulator and writes its output back to the accumulator.

Figure 7-11 shows our Baby ALU implementation. The RAM read is triggered on tick 3, and the accumulator update on tick 4. The subtractor is on the far left of the figure.

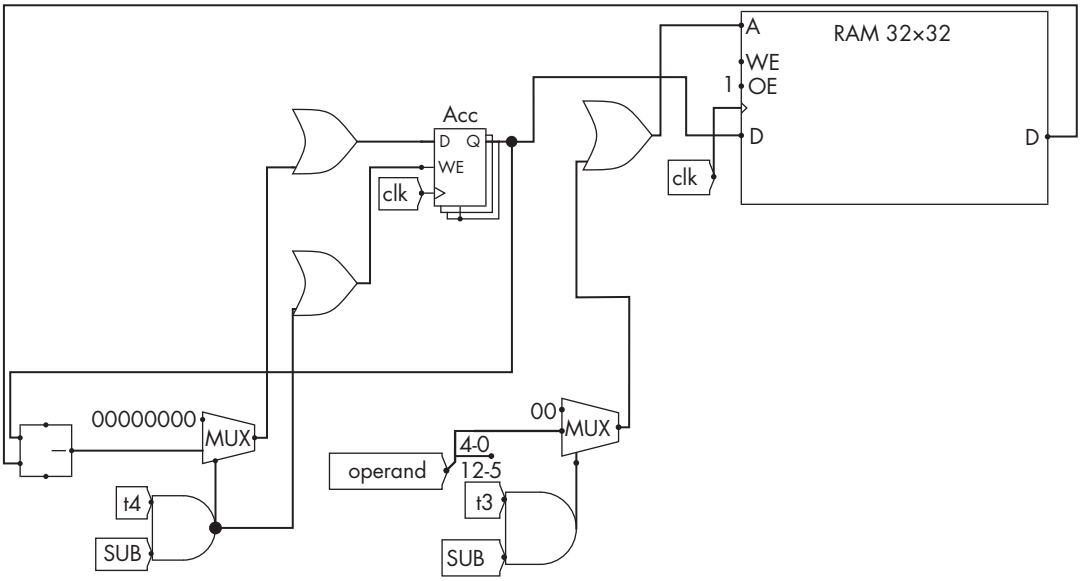


Figure 7-11: Executing an ALU operation, triggered at ticks 3 and 4

This can also be described in RTL as:

```
t3, SUB: RAM_A <- IR[operand]
t4, SUB: Acc <- Acc - RAM_Dout
```

More complex architectures having more arithmetic operations than just subtraction would package them up into a single ALU structure, with select lines to specify which to activate, as you saw in Figure 7-3. The decoder would then need to recognize multiple different arithmetic opcodes, and route each one through some logic to activate the corresponding selection.

Flow Control

At the start of each instruction, the Baby moves to the next address (line) of the program. This can be done by incrementing (adding 1 to) the program counter at tick 0.

If the current instruction is a flow control instruction—that is, a jump or branch—then its execution step also needs to update the program counter to get it ready for the next instruction.

Modern (direct) jump instructions contain the line number to jump to in their operand, so they can be implemented by copying the operand directly into the program counter. As we've seen, however, the Baby uses an indirect jump instruction, `JMP`, in which the operand contains the *address* that in turn contains the actual jump target. To implement this indirect jump we thus first attach the operand to the RAM address lines at tick 4, then attach the RAM data lines to the program counter at tick 5.

The Baby also has a relative jump, `JRP`, which works similarly to `JMP` except that the address in the operand contains a number of lines to advance the program counter, rather than an absolute address.

For the branch instruction, `SKN`, we check its condition and behave as normal if it's false, or increment the program counter an extra time if it's true, to skip over one line of code. (Usually the skipped line will be chosen by the programmer to be a jump to another part of the code.) To implement this, we send the output of the accumulator to a comparator that tests if it's less than zero. The truth or falsehood of this condition is then used (thanks to Boole) as an integer 0 or 1, which is added to the program counter at tick 5 when the branch instruction is active.

If the current instruction isn't a control flow instruction (that is, if it's `SUB`, `LDN`, or `STO`), then no further changes are made to the program counter. This is implemented simply by wiring the program counter's output directly to its input at tick 5.

Figure 7-12 shows our Baby implementation of flow control. In RTL notation, this corresponds to:

```
t0: PC <- PC + 1
t4, JMP: RAM_A <- IR[operand]
t4, JRP: RAM_A <- IR[operand]
t5, SKN, (Acc<0): PC <- PC + 1
t5, JMP: PC <- RAM_Dout
t5, JRP: PC <- PC + RAM_Dout
```

Once the program counter has been updated by any of the means described here, the fetch-decode-execute cycle is complete, and everything is set up for the next cycle to begin.

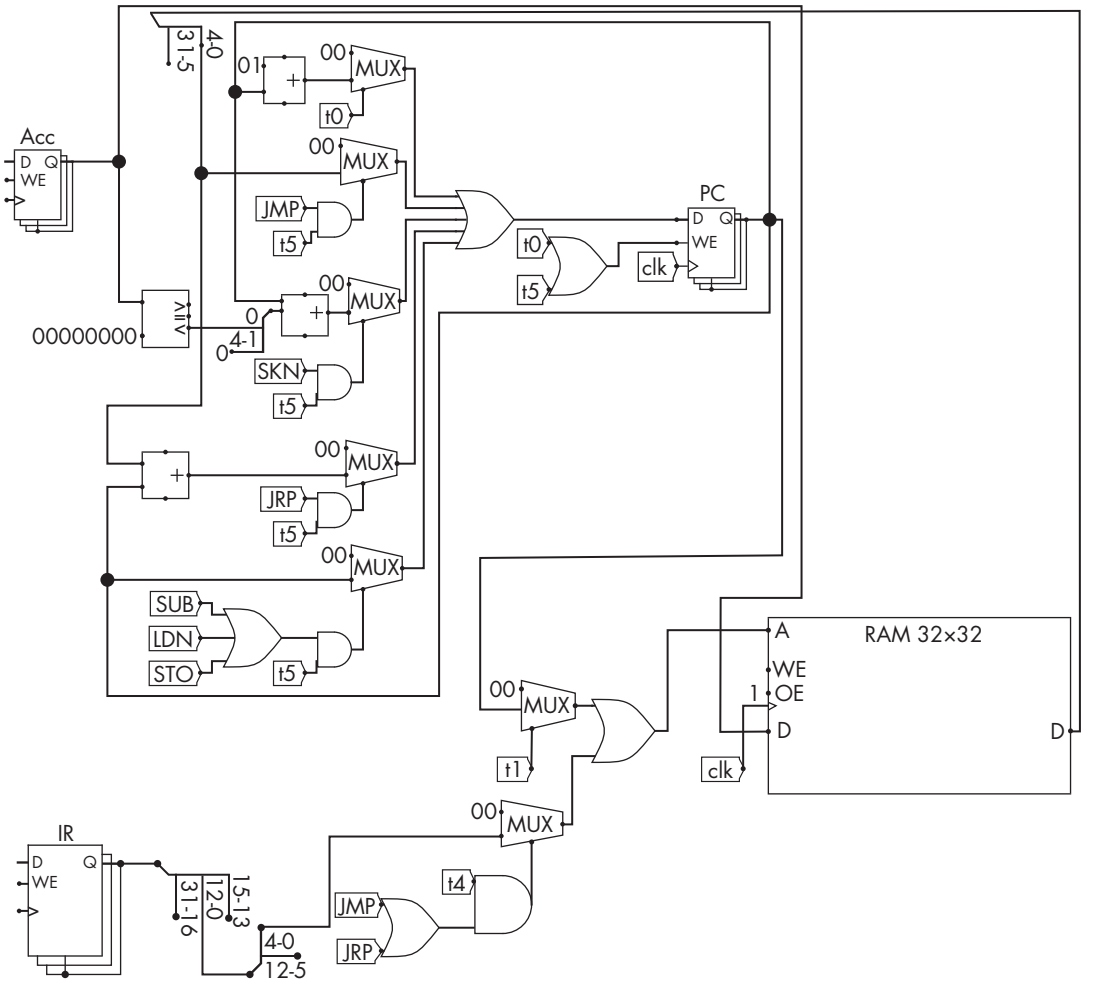


Figure 7-12: Program flow control, triggered at ticks 0, 4, and 5

Complete Baby Implementation

Figure 7-13 shows our complete, working Baby CPU, with all of the above systems shown together. In the lower-left it adds a register and lamp that activate when the halt instruction is executed, preventing any further execution. If you get tired of manually triggering the clock, it also adds a switch connecting the clock signal to an oscillator.

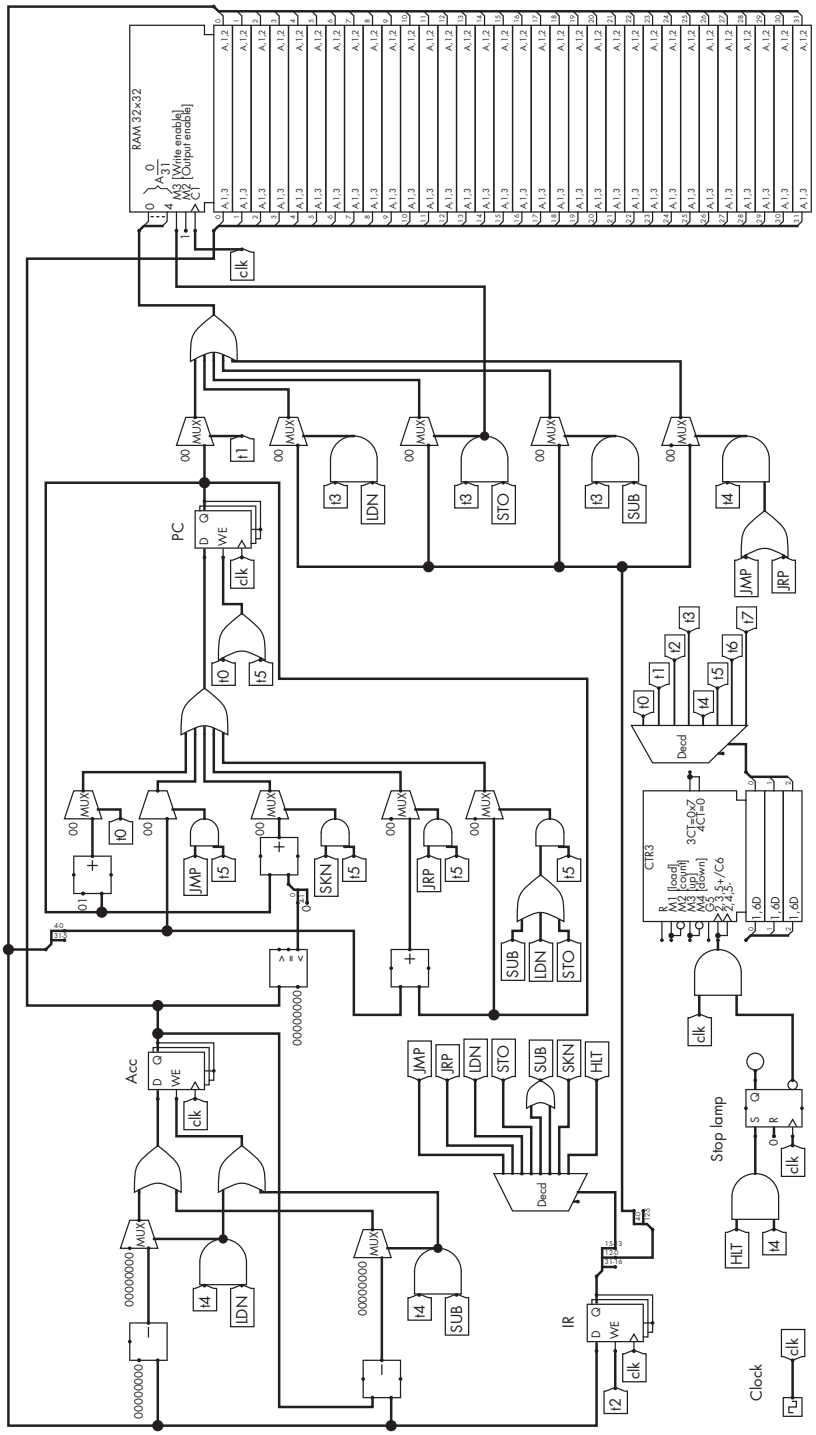


Figure 7-13: A complete, working Baby implementation, including sequencer, fetch, decode, execute, and control flow logic. The RAM contents are now shown in full.

In RTL notation, our complete, working Baby can be written as:

```
t0: PC <- PC + 1
t1: RAM_A <- PC
t2: IR <- RAM_Dout
t3, LDN: RAM_A <- IR[operand]
t3, STO: RAM_A <- IR[operand]
t3, STO: RAM_Din  <- Acc
t3, SUB: RAM_A <- IR[operand]
t4, SUB: Acc <- Acc - RAM_Dout
t4, LDN: Acc <- -RAM_Dout
t4, JMP: RAM_A <- IR[operand]
t4, JRP: RAM_A <- IR[operand]
t5, SKN, (Acc<0): PC <- PC + 1
t5, JMP: PC <- RAM_Dout
t5, JRP: PC <- PC + RAM_Dout
```

We now have a complete computer in digital logic, able to execute machine code programs in RAM.

Summary

The purpose of a digital logic CPU is to execute machine code programs, which can be assembled from human-readable assembly language. These programs need to be placed into memory before the CPU starts its work. They consist of a series of instructions that are in turn read into the CPU and executed.

CPUs can initially scare those trying to understand them. Even a minimal example such as our Baby might take thousands of transistors; modern CPU chips can contain billions. But you've seen in this chapter that the basic structure isn't so complex if you think hierarchically, like an architect. From this perspective, you already saw how to build a variety of simple machines that each perform a basic task; a basic CPU then just connects a small number of these simple machines.

The CU can be built from a sequencer, which triggers the fetch, decode, and execute stages. The execute stage is the hardest one to implement, as it involves different actions depending on what instruction was decoded. The sub-steps of the execute stage therefore need some additional logic to activate the different options.

This chapter has shown roughly how the Manchester Baby was and can be put together. The architecture we built still forms the basic plan for many modern CPUs. Pressures from Moore's law have complicated this plan, however. They prevent modern machines from simply being clocked faster, but they allow them to use many more transistors. In the next chapter you'll see some of the more complex uses that modern CPUs can make of these extra transistors.

Exercises

Build a Baby

1. Build the Baby design from Figure 7-13 in LogiSim Evolution.
2. Once you start working at this level of complexity with sequential logic, it's very easy and common to create hardware bugs around trigger timing. Working architects spend a lot of their time debugging timing issues. The hardware equivalent of a debugger is a *chronogram* (Figure 7-14), a diagram that plots the state of several wires in the system over time. LogiSim Evolution has a built-in tool for generating these (**Simulate ▸ Timing diagram**). Find out how to use it to test some of the sequential subcircuits from the Baby. Recall that sequential logic—write enables for RAM and registers, and RAM read addresses—is usually triggered at the *instant* the clock signal rises from 0 to 1, while combinatorial logic is active at all times. There are also hardware logic analyzers that will capture and display similar data from breadboards, either standalone or sending the data to your PC for analysis.

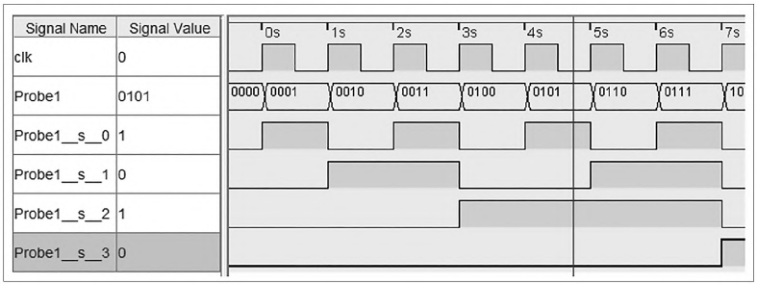


Figure 7-14: A LogiSim chronogram

Programming the Baby

1. Assemble the test programs—including Turing's division program—discussed in this chapter, and run them in your LogiSim Baby. Use the Python assembler provided, with the `for_logisim` flag on line 3 set to `True`. Save the output in a text file and load it as a RAM image in LogiSim by right-clicking the RAM and selecting **Load Image**. You can step through CPU cycles manually by clicking the clock, or by setting it to tick automatically with **Simulate ▸ Auto-tick** in the menu bar. Turing's program divides 36 by 5, to give result 7 (111_2), which gets stored—padded with zeros—in address 28, so it appears as $E000000_{16}$. Try editing lines 29 to 31 to perform different divisions.
2. Can you explain how Turing's code works? Remember that the Baby's two main quirks are that it negates values when loading them, and it has only a subtractor rather than an adder. These lead to a few programming idioms.

Challenging

We've used several layers of notational abstraction in our CPU designs: packaging up transistors, gates, and simple machines into boxes. Estimate how many logic gates, then how many transistors are used in our final design. How does the number of transistors compare with those used in the actual historical designs in Chapter 1? How could these be reduced if we preferred an implementation using less silicon rather than an educationally easy-to-understand design?

More Challenging

The Baby is a very small, simple computer, but it's possible to extend it into a fairly serious modern machine by modifying our design. Try doing this using the following steps.

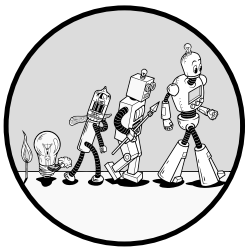
1. Increase the RAM size. To do this, you'll need to increase the size of the addresses throughout the design.
2. Replace the Baby's `LDN` with a more normal `LOAD` instruction, which just loads without negation. Or retain `LDN` and create a new `LOAD` in addition to it, if you want to retain back-compatibility with old code. This will lead to more complexity and silicon, but will keep existing users happy, so is representative of a typical dilemma architects face.
3. Replace the single subtraction module with a full two's complement integer ALU, including add, subtract, multiply, and divide. Create extra instructions to trigger these operations.
4. Look at the later Manchester Mark I and Ferranti Mark I machines to see how the original Baby was actually extended to commercialization. Try to emulate them in LogiSim.

Further Reading

- For the nearest we have to an official modern manual for the Manchester Baby, see the University of Manchester's current web page, *http://curation.cs.manchester.ac.uk/computer50/www.computer50.org/mark1/prog98/ssemref.html*.
- For the original publication describing the real Baby, see F.C. Williams, T. Kilburn, and G.C. Tootill, "Universal High-Speed Digital Computers: A Small-Scale Experimental Machine," *Proceedings of the IEE Part II: Power Engineering* 98, no. 61 (1951): 13–28.
- For details of the later Manchester Mark I, see R.B.E. Napper, "The Manchester Mark 1 Computers," in *The First Computers: History and Architectures*, ed. Raúl Rojas and Ulf Hashagen (Cambridge, MA: MIT Press, 2000), 365–377.

8

ADVANCED CPU DESIGN



The previous chapter presented a minimal CPU design in digital logic. In this chapter, we'll look at extending that basic design to increase performance. These extensions include using more registers, using stack architectures that improve subroutine capabilities and speed, adding interrupt requests to enable I/O and operating systems, floating-point hardware, and pipelining and out-of-order execution to enable "superscalar" execution of more than one instruction per clock cycle. At this level of complexity we won't give full details on how to implement the extensions yourself with digital logic, but you're welcome to try!

Number of User Registers

As we've discussed, the Baby is an example of an accumulator architecture, meaning it has only a single user-accessible register: the accumulator. All

loads go to the accumulator, all stores are taken from it, and when we do two-element arithmetic, such as subtraction, the first element comes from the accumulator and the second directly from RAM, as in a load.

Accumulator architectures are relatively simple to implement, and they give rise to simple instruction sets. The load, store, and arithmetic instructions each need only a single operand. For example, to add the numbers stored at addresses $50A3_{16}$ and $463F_{16}$, we load the content of the first address into the accumulator, then have an "accumulative add" (`AADD`) instruction that adds the content of the second address into the accumulator:

```
LOAD $50A3
AADD $463F
```

Once these instructions execute, the accumulator contains the result of the addition.

On the other hand, accumulator architectures require any data being used to be moved in and out of the CPU every time the data is needed. This can slow the system down, as RAM is typically slower than the CPU. To avoid this slowdown, it can be helpful to provide additional user registers inside the CPU. These extra registers allow more than one datum to be brought into the CPU at a time, so that multiple calculations can be performed without the need for further RAM access. The 8-bit machines of the 1980s typically had a small number of additional user registers, while modern machines might have tens or even hundreds of user registers.

Especially in scientific numerical computing, the ideal for assembly programmers is often to load *all* relevant data into multiple registers at the start of a computation; this allows huge amounts of heavy number crunching within the CPU without requiring any further memory access. In some ways, having more registers makes assembly programming easier, and it allows faster-running programs to be written.

There's always a trade-off around how many user registers a CPU should have, however, as the additional registers come at a cost: they use up a lot of extra silicon, which adds to the costs of design and manufacturing. They're also bigger and use more energy. Then there's the increasing complexity of the instruction set, which in turn requires more silicon in the control unit (CU), with similar additional costs. Likewise, the increasing complexity of the instruction set makes life more complicated for the assembly programmer, whether a human or a compiler. Load, store, and arithmetic instructions now need to have additional operands to say which register or registers are to be used, such as:

```
LOAD  X, $50A3    // load into register X from address 50A3
STORE $463F, Y    // store to address 463F from register Y
ADD   Z, $463F, Y // add register Y to data from 463F, store in register Z
```

Some architectures let us have it both ways: they provide a dedicated accumulator register and accumulative arithmetic instructions, as well as a set of regular user registers. This allows assembly programmers to take

advantage of possibly simpler and faster instructions on the accumulator while retaining the flexibility to work with the other registers as well.

Number of Instructions

The set of available instructions for a CPU, known as its *instruction set architecture (ISA)*, defines the interface between what the programmer can see and use, and what needs to be implemented by the CPU designer. As with any interface, designing an ISA always involves trade-offs. In this case, there's a trade-off between making the assembly language programmer's (or more likely today, the compiler writer's) life easy and pleasant, versus making the digital logic implementer's life easy and bug-free. An ISA that contains instructions in the shapes of human thinking is easier to program and to write compilers for, but it may be hard to implement in digital logic. An ISA that reflects what's easiest to make in digital logic is easy to build and test, but it may not be easy to program or compile to. Then there are also trade-offs between making human assembly programmers happy versus making compiler writers happy.

CISC and RISC are the two historically opposing philosophies of architecture. Most systems actually blur elements of both in various ways, but the CISC versus RISC distinction is still useful to structure our thinking and to consider what aspects of practical designs are more "CISCy" or more "RISCy."

CISC, pronounced "sisc," stands for *complex instruction set computing*. CISC emphasizes the creation of lots of instructions in ISAs. These can include adding many variations on basic instructions that each act as new instructions. For example, loading, storing, and adding can be done in different ways by different instructions. CISC style might also create new instructions that perform more complex arithmetic than we've seen so far, such as the kinds of instructions found in scientific calculators, and even dedicated instructions for particular operations used in signal processing or cryptography.

On the opposing side of the debate is *reduced instruction set computing*, or *RISC*, which says hardware is nasty, expensive to develop, and difficult to debug, so we should make the processor as lean and mean as we can, then do all of the more error-prone work in software, as software is much nicer and cheaper to create and debug. RISC style is to keep the instruction set as small as possible, then focus on making it run as fast as possible. Single complex instructions found in CISC can be performed in RISC using longer sequences of more basic instructions, which you try to make go as fast together as the single CISC instruction due to their simplicity.

Duration of Instructions

In our Baby implementation, our CU is based on a regularly repeating counter cycle that runs independently of any of the events it triggers. Once you start the counter running, its actions follow a fixed sequence

that's completely predestined and blind to what the rest of the CPU is doing. This is known as an *open-loop* architecture, as there's no feedback to the counter about the rest of the CPU's state. Open-loop architectures are relatively easy to design and to debug because of this independence, which is why we used this style for our Baby. The Analytical Engine also uses this style, via its regularly rotating barrel CU.

In a *closed-loop* architecture, by contrast, the timing of triggers isn't set by a central counter. Instead, each stage of work is responsible for triggering the next stage when it's ready to do so. For example, rather than triggering the decode stage from a central counter, it can be triggered by a wire that the fetch stage activates when its own work is done.

The advantage of the closed-loop approach is that some instructions may be simpler than others, requiring fewer ticks to complete. These can use only the ticks that are necessary, then trigger the next instruction as soon as possible, rather than sitting around doing nothing. For example, in our Baby implementation some instructions (`SUB`, `LDN`) need to do work on tick 4, while others (such as `JMP`) do nothing during that tick.

Open-loop style is usually associated with RISC, due to RISC's emphasis on making *all* instructions simple and fast. Closed-loop is associated with CISC, as CISC may want to include single instructions that perform a lot of complex work and take many ticks to complete, as well as short, fast ones.

Different Addressing Modes

RISC and CISC present different ideas about how much work should be done by a single instruction, and how many different versions of each instruction should be provided. In particular, multiple variant instructions can be created that combine memory access with arithmetic.

RISC aims to reduce the size of the instruction set by maintaining a clean separation between memory access instructions and arithmetic instructions. For example, a program to add two numbers together would use two instructions to load each of the two numbers into registers, a third instruction to add them and put the result in another register, then a fourth to store the result in memory, such as:

```
LOAD  R1 $50A3  // load to register R1 the value from address 50A3
LOAD  R2 $463F  // load to register R2 the value from address 463F
ADD   R3 R1 R2  // put into register R3 the result of adding R1 and R2
STORE $A4B5 R3  // store to address A4B5 the result in register R3
```

This separation is often taken as the main defining feature of RISC.

CISC, in contrast, aims to provide multiple variations of the `ADD` instruction to make the programmer's life easier. In addition to `ADD`, which adds the contents of two registers, we could create another instruction such as `ADDM` for "add from memory" that would enable the four-line RISC-style addition program to be written with a single instruction:

```
ADDM  $A4B5 $50A3 $463F
```

We can interpret this as "add the values stored in memory addresses $50A3_{16}$ and $463F_{16}$ and store the result in $A4B5_{16}$." This makes the assembly programmer's life easier, but makes the architect's life harder, as they now need to build extra digital logic in the decoder to decode this extra instruction, as well as additional digital logic in the CU to arrange the sequence of load, arithmetic, and store operations, which were coded explicitly in the RISC version. This design is often taken as the defining feature of CISC.

A CISC-style ISA might also include further variations, such as an instruction to add the content of one memory location ($50A3_{16}$) to the content of one register (R1) and then store the result in a register (R3). For example:

```
ADDRMR R3 $50A3 R1
```

It likewise might include an instruction to add the contents of one memory location ($50A3_{16}$) to the contents of one register (R1) and then store the result in a memory location ($A4B5_{16}$):

```
ADDRMR $A4B5 $50A3 R1
```

Another common variant is to add instructions that use *indirect addressing*, meaning the operand of the instruction contains the *address of the address* to be used. For example:

```
ADDI   $A4B5 $50A3 $463F
```

This means "add the value stored at the address $50A3_{16}$ to the value stored at the address $463F_{16}$ and store the result at $A4B5_{16}$." This is a quite complex instruction that requires the contents of $50A3_{16}$ and $463F_{16}$ to be loaded into registers, but then these values themselves to be interpreted as addresses, and the values at *those* addresses loaded into registers before performing the addition and store. This sounds like a fairly obscure thing to want to do, but it is very common and useful when compiling high-level languages, such as C, that have *pointers*. The indirection operations allow for a fast and efficient hardware implementation of pointer commands.

Of course, there are also variations on this indirect form of instruction, such as an instruction to perform indirection on just one argument and add the result to a register's contents:

```
ADDIR  $A4B5 $50A3 R1
```

We could dream up further variations, such as using the contents of registers as memory addresses for the indirection, performing more than two layers of indirection, and so on.

Offset addressing (aka *index addressing*) modes are another popular ISA inclusion. The idea here is that assembly programmers often need to make repeated use of many variables that they tend to store close together in memory. Their life can be made easier if they can first use a new instruction to specify the address of this general region of variable storage, such as $A7B2_{16}$, then refer to each individual variable by the *difference* between its address

and this region's address, such as 0_{16}, 1_{16}, 2_{16}, 3_{16}, and so on, to pick each of the variables in order. Here we'd use new offset instructions such as:

```
SETOFFSET $A7B2
ADDOOO $2 $O $1
```

This adds the contents of addresses $A7B2_{16}$ and $A7B3_{16}$ and stores the result in $A7B4_{16}$.

Offset addressing was especially nice for assembly programmers in the 1980s, working on machines with 8-bit words but with 16-bit address spaces. This was because they would otherwise need to use two words and two registers every time they wanted to represent a 16-bit address. Using offsets, they could instead divide memory into 256 *pages* of 256 addresses each. They could then choose to work on a single page at a time, using the page's start address as the offset. Then they would need only a single word to specify an address within the page. For example, $A7B4_{16}$ would be considered to be location $B4_{16}$ on page $A7_{16}$.

To implement offset addressing, an additional register is usually added to the CPU design and used to store the offset. Its value can then be joined onto new operands to form complete addresses when needed by later instructions.

It is easy to see how the size of an instruction set can get very large once all these variations come into play. We've only considered variants of a single instruction, `ADD`. To be consistent, an ISA must typically create the same chosen variations for *every* type of arithmetic instruction, which can lead to hundreds of new instructions in total.

Subroutines

The word *subroutine* is one of many names—subprograms, procedures, functions, methods—for a very similar concept: a piece of code sitting somewhere in memory that will do something when your main program *calls* it, and will *return* to the same line in the main program after it was called. This last bit distinguishes a subroutine from a simple jump like the `goto` statement, which forgets where it was called from. The invention of the subroutine is generally credited to Maurice Wilkes and his team around 1950.

In early high-level languages, only one level of subroutine calling was allowed at a time. You had a main program that could call subroutines, but subroutines could not then call other subroutines. More modern high-level languages rely on the ability for subroutines to hierarchically call other subroutines—including themselves, as in *recursion*—to encapsulate complexity.

The term *subroutine* is generally used at the level of architecture and assembly language. The other names are used in higher-level languages and have historically had somewhat different meanings that have never been formally defined or used consistently across most languages. The following list is an attempt at definitions that capture what the words *would* mean if they were ever used consistently by language designers:

Function This is the easiest to define formally, as it's a concept used in the most heavily formalized functional programming languages. A function is (ideally) a mathematical object that takes arguments as inputs and returns a value computed only from these inputs and not from anything else beyond them. The function shouldn't have any other *side effects*, meaning it shouldn't affect anything else.

Procedures This is a name for subroutines in some languages that may or may not take inputs and don't usually return an output, so they act only via side effects. Some older languages allow only one level of calling, meaning procedures can't call other procedures.

Method This is a concept that comes up in object-oriented programming to name a subroutine associated with an object. A method may both return a value, like a function, and have side effects, like a procedure.

All of these names are heavily abused and confused by practical programming languages; for example, it's common for languages to have "functions" that produce side effects and don't return values. Functional programming languages are more likely to enforce the mathematical concept, but even some functional programming languages allow side effects.

Now let's look at how to implement subroutines.

Stackless Architectures

It's possible to implement subroutines purely in software, without any additional hardware or instructions. You could do this by writing programs with jump instructions and then having some convention to keep track of the return address. However, this is hard work for the programmer and slow for the computer.

Early subroutine-capable architectures such as ENIAC added dedicated CPU instructions to call and return, and simple hardware in the CPU to execute them. One approach used a single return address location in hardware. A special dedicated internal register can be easily built into a CPU to store a return address. This allows the main program to call and return from one subroutine at a time; once you're inside the subroutine, you can't call and return from another subroutine, because this would overwrite the single return address.

To enable subroutines to call one another (including recursively calling themselves), architectures can add a hardware stack.

Stack Architectures

A *stack* is a simple data structure with two operations, push and pop. It behaves like a physical stack of papers on your desk. You can *push* a new document to the top of the stack when it arrives, and you can *pop* only the top document on the stack by picking it off and removing it. You can't take documents from lower down in the stack.

Using a stack, you can create a full trail of addresses to return through in the case of nested subroutines. Each time a subroutine is called, its return address is pushed to the stack. When the subroutine returns, this address is popped off the stack and used to set the program counter.

Keeping track of subroutines this way can be especially useful in cases of recursion. The stack typically grows very large during recursive execution and (hopefully) is reduced as the program completes and data is read and removed from the top of the stack. A *stack overflow* error is a failure condition where we run out of stack space; if you use a specific chunk of memory to hold your stack, and you run out of space, the program will create a stack overflow error as you try to write outside the stack boundaries. This usually happens because of something that's gone wrong in an infinite loop of functions calling themselves or each other. In modern computers, stacks are much less resource-expensive to implement than they used to be, so they're the standard way to store return addresses.

Hardware stacks are found in most modern machines from the 8-bit era onward. They use hardware digital logic implementations of the stack concept in their CUs to enable arbitrary subroutine calling with enhanced speed and security. These stack architectures have an extra, dedicated *stack pointer register*, an internal register that contains a pointer to the top of the stack. The stack itself may be stored in some area of RAM, with access to this part of RAM often restricted at the hardware level. For example, a stack architecture might have dedicated digital logic to test all load and store instructions from the user, to make sure they aren't trying to access the stack's portion of RAM. This prevents malicious programmers from interfering with the stack.

Some stack architectures hide their internal workings from the user, and provide only new call- and return-style instructions in the instruction set. When executed, the call instructions will activate digital logic that automatically pushes the program counter to the stack, increments the stack pointer, and jumps to the subroutine. Likewise, the return instructions will pop the program counter, decrement the stack pointer, and jump back to the calling function.

Other stack architectures allow full user access to the contents of the stack in addition to or instead of call and return. For example, some designs provide instructions such as `PHA`, for PusH Accumulator, and `POPA`, for POP Accumulator. The former pushes whatever is in the accumulator onto the stack and increments the stack pointer, and the latter pops the stack to the accumulator and decrements the stack pointer. This design provides a method to pass arguments to subroutines by storing them on the stack along with the return addresses.

CALLING CONVENTIONS

Whatever stack or stackless architecture is used to enable subroutines, it will rely on the programmer to maintain a consistent *calling convention* so the different parts of the program can correctly understand one another as they pass and receive arguments. This is especially important when subroutines are written by a different author from the caller code, as is the case when a general-use library of subroutines is provided. A calling convention includes the following:

- Where arguments, return values, and return addresses are placed: in registers, on the call stack, a mix of both, or in other memory structures
- The order and format in which arguments are passed
- How a return value is delivered from the callee back to the caller: in a register, on the stack, or elsewhere in RAM
- How the task of setting up for and cleaning up after a function call is divided between the caller and the callee
- Whether and how metadata describing the arguments is passed

Calling conventions aren't part of CPU architecture. Rather, they're social agreements between programmers. A given architecture can often be used with any one of many different possible calling conventions. In some cases, CPU architects will suggest a convention to try to discourage fragmentation between their users. In other (or sometimes, the same!) cases, programmers create their own conventions and standards wars break out when they need to interface their programs.

Calling conventions also define additional features for compatibility between modern high-level languages and compilers. For example, compiled executable code from two languages, such as C and C++, can link to and call one another's subroutines as long as they obey the same calling convention.

Floating-Point Units

We saw in Chapter 2 how floating-point numbers are represented with a sign, an exponent, and a mantissa. *Floating-point registers* are specialized user registers designed to store floating-point data representations for use in floating-point computations.

Performing arithmetic on these representations is more complicated than the arithmetic logic unit (ALU) operations on integers seen so far. To multiply two floating points, for example, we need to multiply their mantissas, add their exponents, and multiply their signs. To add two floating-point numbers, we need to shift one of them by the difference in their exponents, then add them, and possibly shift again and update the exponent. Dividing can be error-prone when a large number is divided by a small one, and can also result in special cases defined to yield infinity or NaN (not a number) representations.

This can all be done by combining simple ALU-style operations together and using new components of digital logic. The resulting structure is called a *floating-point unit (FPU)*. FPUs are complex pieces of digital logic and expensive to design; they also take up lots of silicon and are prone to bugs. In 1994, Intel made an error implementing the FPU in their Pentium chip that cost them half a billion dollars in recalls and reputational damage.

FPUs appeared in the 1980s, not inside the CPU but as optional additional chips. For example, the Intel 8086 CPU could be paired with an optional extra FPU chip, the lesser-known 8087. Nowadays, FPUs have all moved onto the CPU and behave similarly to their ALU counterparts.

If you want to see what the dedicated registers and instructions on a modern FPU look like, they take up most of a full book, volume 3 of the amd64 reference manuals, which you'll meet in Chapter 13.

Pipelining

Everything we've looked at so far involves writing a program in assembly language, compiling that into machine code, and executing the machine code from top to bottom, with some branching and looping. Fundamentally, the instructions are brought in one at a time, and each individual instruction is executed before the next is brought in. Most modern CPUs don't work like this. Instead, they work on parts of multiple instructions in parallel. We'll look at many more forms of parallelism in Chapter 15, but those that operate at this CPU level are known as *instruction-level parallelism*, and we'll study them here.

Pipelining is a form of instruction-level parallelism that appeared in the 32-bit era. A pipeline is like a production line, where there are multiple workers doing tasks at the same time, as in Henry Ford's 1913 car factory, shown in Figure 8-1.



Figure 8-1: Ford's 1913 production line

Ford assigned one specialized task to each worker and positioned them at fixed locations along a conveyor belt. Car parts moved along the conveyor, with each worker in turn doing their work on each car part.

Now replace Ford's car parts with instructions from your machine code program, and imagine them being run down this production line. Instead of human workers, you have parts of the CPU performing tasks such as fetch, decode, and execute. Suppose you've written 20 lines of code, and assume there's no jumping or branching. In the CPU designs we've seen so far, a single instruction is placed on the production line and passes by the fetch, decode, and execute workers in turn. Once it gets to the end of the production line, the next instruction is placed at the start of the line. Most of the workers thus end up standing around doing nothing for most of the time when it isn't their turn to work.

We could extract much higher efficiency from our workers by keeping the whole conveyor belt full of instructions the whole time, rather than waiting for one to finish before starting the next one. One worker could be executing one instruction at the same time that a second worker is decoding the next instruction and a third worker is fetching the instruction after that.

There are many different ways of dividing up the work of a CPU into such stages, depending on the architecture type. The classic split considers fetch, decode, and execute stages. Our LogiSim Baby used a cycle of five ticks. Modern CPUs can have many more subdivisions—there are around 37 stages in a modern Intel processor's pipeline!

At the digital logic level, pipelines can be implemented by having the CU trigger multiple components at the same time rather than one at a time. It's common to show pipelines as diagrams like Figure 8-2.

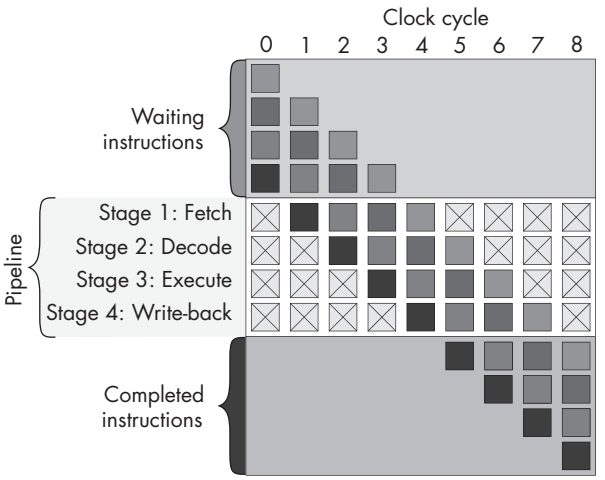


Figure 8-2: A diagram showing how instructions are handled in a basic pipeline

In Figure 8-2, clock cycles go across from left to right, and four instructions (represented by squares) are shown passing through the pipeline. This

is a four-stage pipeline, so it can work on up to four instructions at once (as shown on clock cycle 4), each at a different stage of processing.

Pipelining is simpler and more efficient for open-loop, RISC architectures, in which all instructions have equal durations so they can progress evenly along the pipeline. It can quickly become complex and less efficient for closed-loop, CISC architectures, where different durations for different instructions must be taken into account, and where some parts of the pipeline get left empty as shorter instructions execute alongside longer ones.

Hazards

There are several well-known scenarios, called *hazards*, in which problems occur in pipeline execution. Let's take a look at the main types of hazards. Then we'll consider how to fix them.

Branching Hazards

A *branching hazard* occurs when, somewhere down the pipeline, an `if` statement is found and the other, earlier stages were working to complete one outcome of the branch, but you need to go to the other outcome instead. When a conditional branch is reached, you don't know which condition will be followed until the branch gets executed.

Data Hazards

If two of the workers are trying to hit on the same memory location—to fetch and output to, for example—bad things are going to happen. We can split these *data hazards* into three main categories:

Read after write This is where you have two instructions, the first trying to write to memory and the second trying to read from the same address. The logic of the program is supposed to be that the value that gets read should be equal to what has just been written. But when pipelining is in play, it may be possible for the RAM access of the read to occur before RAM has been changed by the write.

Write after read This is the other way around: here two instructions are supposed to first read the old RAM value and then update it with a write. But when they are interleaved by pipelining, it may be possible for the part of the write instruction that actually changes the RAM value to occur before the part of the read that accesses it.

Write after write This is where two write instructions interfere with one another when trying to write to the same address. The program logic is supposed to be that the first one writes, then the second one, leaving the address containing the second one. But again, pipelining may interleave stages of their executions, in some cases performing the intended first write after the second.

Structural Hazards

The third type of hazard, a *structural hazard*, is where multiple stages are fighting for resources at the same time. In the production line example, the factory might contain one physical calculator on a shelf behind the workers, for their shared use. There may come a time when two of the workers both want to use this calculator at once. For example, one might need to check if something equals zero, while the other needs to execute an addition operation. The digital CPU analog of this would be two regions of digital logic computing two pipeline states both needing to access the ALU or memory at the same time.

Hazard Correction

Pipelining tends to work well for all types of signal processing—including audio, video, and radio processing—because there isn't much branching to handle. The same kind of data is expected to flow through the pipeline in real time and always be processed in the same way. For example, the codecs in your digital TV or laptop used to decode and display movies will reliably chug through frame after frame of incoming video and audio, doing the same operations to decode and display each one, in the same order. They don't usually have to look at the content of the signals and change their behavior in response to this content.

Hazards become more problematic when you're doing computations that are continually checking the results and changing their flow based on this state. As soon as you have programs with branches—and to a lesser extent jumps and subroutines—you have to think about how to address hazards. Let's go through a few general strategies.

Programming to Avoid Hazards

Skilled assembly programmers can write assembly code to avoid many hazards, if they understand the architecture. This often involves considering groups of neighboring instructions and thinking about how they could affect one another in the pipeline, and changing the order of some instructions to make them further apart and less likely to affect one another.

Nowadays, most programming is done in compiled languages, so some of the tricks that end-user programmers once employed have moved into compilers. A good compiler can inspect the assembly code it's produced and look for places likely to lead to hazards. It can tweak this code as a human programmer would to reduce the likelihood of the hazard. For example, the order of instructions that don't affect each other can be swapped to make two accesses of the same data occur further apart in the execution. Of course, there's still a human behind these sorts of optimizations: the authors of the compiler, who have likely taken a strong interest in hazards and how to avoid them.

Some ISAs provide a *null operation (NOP)* as an extra instruction that means "do nothing." NOP instructions still go through the pipeline, taking up time slots, so a human programmer or compiler can insert them between

hazard-causing instructions to spread them out and avert the hazard. This typically requires less intelligence than reordering instructions, but will slow down execution as the NOPs are processed through the pipeline.

Stalling

Stalling (sometimes known as *bubbling*) simply means putting the result of the pipeline on hold to allow some stage to complete its work. For example, if there's a structural hazard and two stages want to use the ALU at the same time, we just let one of them use it and tell everyone else to do nothing until the ALU is free. In the production line analogy, this is like the system used in factories where if a worker gets into trouble they can hit a button to stop the conveyor belt to give them time to fix the problem.

To allow for stalling, additional digital logic can be added to the CPU to detect the upcoming potential occurrence of hazards—for example, temporally ceasing to trigger stages for the next instructions as soon as a jump or branch is seen to be coming in. This is a heavyweight solution that has a large time cost if it's used frequently. As with NOPs, the whole pipelining system is effectively disabled around hazards, so if we have to do this all the time then we might as well just use a non-pipelined CPU. But stalling is relatively simple and cheap in terms of silicon and design time to implement.

Redoing Work

Redoing work means that as a potential hazard instruction is being processed, we allow the following instruction to begin its cycle as normal, with the hope that the hazard won't actually occur. If we later complete execution of the potential hazard instruction and find that a hazard *has* actually occurred, then we throw away the work that's been done on the next instruction and do it again.

For example, when a branch instruction arrives, we assume that it won't be taken, and begin fetching and decoding the following instructions at the same time as testing the branch's condition. If we then find the branch is not to be taken, the work already done on the next instructions is useful and is kept, progressing to execution. But if we find that the branch is to be taken, we discard the work on the subsequent instructions and start fetching and decoding different ones from the branch target address.

This strategy is more efficient than stalling, where a performance hit is taken at every *potential* hazard, even the ones that don't end up actually occurring. Say there are 100 branches in your program, only half of which are taken; stalling would delay all 100 of them, whereas redoing work delays only 50.

Eager Execution

Eager execution means executing *both* possible branches at the same time for a short period, and then killing the one not taken later on, once we figure out which it should be. For example, a typical use of eager execution occurs in instruction sequences such as (using Baby assembly):

```
1: SKN 3
2: LDN 10
3: LDN 11
```

This sequence first asks if a condition is true; depending on the result, it loads from either address 10 or address 11. In eager execution, we begin fetching, decoding, and executing both lines 2 and 3 while we are still executing line 1. Only later, once the result of the line 1 comparison is known, do we decide which of line 2 or 3 is wanted. We keep the work that's been done on the desired line and throw away the work done on the unwanted one.

Implementing eager execution requires doubling up our physical digital logic to perform twice as much computation in parallel during the period of uncertainty. This could involve having two physical copies of the ALU, registers, and execution logic. This can be a good use for the additional silicon that the transistor density form of Moore's law currently provides, while not allowing faster clocks.

Branch Prediction

Branch prediction is where we try to predict whether a branch will be taken *before* actually executing it. Such prediction may initially sound impossible (surely the very meaning of execution is to find out what the branch will do), but we can often make use of prior knowledge to give us at least a better-than-random guess.

For branch hazards, the redoing work approach can be viewed as always predicting that branches won't be taken. It begins to fetch and decode the instruction from the next numerical address while working on execution of the branch instruction. Branch prediction generalizes redoing work by trying to make a more accurate prediction about whether a branch will be taken. Fetch and decode can then begin for whichever branch is predicted, and work redone only in cases where the prediction turns out to be incorrect.

Branch prediction remains an active area of research, with several strategies under investigation. One is to assume that all branches *are* taken—essentially the opposite assumption of the redoing work approach. If users wrote only programs whose branches originated from `if` statements, then these branches would have a 50/50 chance of being taken, in the absence of any other information. However, many or most of the branches that appear in practical machine code originate from loops rather than `if` statements, and the usual purpose of a loop is to repeat many, rather than zero or one, times. Therefore, when branches originate from loops, they usually *are* taken because the user wants to loop a few times.

Large-scale statistical studies of real-world machine code found in the wild have confirmed this: their estimates range from 50 to 90 percent of branches being taken.

In some cases, another strategy is for the human programmer or compiler to provide hints about which branches will be taken. This could include

human programmers adding special comments to their assembly or high-level code, or compilers using code analysis to create their own predictions and annotations. For some compilation tasks, this is easy to do—for example, if a user program says "repeat 100 times," then we can make a good prediction. Predictions are harder—in some cases uncomputable—to make in the case of `while` loops.

A third, state-of-the-art approach is to use dynamic runtime branch prediction, which involves building statistical or machine learning classifiers into CPU digital logic and using them to make on-the-fly predictions. As with all prediction systems, this requires choosing some features of programs that may be informative about temporally and spatially nearby branch behaviors.

Simpler cases include keeping a log of observed frequencies of branch-taking at each branch instruction during execution of the user program, and using these frequencies as probabilistic prediction for which way the branches will go if the same instructions are executed again.

More advanced cases now include linear regression and even neural network classifiers built from digital logic and pretrained on large collections of machine code gathered from real-world programs in the wild. These may be trained on all kinds of features of the machine code, such as values of opcodes and operands in many lines before and after a branch instruction.

Operand Forwarding

Operand forwarding is a technique for avoiding data hazards by adding digital logic to directly route the result of an instruction to become an input to a next or nearby instruction. For example, consider this program:

```
1: ADD R3 R1 R2
2: ADD R4 R3 R1
```

This computes R3 = R1 + R2, then R4 = R3 + R1, where all the operands are registers. Here, instruction 2 requires the result of instruction 1 to be placed in R3 before instruction 2 can execute. This will result in a data hazard for most pipelines. However, the value destined for R3 may in fact be available on the output lines of the ALU during execution of instruction 1, but before it appears in its destination register. By connecting a physical wire directly from the ALU output to where the data is required (such as an ALU input), we can bypass the wait for the result to be deposited and reread, and instead start using it immediately.

Out-of-Order Execution

Out-of-order execution (OOOE) is a more advanced form of instruction-level parallelism than pipelining. It involves actually swapping around the order of instructions as they come into the CPU, so they're executed in a different order than they appear in the program. OOOE architectures were first enabled in theory in 1966 by Tomasulo's algorithm, and appeared commercially in the 1990s.

The key to OOOE is recognizing that instructions in serial programs can often be swapped without changing their results. For example, if we have some variables being assigned values, we can make those assignments at any time before the variables are next used without affecting the result; we'll end up with the same state overall. This gives us freedom to swap instructions around to prevent pipeline hazards from occurring and to maximize efficiency. Whether we have single or multiple hardware copies of CPU substructures available, we can choose orderings that try to make the best use of these resources, keeping them all as busy as possible.

To see how OOOE works, consider the following program:

```
1: DIV R1 R4 R7
2: ADD R8 R1 R2
3: ADD R5 R5 R9
4: SUB R6 R6 R3
5: ADD R4 R5 R6
6: MUL R7 R8 R4
```

There are six instructions here. Instruction 1, for example, sets register R1's contents to the result of dividing the contents of register R4 by the contents of register R7. We'll assume we have simple machines available for division and multiplication, but that they're slower than those for addition and subtraction. The left of Figure 8-3 graphs the dependencies between the instructions.

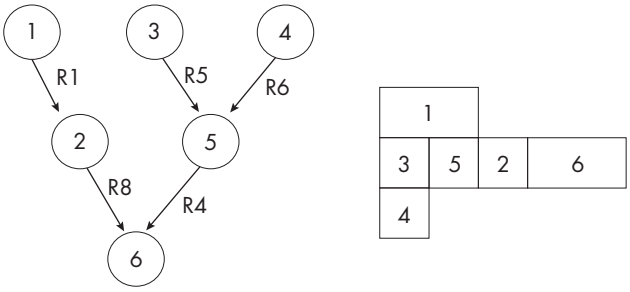


Figure 8-3: A dataflow graph (left) and a schedule (right) for the sample program, allowing for OOOE

As an example of a dependency, consider that instruction 2 can't begin execution until instruction 1 is complete, because instruction 1 writes to register 1, which is needed as an input to instruction 2. The dependency graph shows that we don't need to execute the instructions exactly in their original order. As long as any instruction is executed after all of its parents in the graph, the result will be the same. We can reorder the sequence of instructions and/or execute them in parallel as long as the arrows in the graph are respected.

The right of Figure 8-3 shows one possible ordering for execution of the instructions. Here, instructions 1, 3, and 4 are executed in parallel to start. Instruction 5 comes after 3 and 4, but can still occur in parallel with 1 (which takes longer, being a more complex division operation). Instruction 2 can

occur once 1 has finished, and 6 (another longer instruction, as a multiply) must come last, since it needs the results of both 2 and 5. Depending on how many ALUs we have available, we could execute this or similar schedules much faster than a single series or even a pipeline of the original program.

OOOE is usually performed by digital logic in the CPU, in real time during program execution. Usually only a short window—such as 10 or 20 instructions—around the current instruction in the program is considered for reordering.

NOTE *If you extend the idea of OOOE into reordering and parallelizing entire programs, you'll arrive at GPU dataflows, which you'll meet in Chapter 15.*

Hyperthreading

In a basic CPU, only the fetching hardware is active during the fetch stage, only the decoder is active during the decode stage, and only the ALU or CU is active during the execute stage. Pipelining and OOOE are two ways to make better use of the CPU hardware resources that are otherwise idle during the fetch-decode-execute cycle, by having them work on parts of multiple instructions at the same time.

Hyperthreading is another way to make use of CPU resources when they would otherwise be sitting idle during the cycle. Rather than work on consecutive instructions from one program, we put them all together to form a second virtual CPU core that operates on a separate set of instructions. Each component of this virtual core runs out of phase with its use in the main CPU core, when it would otherwise be idle. By collecting all the components together, all out of phase, we create a whole extra CPU, keeping all the silicon in constant use at all times.

Hyperthreading was conceived in the 1970s and became widespread in commercial CPUs during the 2000s. It effectively doubles the number of apparent cores over the number of physical cores in a device, which is why you often see your computer report having twice the number of cores that were advertised on the hardware you bought.

Hyperthreading has the advantage over pipelining that you no longer have to worry about hazards because the two cores can operate completely independently of one other. On the other hand, it doesn't increase the speed of any one program. It also requires additional digital logic to read, store, and write the states of the two virtual CPUs at the right times, and duplication of some hardware components, so that one doesn't affect the other. In practice, pipelining and hyperthreading may be used together, especially when pipelines are broken down into many smaller stages. Figuring out how to balance them is advanced work that's beyond the scope of this book.

Summary

Beyond a minimal CPU such as the Baby, architects are faced with many decisions about what trade-offs to make between speed, usability, silicon size, and energy costs. Adding more features to a CPU, such as more registers, ALU and floating-point simple machines, stacks, and different addressing modes, can make life easier and faster for the user programmer or compiler, but at the cost of silicon and energy. Likewise, adding more instructions can make life easier for some programmers and compilers who may ask for them, but harder for others who have to keep up with the extra complexity. Giving all instructions the same fixed duration makes life easier for pipeline and OOOE designers and CPU debuggers, but may be less efficient if a few complex instructions that require long execution times are in use.

RISC is a style that generally aims to keep instructions and instruction sets small and simple, while making use of extra silicon to speed up the instructions via more registers, pipelines, and OOOE. CISC is the opposite style: it prefers to make use of extra silicon to add more complex instructions and create larger instruction sets. The two styles tend to fit different applications, as we'll see in Chapters 13 and 14.

Even with the most advanced CPU design, your computing experience would be very limited without input, output, and memory, and the next two chapters will look at how to add these to your computer.

Exercises

Confusing a Pipeline

Design the simplest assembly program needed to confuse a basic pipeline, making it run as slowly as a non-pipelined system. Try to extend this program to further confuse each of the hazard-handling strategies as much as possible. How likely are such programs to occur in practice, and what might be done to avoid them?

Challenging

Try to build an FPU in LogiSim, based on the floating-point data representation seen in Chapter 2. Consider how previously seen simple machines can be combined in each of the arithmetic operations of addition, subtraction, multiplication, and division. For example, when two floats are multiplied, their exponents are added.

More Challenging

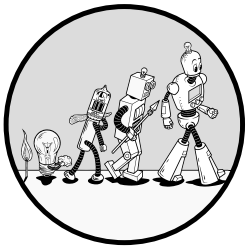
Try to extend the previous LogiSim Baby design with some minimal pipelining. For example, you could try to increment the program counter and start performing the next fetch while the current instruction is executing. The hard part is dealing with branching hazards. You may want to assume initially that the branch will be taken, then add logic to clear things out and start again if this turns out to be incorrect.

Further Reading

- For the origin of the controversially named "von Neumann" architecture, see John von Neumann, "First Draft of a Report on the EDVAC," June 30, 1945, *https://history-computer.com/Library/edvac.pdf*.
- For the invention of subroutines, see Maurice Wilkes, David Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines* (Cambridge, MA: Addison-Wesley, 1951).
- *Human Resource Machine*, *Shenzen I/O*, and *TIS-100* are educational video games that present CPU-like environments with different instruction sets and goals for you to explore.

9

INPUT/OUTPUT



You've seen how to build a basic CPU and RAM, which together can run programs. CPU and RAM are great for performing calculations, but to bring a computer to life with graphics, sound, joysticks, and other interactions with the real world, we also need input and output, known together as *I/O*. In this chapter, you'll see how to add I/O capabilities using buses, I/O modules, devices, and peripherals.

Basic I/O Concepts

To discuss I/O in detail, let's first define a few terms. *I/O modules* are digital electronics that—like RAM—are assigned and connected to addresses in the computer's *address space*, the range of possible addresses that the CPU can access. I/O modules are also connected to *devices*, which are electronic systems, including digital and analog electronics, that aren't connected directly to the computer's address space but that can communicate with it via the attached I/O modules. Devices may be physically inside the computer, such

as an analog circuit that controls the scanning beam of a CRT monitor, or outside it, such as the electronic circuits inside a printer.

Peripherals are the most obvious elements of I/O for most computer users: they're the physical objects that connect to the computer from the outside, such as mice, joysticks, monitors, and printers. Peripherals are encased in their own plastic and connect to the computer's box via a wire that the end user can easily plug and unplug. Some peripherals, such as printers, physically contain their devices. Others, such as monitors, rely on a device inside the computer box (CRT controllers, in the case of a monitor).

In the 8-bit era, computer design meant building a complete computer by buying and connecting together CPU, memory, devices, and logic chips, and perhaps custom-designing I/O modules. For example, the Commodore 64 mainboard shown in Figure 9-1 shows that a large part of the machine is devoted to I/O.

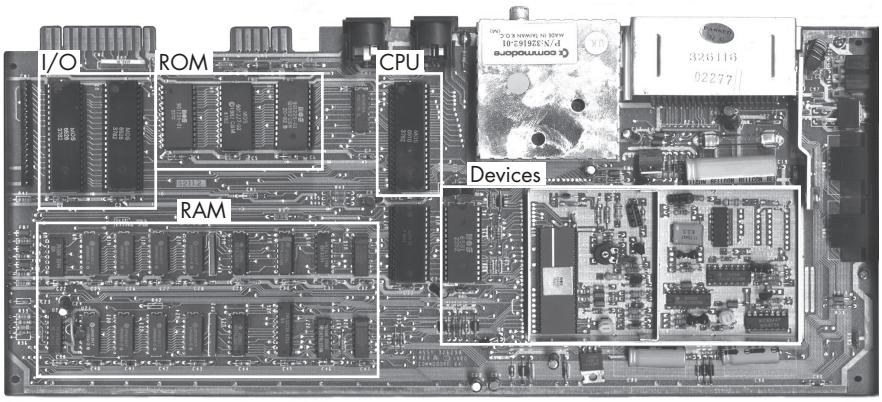


Figure 9-1: A C64 mainboard, showing CPU, memory (RAM and ROM), I/O modules, and devices

The I/O section in the top-left of the figure includes two *Complex Interface Adapter (CIA)* chips, each of which contain multiple I/O modules. The devices section in the bottom-right includes graphics and sound chips.

Nowadays, something like the entire mainboard of the Commodore 64 is shrunk down to a single system-on-chip (SoC) in your phone, but the structure was easier to understand and learn from when the parts were in physically separate integrated circuit (IC) packages. Keep this image in mind as we move through this chapter.

To the CPU, I/O modules appear exactly like part of RAM. They have addresses that can be read and written to, using the same load and store instructions as reading and writing to RAM. Now that we have both RAM and I/O modules connecting to the same CPU address and data lines, we need a way for them to share these resources. This can be done using the bus architecture seen in the next section. After discussing buses, we'll look inside the I/O modules and see how to communicate with them in more detail.

Buses

A *bus architecture* is a specific type of network architecture in which every device involved in communication has equal access to a shared wire or set of wires, called the *bus*. Like the public transportation vehicle of the same name, a computer bus is so-called because it is a public place (it abbreviates the Latin *omnibus*, meaning "for all"). To illustrate the public quality of a bus architecture, consider the example of a prison water pipe tapping system: all the prisoners in a prison have plumbing connected to the same pipes, so tapping on one pipe to transmit a plan for a prison breakout in Morse code inevitably broadcasts the message to anyone and everyone who is listening to the pipes. There's no privacy in a bus architecture (unless encryption is used), which may have interesting security implications if untrusted devices are allowed to access it.

A bus is the simplest form of network, lacking the complexity of the internet's packets, error handling, and routing. For example, it's possible that two prisoners will try to tap the pipe at the same time, creating a collision that destroys both of their messages.

In general, a bus is composed of several *nodes* (things that want to talk to each other) and communication lines (wires) between them. Modern buses usually have many lines used in parallel, though there are also buses with only one. We may divide these lines into control, address, and data lines.

A protocol is needed to ensure that signals don't collide with the signals being sent by other nodes, so for a node to send a message to another it must first announce whom the message is for—the address, on the address lines—and announce what type of message it is—the control, on the control lines. It then broadcasts the data on the data lines. Either you can have one of the nodes in charge of the bus, enforcing the protocol by only allowing nodes to write when it gives them permission, or you can trust the nodes to implement the protocol themselves and play nicely with each other.

THE VICTORIAN INTERNET

The telegraph system of Babbage's and Boole's time has been called the "Victorian Internet." It was a bus architecture connecting sites in Britain, America, and the British Empire. Human operators at a local station would tap out Morse code text messages (telegrams) for customers, and listen for messages addressed to their station from elsewhere. All of the messages were transmitted onto the same wire, which could be written to and read from by all operators. These operators spent thousands of hours listening and writing to the wire, becoming fluent in Morse code and developing their own Morse "speaking" styles that could be used to recognize who was talking. They would also chat with one another when not sending messages for customers, engaging in typical modern chat-room behaviors such as using abbreviated slang (textspeak), falling in love, and even getting married to operators on other continents without having met them in person.

An advantage of a bus architecture is that it's easy to add new devices to the bus, as the same set of shared wires connect all the components. The shared wires also make buses cheap to implement. On the other hand, the bus can be a bottleneck, limiting the performance of the system. This is particularly annoying if you're optimizing your CPU or memory to go very fast, only to then have the data hit a bus and slow down. Bus performance can also be limited by physical factors such as wire length and the number of connections.

Bus Lines

The lines on the bus are the same as the wires we've previously connected point-to-point between CPU (or its cache) and RAM. There are three different kinds:

Address lines These are used to designate the source or destination of data on the data bus. The width of the address bus determines the maximum possible memory capacity (that is, the amount of memory a system can address). For example, a system with a 32-bit address bus can address 2^{32} (4,294,967,296) memory locations. If each memory location held an 8-bit word (byte), the addressable memory space is 4 GiB. For a 64-bit address space of 64-bit words, exactly 1 zebibit ($2^6 \times 2^{64} = 2^{70}$) of memory can be used, which is enough to allow all of the data in a search engine–sized data center to have its own RAM address.

Data lines These provide the path for the actual transfer of data among nodes. A key performance factor is the width of the data bus (that is, the number of data lines). A typical data bus consists of 32, 64, 128, or even more separate lines. To send messages that are longer than the data line width, you need to split them up and send them over several cycles. For example, if a data bus is 32 bits wide and each instruction is 64 bits long, then the CPU must access the memory module twice during each instruction cycle.

Control lines These are used to control access to and use of the data and address lines. For example, the write-enable wire used previously in Figure 7-10 when we discussed the Baby's store operation is a simple control line. More generally, as the data and address lines are shared by many components, there must be a means of controlling their use such that multiple components don't attempt to write to those lines at the same time. Further control lines can be used to request and negotiate for this access.

The CPU-Bus Interface

Most CPUs are designed to connect to an external bus, printed onto the mainboard, via pins on the CPU chip connecting to sockets on the mainboard. Where the bus wires physically connect to the CPU, the connection is known as the *front side bus (FSB)*. The vast majority of a CPU's pins are taken

up by the FSB, as can be seen in the Commodore 64's 8-bit 6502 CPU chip pinout and the 1990s 32-bit Intel Socket2 pinout in Figure 9-2.

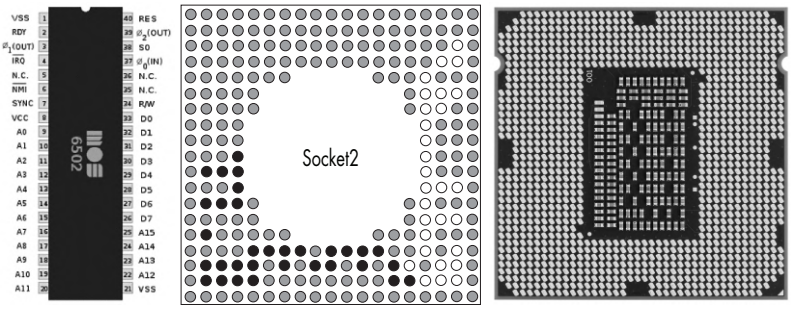


Figure 9-2: Pinout diagrams for an 8-bit 6502 (left) and 32-bit Socket2 chip (center), plus a pin photo of a 64-bit LGA1155 socket CPU (right)

The 6502 here uses 16-bit addresses and 8-bit words, so it has 16 address (A) pins and 8 data (D) pins. The R/W pin is for the read/write control line. In all, more than half of the chip's 40 pins are devoted to the bus. The Socket2 was used with 32-bit address spaces and 32-bit data words, so it has 32 each of A and D pins (colored white and black in the diagram, respectively). Meanwhile, 64-bit CPU chips and sockets need twice as many of each, requiring them to be smaller and more fragile.

The CPU needs to communicate with the bus, but the bus is usually slower than the CPU. Hence, CPU designers prefer to use registers to stage data going in and out of the CPU (as in the Analytical Engine's ingress/egress axes, connecting the CPU to its mechanical rack bus). The bus is a scarce resource, so we don't want to use it for any longer than needed; if data is staged, it can be put on and off the bus at whatever time the bus becomes available. Typically, these staging mechanisms include a *memory address register (MAR)*, which stores the address from which we want to read or write, and a *memory buffer register (MBR)*, which stores a copy of the data being written to or read from that address, as shown in Figure 9-3.

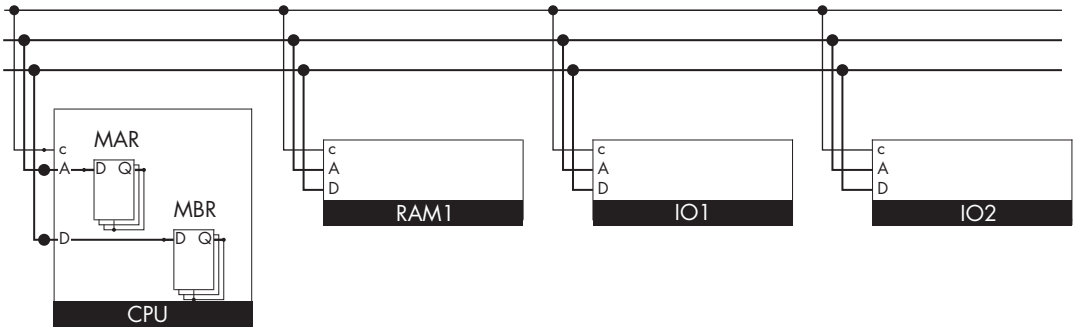


Figure 9-3: A bus architecture, including a CPU, a RAM module, and two I/O modules

To execute a load instruction, the operand containing the address to be loaded is temporarily connected from its instruction register (IR) bits to the MAR, creating a copy in the MAR. When this copy is completed, the MAR is temporarily connected to memory as a read request, and the data from memory is temporarily connected to the MBR, which takes a copy of this data. Then the control unit (CU) can temporarily connect the MBR to the accumulator or other user register. In register transfer language (RTL) style, this can be written as:

```
t2, LOAD: MAR <- IR[operand]
t3, LOAD: BUS_A <- MAR
t4, LOAD: MBR <- BUS_D
t5, LOAD: ACC <- MBR
```

The same MAR and MBR registers can be used to execute a store instruction as well. The CU temporarily connects the MAR to the operand bits in the IR containing the address to be written to; the MAR takes a copy of the address. Then, the CU temporarily connects the MBR to the register containing the value to be written; the MBR takes a copy of the value. The MAR and MBR now contain all the required information describing the store. Finally, the CU temporarily connects the MAR to the RAM's address lines and the MBR to the RAM's data lines, and sets its command line to store, which performs the store in RAM. In RTL this can be written as:

```
t2, STORE: MAR <- IR[operand]
t3, STORE: MBR <- ACC
t4, STORE: BUS_A <- MAR
t4, STORE: BUS_D <- MBR
t4, STORE: BUS_C <- True
```

Having the MAR and MBR also simplifies the design of CPUs that have multiple user registers rather than just an accumulator. They make it easy to separate out the logic and timing for selecting which register is to be connected to the bus from the logic and timing of transferring the data to and from the bus.

I/O Modules

Devices usually attach to a computer via an I/O module. This is a chip that sits on the bus and at some addresses; it looks just like RAM to the CPU. If you learn only one thing from this chapter, it should be this: I/O modules appear to the CPU and assembly programmer as an area of readable, writable memory, just like main RAM. Unlike RAM, an I/O module also

has wires coming out of the other side that go to the device. The module presents a standardized interface to the CPU, and translates requests from the CPU to specific signals on the wires to the particular device. Hence, we can buy any device, such as a sound chip, on eBay and install it in a particular type of computer, as long as we make an I/O module that provides a suitable address space for the computer and sends whatever signals the sound chip is expecting.

Storing to these addresses might transmit commands to the device; it might write assembly-like instructions specific to the module for further translation into device commands (used in modern graphics cards, for example), or it might send data to the device (such as what audio to play). Loading from these addresses might read data from the device, such as a keyboard key press or a microphone sound wave, or read status information from the device, such as whether there's a printer jam. It's up to the designer of an I/O module how they want to interpret these load and store commands.

Some I/O addresses may be implemented by actual RAM inside the module (distinct from regular RAM chips, because this specialized RAM has extra connections to the rest of the I/O circuitry); other times, it may just be immediate digital logic with no RAM. Both methods present the same interface to the CPU, which doesn't know if there's real memory there or something else.

In addition to device communication, the I/O module will also usually handle control and timing, data buffering, and device errors. Let's turn there now.

Control and Timing

An I/O module must be able to coordinate the flow of data between the internal resources and external devices. The latter may be slow, so the module manages them independently of the CPU. This allows the CPU to go and do other things while it's waiting. This is a form of non–CPU level parallelism.

The I/O module achieves this independent management by using data buffering to transfer data into and out of main memory or CPU. *Buffering* means using a dedicated area of memory, called a *buffer*, as a staging area. Slow devices can take their time writing to or reading from a buffer, independently of the CPU. The fast CPU can also read or write to the same buffer, independently of the device.

Ring buffers are used in audio and similar real-time signal-processing I/Os. Conceptually, a ring buffer is a region of data in which the data items are organized in a circle, each with a previous and next neighbor, as in Figure 9-4.

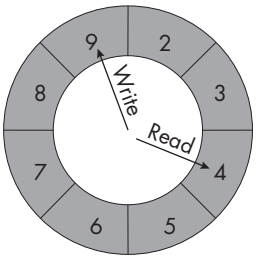


Figure 9-4: A ring buffer. Both pointers move clockwise. The string 0123456789 has been written, with the initial 01 now overwritten by 89. Of this, 01234 has been read.

Two pointers—which can be visualized as clock hands—keep track of the read point and the write point. As new data arrives in real time, it's written to the write point, which is then incremented to point to the next slot. Eventually the write pointer makes it all the way around the ring, at which point it starts overwriting old data. The user program can request to read the next available items at any time. When this happens, the data at the read pointer is copied out and the read pointer is incremented until the number of items requested is met or the read pointer hits the write pointer, meaning there's no further new data available.

Double buffers are often used for graphics rendering. Here, two buffers are maintained, each representing the layout of the screen, as in Figure 9-5.

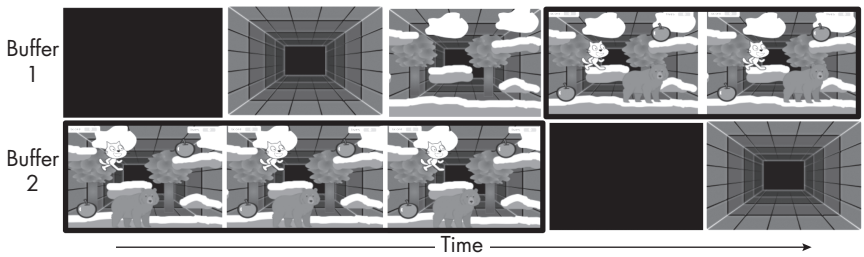


Figure 9-5: Double buffering

At any instant, one buffer stores a completely rendered image and is connected (shown by the thick black outlines in the figure) to the graphics display hardware, which works to physically display it. Meanwhile, the other buffer is used to gradually build up the next image to be displayed—for example, by drawing the background and then adding sprites and overlays to it. Only when the new buffer is finished is the output line swapped over to connect it to the display; then the original buffer is cleared and used to start drawing the third image in the sequence. This approach means that only completed images are ever shown on the screen, which avoids flickering images that show the parts of the images being built up in real time. (Triple buffering is also used in some cases, allowing *two* future frames to be drawn in parallel while the current frame is being displayed. This can achieve a

higher frame rate, as long as it's known far enough in advance what will be wanted.)

Error Detection

Another major function of I/O modules is device error handling. What should an I/O module do if the CPU asks it to do something, but then it detects an error from its device? The error could be a mechanical or electrical malfunction in the device (for example, a paper jam in a printer or a bad disk track), or it could be the result of unintentional changes to bit patterns as they're transmitted between the device and the I/O module, often due to noisy external cables.

Typically, a device shouldn't report an error directly to the CPU, which may be busy doing other things. Instead, it reports the error to the I/O module, which can then pass it on to the CPU.

I/O Module Techniques

Transferring data from an external device to the CPU requires several steps. First, the CPU writes to the bus, asking the I/O module to check the status of the device. Next, the I/O module returns the device status in reply, also writing to the bus. If the device is ready, the CPU requests transfer of data with another bus write. The I/O module then obtains a unit of data from the device. This data is finally transferred via the bus from the I/O module to the CPU.

This process will be slow if it requires waiting for things in the real world. If a gigahertz CPU asks to read 100 audio samples, and audio samples arrive only at 44 kHz, it will need to spend most of its time doing nothing and waiting around for each of these samples to arrive and be sent on the bus by the I/O module. We would prefer the CPU to keep busy doing other things while waiting for the requested I/O to take place. This can be accomplished via three common techniques. We'll discuss each in turn.

Polling

Suppose your boss needs you to get a report finished. One management strategy they could use is *polling*, in which they repeatedly come back and ask you, "Have you finished that job yet?" every hour, day, or month.

A CPU can similarly use polling to check whether and when an I/O request has completed. The CPU requests an action by the I/O module over the bus. The I/O module starts to perform the requested action, setting appropriate bits in an internal I/O module status register as it goes. The CPU then periodically checks (or polls) the status of the I/O module by reading this status register until it finds that the action is complete.

For example, the CPU could ask a webcam's I/O module to grab a new frame of video data. It could then poll until the status is reported complete, then load the data from the module, knowing that it's ready.

Advantages of polling are that it's simple to implement and the CPU has direct control over I/O operation, requiring very little hardware support. The disadvantage—as in the human boss case—is that the CPU must periodically poll the module to check its status. This ties up the CPU, creating long periods where it does no useful work. The CPU is slowed to the speed of the peripheral, which is inefficient. Just like for humans, it gets quite exhausting needing to remember to ask, every day, if you've done your job yet—and to do the actual asking, too. It interferes with the mental workflow of other tasks for the manager and the worker.

Interrupts

Most managers would prefer to ask you to tell them when you've done your job, so they can forget about it until you take the initiative to tell them it's complete. This approach is an example of an *interrupt architecture*. It frees the manager up to focus on other useful work.

In a computational interrupt architecture, the CPU is extended—for example, by adding an extra register and an instruction to set its contents—to enable the programmer to tell it the address of a special subroutine called a handler. The CPU is also extended by adding an extra dedicated physical pin called an *interrupt request (IRQ)* input, along with adding digital logic to the CU to make use of the pin. A high voltage on the IRQ tells the CU to alter the program flow by immediately calling the handler subroutine.

To use an interrupt architecture, the IRQ pin must be connected to a dedicated output from the I/O module. The programmer creates a handler subroutine intended to be executed once the I/O work is done, and tells the CPU its address. The programmer then writes a main program that instructs the I/O module to do actions. When an action command is sent to the I/O module, the CPU forgets all about it and continues executing the main program. The I/O module makes its device do its thing, which can take some time. When the device is done, the I/O module interrupts the CPU by setting the IRQ line to *high*. This calls the handler subroutine, which makes use of the new data from the device or tells it what to do next. Like any subroutine, calling the handler includes storing and returning to the value of the program counter for the instruction being interrupted, so the main program resumes after the interruption has been handled.

An advantage of interrupts is that they're fast and efficient, with no need for the CPU to wait or to have to manage polling requests. A disadvantage of interrupts is that they can be tricky to write, especially when multiple I/O modules are in play, all sending interrupt signals at the same time. A *re-entrant architecture* allows interrupt-handling subroutines to be themselves interrupted by higher-priority IRQs, while a *non-re-entrant architecture* might ignore or delay these meta-interrupts. The code for a re-entrant architecture needs to think very carefully about how to handle meta-interrupts correctly, as a form of concurrent programming.

CPUs have a finite number of physical IRQ pins—sometimes fewer pins than there are devices that want to use them. Pins are a valuable, limited "real-estate" resource for modern CPUs, as adding more pins would force an increase in the physical package size of the chip.

IRQ HELL

Interrupts were a major bane in the lives of computer music creators in the 1990s, as they needed to use a lot of external devices, such as multiple sound cards, MIDI cards, and input controller devices, all at the same time. You would get several physical IRQ lines on your Intel CPU chip, each intended to represent one physical device connected to the computer. If you had more devices than available IRQ pins, you needed a hack to get around this limitation. Hacks included trying to convince hardware and drivers made by different manufacturers to share a single IRQ line, or disabling IRQs used by system hardware to free them up for use by audio devices. Sometimes the latter had system-destroying side effects.

Direct Memory Access

Both polling and interrupts are very slow for tasks that involve transferring large amounts of data from a device (such as a hard drive) into RAM. For example, if we request a 1Mb transfer, the I/O module will go off and do this, leaving the CPU free and happy, but when the interrupt is made this will create a big, slow job for the CPU to load every bit of that data into registers and then send it out to RAM. *Direct memory access (DMA)* is a technique to avoid this problem.

DMA requires a dedicated hardware DMA controller (an I/O module itself) to be placed on the system bus. So far, all our uses of the system bus have involved the CPU talking to another node on the bus, which may be RAM or an I/O module, but buses also allow non-CPU nodes to communicate directly with one another, independently of the CPU. Any node can put a message to any other node on the bus, and this is done in DMA: the CPU grants authority for the I/O module to communicate directly with RAM over the bus, reading from or writing to memory without any CPU involvement.

This frees the CPU to do other things; as with an IRQ, the CPU can "set and forget." DMA usually sends an interrupt when a task is complete, so the CPU is involved only at the beginning and end of the transfer. This is especially useful for large data movements because the data doesn't have to go through the CPU.

I/O Without Modules

I/O modules are the preferred architecture for I/O in most cases, but other module-less I/O architectures also exist and have their places in the world. We'll consider some of them now.

CPU I/O Pins

Some older CPUs, as well as some modern embedded CPUs, forgo I/O modules and bus-based I/O and have the CPU communicate directly with a few specific devices via dedicated pins. This approach isn't scalable, as pins are a limited and valuable CPU resource (they determine the package's physical size). But it can reduce the complexity of the architecture in cases where we know firmly in advance that only a couple of specific devices will ever be attached. If the whole I/O system is designed this way, it can remove the need for IRQ pins and control logic. It also frees up the bus for other activities.

Memory Mapping

Rather than having an addressable I/O module, some architectures use areas of regular RAM as the interface between CPU and device. In these architectures, the RAM is readable and writable by both the CPU and the device (so it needs some extra non-bus wires connecting the pins to devices and to the bus). With this setup, the CPU writes directly to actual RAM as usual, then the device (or a module-like chip interfacing between the device and the RAM, but not on the bus itself) reads out of the RAM and translates into device commands like an I/O module. To the programmer it might be invisible whether the video RAM addresses they're writing to are in fact regular RAM used in this way, or whether they're part of a hardware I/O module.

Bus Hierarchies

In modern architectures we often have more than one bus, forming a hierarchy of buses, as shown in Figure 9-6.

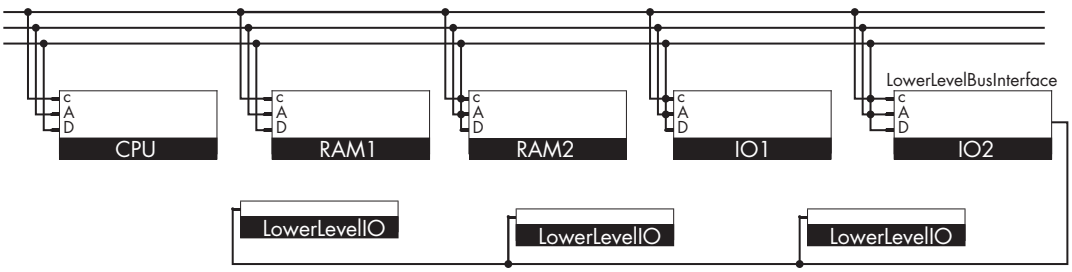


Figure 9-6: A bus hierarchy

The upper level here shows the same bus as in Figure 9-3. However, the I/O module IO2 is an interface to a lower-level bus, which hosts three further lower-level components. This structure can improve usability and speed. Traditionally, each I/O module was connected to a single device and had to be mounted at particular addresses in address space when the computer was turned on. It was hard to add or remove ("plug and play") devices while the computer was on. By introducing a single I/O module, such as a USB hub at a fixed address, we can allow for multiple plug-and-play devices to all connect to this same I/O module via a lower-level protocol, USB. This arrangement also solves the IRQ hell problem, as the I/O module can use a single, valuable IRQ line to alert the CPU to interrupts from any of these devices. The lower-level bus can be built from slower and cheaper technology than the system bus, as it only needs to run at the speed the data is actually available (which may be limited, for example, by waiting for real-world audio or spinning hard disks).

Summary

For a computer to interact with the outside world, such as through graphics and sound, it needs input and output. This can be achieved through I/O modules, which are digital logic components that to the CPU look and act like RAM. Stores sent to their addresses are interpreted as commands to control devices in the outside world, while reads from them are used to send data that has been obtained from sensors in the outside world.

CPU, memory, and I/O all share the same address space and communicate using a shared, public bus of wires, which include address, data, and control lines. CPUs interface to the bus via the staging registers MAR and MBR.

CPU may also interface directly to a limited number of I/O modules via interrupt lines, which the I/O module uses to request the CPU to jump to a handler subroutine. I/O modules have also become increasingly independent of the CPU and can use methods such as DMA to communicate with one another and with RAM over the bus without involving the CPU.

An important use of the bus and of I/O is for managing real-world memory, including multiple physical RAM and ROM modules, and hard disk and optical disc devices. We'll study these in the next chapter.

Exercises

Challenging

1. Take your LogiSim Baby from Figure 7-13 and extend it so that storing to one of its addresses acts to turn a simulated LED on and off.
2. Extend it again so that loading from another address acts to read the state of a simulated switch. You can do this by reducing the size of the RAM by two addresses, then adding a new digital logic

I/O module to the bus that listens for those addresses and acts accordingly.

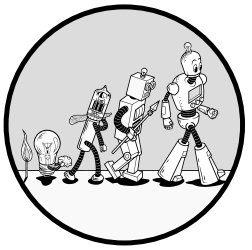
3. Extend it yet again so that the I/O module decodes multiple commands sent as data inside the store instruction and uses them to command the LED to do different things, such as flash at different speeds. Here the LED and switch represent general devices that could be controlled in this way.

Further Reading

See *The Victorian Internet* by Tom Standage (London: Weidenfeld & Nicolson, 1998) for a comparison of the 19th-century telegraph and the modern internet.

10

MEMORY



So far we've constructed registers and a small, Baby-sized RAM to use as memory. We made these from flip-flops. Larger memories can't usually afford to use flip-flops, however, so they're typically made using other technologies, like DRAM and hard disks. These other technologies are slower, creating a trade-off between speed and size. In this chapter, we'll look at the details of larger memories. We'll discuss primary memory, caches, and secondary and offline memory, and begin by looking at the memory hierarchy.

The Memory Hierarchy

At any point in time, usually only some of our data is important and in frequent, current use. Other data is used occasionally, and some is out of use entirely. We usually want to arrange our data so that the parts in working use are kept in fast, easily available memory, while the other parts are kept in slower, cheaper memories. This arrangement is known as a *memory hierarchy*.

Memory hierarchies played out in pre-digital life, too. For example, people used to carry around shopping lists and important phone numbers written on scraps of paper for immediate, regular use. On their desks would be larger paper documents used only when at work. Beyond the desk were shelves and cabinets containing books and files with data used less often. Still further removed were storage boxes in attics, then local and national libraries and archives that required increasing time to visit. Data could be promoted and demoted between these different stores at different times. For example, a book might sit in the library unused for years, then be promoted to your desk for a few weeks when you needed it. Unused documents on your desk could be demoted to a filing cabinet then to the attic.

The same concepts apply to computer memory. When fast and slow versions of the same technology are available, the fast one is better, so it can command a higher price, meaning you can buy less of it compared to the slower one. Given a budget, you can thus trade off speed for capacity. Since most people want some data to be more readily accessible than other data, it makes economic sense to buy and use a mixture of memory types, ranging from small and fast for working data to large and slow for rarely used data. Figure 10-1 shows the approximate speeds and capacities for each of the levels of memory hierarchy that we'll discuss in this chapter.

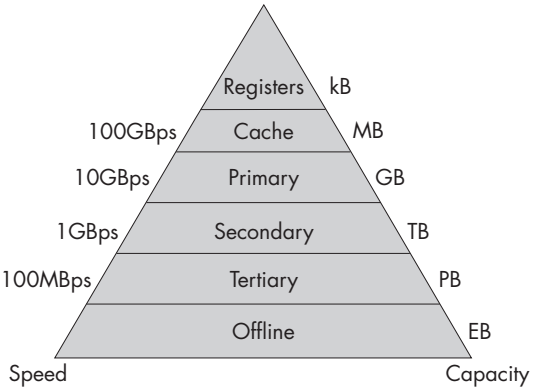


Figure 10-1: The memory hierarchy

These levels can be defined as follows:

Registers Memory inside the CPU, as described in Chapter 7.

Cache Memory outside but close to the CPU, which contains fast copies of primary memory.

Primary memory Memory stored in an address space that is directly accessible by the CPU's load and store instructions.

Secondary memory Memory not directly accessible to the CPU via its registers and address space, but that can be moved into primary memory by I/O to enable such access.

Tertiary memory Memory that isn't directly connected to the address space or to I/O, but that can be mechanically connected to I/O without human intervention.

Offline memory Memory that can be connected only to the computer with human intervention.

According to Church's definition of a computer, any machine that relies on fixed-length addresses—such as the Manchester Baby we built in Chapter 7—isn't quite a computer. A Church computer needs to be able to simulate any other machine, and to do this it needs to be able to ask for and get more storage as needed. Machines based on a CPU and bus with fixed-sized addresses can't easily extend their memory beyond that fixed size, however. To get around this problem, and to allow for unlimited memory, we need to use memory levels below primary memory, such as the secondary and tertiary levels shown in Figure 10-1. These lower levels aren't addressed directly from the CPU, but instead are devices that connect to it through I/O modules.

Primary Memory

Primary memory (aka *system memory*) is memory stored in an address space that's directly accessible by the CPU's load and store instructions. This includes RAM and ROM. Most modern machines use von Neumann architectures; remember, this means that the program and data are stored together in the same primary memory.

In primary memory, each memory location is given a unique address. For example a 16-bit address space has $2^{16} = 65{,}536_{10}$ unique addresses, numbered from 0000_{16} to $FFFF_{16}$. Each address stores a fixed-size array of bits called a *word*. Often, but not always, the word length is chosen to be the same as the address length, such as storing 64-bit words in a 64-bit address space on a modern laptop. You saw a simple way to implement this structure using flip-flops in Chapter 6; you saw how to attach it to a CPU directly in Chapter 7 and indirectly via a bus in Chapter 9.

Bytes and Endianness

Related to the SI versus binary prefix debate is the question of whether to measure memory in bits (b), bytes (B), or words (W). Bits are the most basic unit, and they work well with SI units.

In modern use, a byte means 8 bits, and the term comes from the 8-bit era, when what is now known as a word was by definition 8 bits. One byte was what was stored at one memory address, and what was brought into one register of the CPU for processing. The term *byte* is supposed to suggest the CPU taking the smallest "bite" of memory to process. It was deliberately misspelled to avoid confusion with the term *bit*. "Byte" originally meant *any* such natural CPU size, ranging between 1 and 6 bits in early processors of the 1950s. It only later came to be standardized to mean 8 bits.

In the 8-bit era, it was very natural to measure primary memory in bytes and what are now called kibibytes. You would compute the number of addresses, such as 2^{16} for addresses that are 16 bits long, then append the word *bytes* to this number to get the total addressable memory size. For example, a "64 kibibyte" machine such as the Commodore 64 had 2^{16} addresses containing 1 byte each.

The byte really should have little or no relevance in the modern 64-bit age, in which words are 64 bits rather than 8 bits. If we were to store 64-bit words at each of 2^{32} = 4 gibi addresses, we would talk about having primary memory sizes such as "4 gibiwords."

However, most actual current machines *don't* address memory per word. For historical reasons, they usually continue to address memory per byte, just as they did in the 8-bit era. This is called *byte addressing* and it means that a word on, say, a 32-bit architecture is stored across 4 bytes with separate addresses. Suppose we want to store a 32-bit word such as $12B4A85C_{16}$. We do this using 4 bytes containing 12_{16}, $B4_{16}$, $A8_{16}$, and $5C_{16}$.

A standards war raged for decades over the order in which these bytes should be stored in memory addresses. The ordering is referred to as *endianness*. *Big endians* believe the bytes should be stored in the order (12_{16}, $B4_{16}$, $A8_{16}$, $5C_{16}$) because this looks like the human-readable number $12B4A85C_{16}$. Big endians say this makes life easier and nicer for the humans who see architecture, including architects themselves and assembly programmers.

Little endians, on the other hand, believe the number should be stored as ($5C_{16}$, $A8_{16}$, $B4_{16}$, 12_{16}). This initially seems crazy to most Western people. In particular, if you string the bytes together in this order, you have the nonsensical number $5CA8B412_{16}$ rather than the desired $12B4A85C_{16}$. However, little endians point out that such stringing is based on certain cultural prejudices.

The West uses the Arabic decimal number system, which writes numbers with the highest power on the left and the lowest on the right. It imported this system unchanged from the original Arabic. But Arabic *text* is written and read from right to left, the opposite of Western text. In Arabic, a number string such as "24" is written the same, and has the same value, 24, as in the West, but it's *read* from right to left as "four and twenty." The zeroth column is the units, and the first column is the tens. This makes sense when arithmetic is performed using the number, because almost all arithmetic algorithms begin by operating on the zeroth column and move progressively up the higher-numbered columns. The numbers of these columns match the powers that the base is raised to—for example, the zeroth column is the units, or zeroth power.

The little-endian system assigns numerical addresses so that the zeroth byte is at zero offset from the address of the word, and the nth byte is at an n byte offset. This can make arithmetic easier and faster for the machine in some cases. For example, if the machine is adding two words of different byte lengths (say, a short int plus a long int), it's easy and quick to find the

*n*th byte of each. Similar issues can also arise for words containing instructions of variable lengths: with little endianness, you can always be sure that the opcode is at zero offset rather than having to look for it. Little endianness is now dominant in commercial architectures, so it has effectively won the war.

Memory Modules

RAM and ROM often come in discrete modules that can be added and removed to change the amount of available memory. With a bus architecture, these modules can easily be attached and detached. For example, Figure 10-2 shows one ROM module and two RAM modules on the same bus as a CPU.

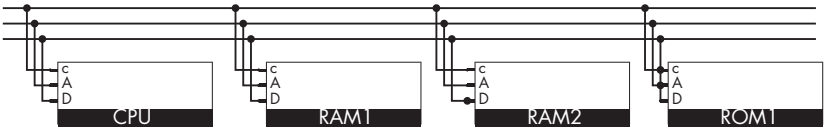


Figure 10-2: A bus architecture including a CPU, two RAM modules, and a ROM module

In general, there could be many modules of both RAM and ROM. All the RAM modules can see the same signals passing along the bus, but each module is configured with a different part of the address space, so only the single module that hosts the specified address will actually respond.

All bus modules—including memory and I/O modules—are usually manufactured to respond to some default address space, such as starting at address 0. However, when they're mounted onto a bus, these addresses need to be remapped to be unique when compared to the other modules. This remapping is done by digital logic components called *memory controllers*, which listen to the bus for global addresses and route them to the appropriate module, converting to the module's own local addresses.

Random-Access Memory

Random access means that any random location in memory can be chosen and accessed quickly, without some regions being faster to access than others. By contrast, something like a cassette tape or punch-card deck isn't random access because it's faster to access data in sequence than to fast-forward or rewind to a far-away location. While RAM stands for "random-access memory," it's a historical misnomer that doesn't paint a full picture. By modern convention, RAM refers to memory that's not only random access but also both readable and writable, as well as *volatile*, meaning its data is lost when the machine is powered off. Many ROMs are also random access, but they aren't considered RAM under the conventional use of the term because they don't fit the other parts of the definition.

HISTORICAL RAMS

We've already discussed Babbage's Analytical Engine RAM, which is still the foundation for RAM architecture today, in Chapter 3. In the Analytical Engine, each memory address corresponds to a stack of gears whose rotations represent a word. One address at a time can be physically connected to the bus. Once connected, any rotation of the gears will be transferred first to the linear motion of the bus, and then to rotation of a register in the CPU, and vice versa. Now let's consider a few other historical examples of RAM.

Acoustic Mercury Delay Line RAM

In "From Combinatorial to Sequential Logic" on page 144, we discussed how the presence and absence of the audio feedback created by an electric guitar and amplifier feedback loop could be used to store 1 bit of information. This was, in fact, exactly how computer memory was implemented in the UNIVAC era, using mercury delay lines, as shown in the following figure.

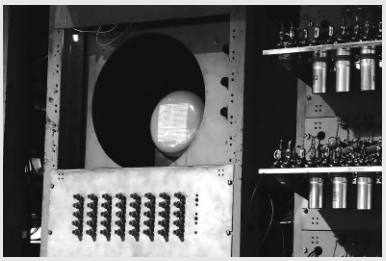


A delay line was literally a microphone and speaker placed some distance apart and used to store a bit of information through feedback. By placing them at two ends of a tube and filling the tube with mercury, the speed of sound is delayed, so the tube can be made shorter than earlier versions using air.

In machines of this era, delay lines could be organized into an address space, as in the Analytical Engine. When the CPU executed a load or store, this would be implemented by making and breaking the electric circuits to connect the required delay line to the bus, disconnecting the others and placing a copy of the data onto the bus for transmission.

Williams Tube RAM

The Manchester Baby was built to research a new type of RAM, known as the Williams tube. The technology, shown below, was conceived in 1946, based on the cathode ray tube (CRT), as found in old TV screens.



As with CRT screens, the Williams tube fires a stream of electrons in a beam, and uses adjustable magnets to deflect the beam to land on one pixel at a time, in a scanning pattern covering a screen. The screen is made from a fluorescent material, meaning that each pixel glows when absorbing the electron beam. Unlike CRT televisions and monitors, the Williams tube's purpose was not as a human-readable display but as actual RAM storage. Pixels retain their charge and color for a short period of time after they're hit by the beam. This means they can be used in a feedback system: we write a screen-full of pixels using the scanning beam, quickly read the screen's state, and pass the data read off the screen back to the scanning beam to be written to the screen again. This refreshes the data on the screen, keeping it alive for as long as we like, rather than allowing the pixels to fade away.

The original Williams tube's screen contained 32 words of 32 bits each, with each row of the screen being one word and each column of the screen being a bit within a word. Thus, the whole system stored 32×32 = 1,024 bits. Phosphor was used as the fluorescent material, which glows green when stuck by the electron beam.

Static RAM

The kind of RAM we saw previously in Figure 6-22, made from flip-flops, is known as *static RAM* or *SRAM* (pronounced "es-ram"). Because SRAM is made from flip-flops (the same structures that are used to make CPU registers), it's fast and expensive. The flip-flops are typically built from around four to six transistors each (depending on the flip-flop type and on how the logic gates are implemented). They have stable memory states, meaning they don't have to be actively refreshed. They're available for reading almost immediately after being written to. What sets SRAM apart from CPU registers is that SRAM is addressed, and CPU registers aren't.

SRAM is typically used to implement caches, as we'll discuss later in the chapter. It isn't usually used for main memory, except in some specialized and expensive machines, such as high-end routers, where main memory access speed is critical. Figure 10-3 shows an SRAM chip.

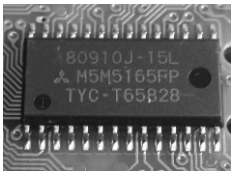


Figure 10-3: An SRAM chip

Cache chips like this may be placed between the CPU and RAM. Alternatively, a similar SRAM cache might be found on the same silicon as the CPU.

Dynamic RAM

Dynamic RAM (DRAM) is cheaper and more compact than SRAM, but slower. Instead of being made from flip-flops, it's made using cheaper and slower capacitors. A *capacitor* is a component for storing electric charge. It consists of two metal plates separated by an insulator. Current can't flow across the plates, but placing a current on them causes them to accumulate charge until they're full of it. Capacitors don't usually appear in CPU design; they're a different kind of electronic component. One bit of DRAM storage is made from just one transistor plus one capacitor. Capacitors can be manufactured on silicon using similar masking processes to transistor manufacture.

As RAM, DRAM features the same addressing system as SRAM, and its circuit diagram has the same overall structure as SRAM, based on words stored at addresses. The difference is that the words are implemented with capacitors instead of flip-flops (Figure 10-4).

DRAM is structured as a 2D array of words or bytes, with each located at a "row" and "column." The requested address is converted (by a memory controller chip) into two smaller addresses per row and per column, which are AND gated together using a single transistor at the combined address. This saves a huge amount of digital logic, but the work needed to split the address into two parts makes DRAM addressing slower than SRAM addressing.

Due to the nature of capacitors, reading the DRAM discharges it and destroys the stored information (as in the Analytical Engine's RAM). Reading and writing the capacitor state is an analog process, which takes time to complete. The charge can also leak away over time, as capacitors are analog devices. To handle these related problems, DRAM must be periodically refreshed, for example, around every 64 milliseconds on a 2018 DRAM. (The need to constantly refresh is the source of the "dynamic" in DRAM.) Like mercury lines and Williams tubes, a refresh reads the current state and then rewrites it a short time later. Refreshing must be timed carefully and may sometimes conflict with and stall a CPU read or write, which then has to wait until the refresh completes before trying again.

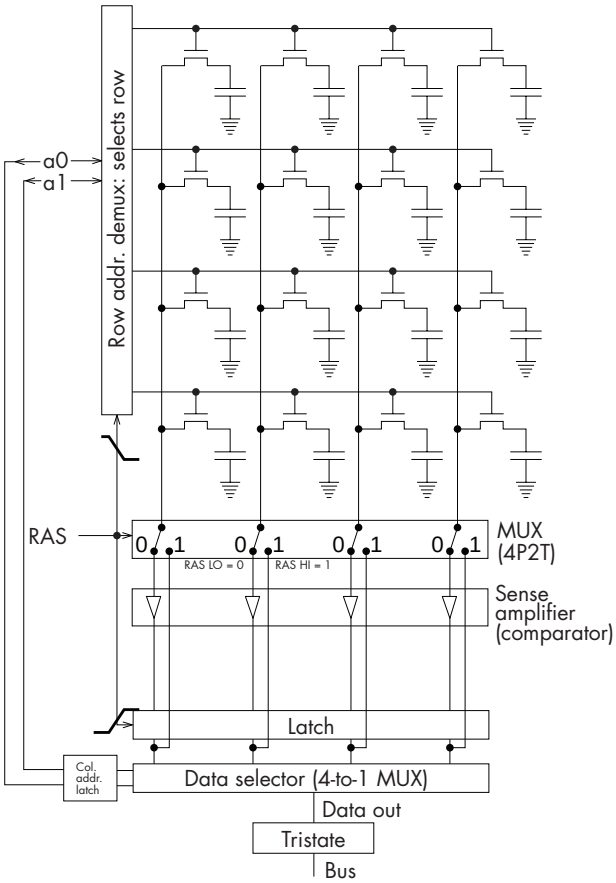


Figure 10-4: A DRAM circuit, showing capacitors and addressing

DRAM benefits from *pre-charging*, roughly a way to "warm it up" just before it's used; this avoids recharging conflicts with access. Hence, modern CPUs and memory controllers work together to try to predict—several instructions in advance—which memory should be "warmed up" before use.

Modern DRAM chips are usually packaged together on printed circuit board modules of around eight chips, each sharing part of an address space, as shown in Figure 10-5. These modules attach to a motherboard via a standard interface, as seen previously in the introduction (Figure 2). Extra memory can be added to a desktop PC by adding more DRAM modules to its memory slots.

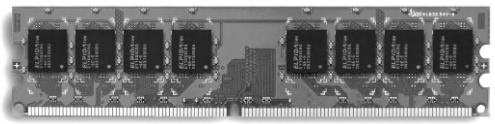


Figure 10-5: A DRAM module

Single in-line memory modules (SIMMs) have a 32-bit bus width, and they were standard in 1990s PCs. Double in-line memory modules (DIMMs) replaced SIMMs in the 2000s. They have a 64-bit bus width, and each stores many gigabytes. Double data rate (DDR) DRAM doubled the speed of DRAM through technology that enables data to transfer on both the rising and falling edges of the clock. This doubles the bandwidth (as *bandwidth = bus width × clock speed × data rate*). SIMMs and DIMMs have gone through several improved standards that can be visually distinguished by the different notch positions, designed so they can be inserted only into the right type of sockets.

Error Correction Code RAM

RAM, like other chips, has become so miniaturized that the component size is getting close to atomic scales. At these scales, quantum effects and particle physics come into play. Quantum effects can include various types of inherent noise and uncertainty about the location of particles used in memory. Cosmic rays are random particles most commonly including electrons, alpha particles, and muons, hurtling at high speed through space from either the sun or elsewhere in the galaxy. If a cosmic ray collides with a sensitive component of RAM, then it can corrupt it and flip its Boolean state.

Error correction code RAM (ECC-RAM) has extra chips on the DIMM that store extra copies or checksums of the data and use them to automatically correct such flips at the hardware level. ECC-RAM is primarily used in space applications where computers are located outside the protection of Earth's atmosphere and so are more exposed to cosmic rays. As its price falls, it may also be found in other high-value, safety-critical systems on the ground.

THE ROWHAMMER VULNERABILITIES

Rowhammer refers to a set of memory hardware vulnerabilities currently affecting computer security. DRAM capacitors are now so small and tightly packed that their electric fields may affect neighboring rows of memory. Security researchers have begun to exploit this effect to read and write memory belonging to target programs. The researchers write new programs and arrange for them to be stored in a region of memory physically next to, for example, the addresses containing your online banking password, owned by the target program. They then load and store data in their own program's locations, in ways that are likely to trigger physical interactions between the capacitors in their own and the target's memory. For example, this could include putting their own addresses into states likely to cause cosmic ray–style errors in the target memory. Or they might be able to infer the state of the target memory by observing similar errors or small time delays in their own reads and writes caused by the target's capacitor states.

Research is currently ongoing into defenses against rowhammer attacks. Approaches include use of ECC-RAM to correct any maliciously induced cosmic ray–style errors, use of higher memory refresh rates, and software-level solutions such as operating system code to randomize the locations of programs in memory and prevent deliverable co-location of code next to targets.

Read-Only Memory

Read-only memory (ROM) traditionally refers to memory chips that can only be read from, not written to, and that are pre-programmed with permanent collections of subroutines by their manufacturer, then mounted at fixed addresses in primary memory. ROMs have since evolved to include other types of memory that don't fit this traditional definition or name very well or at all.

First, the ROM versus RAM distinction has never been a true partition because, as noted earlier, ROM chips are random access, just like RAM: they're mounted in the main address space and accessing any address within them takes the same amount of time. The difference between ROM and RAM is that RAM is readable and writable, while ROM is traditionally only readable.

Second, ROMs have evolved over time to allow increasing ease of rewriting, with programs stored in ROM that are able to be rewritten in some way now known as *firmware*. The following sections describe the main steps of this evolution, as illustrated in Figure 10-6.

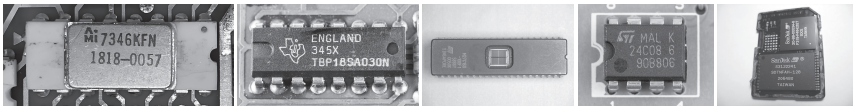


Figure 10-6: Evolution of ROMs: MROM, PROM, EPROM, EEPROM, and SD card–mounted flash. Note that, unusually, the actual silicon is visible in the EPROM package, through a transparent window, which is needed to expose it to light.

Let's go through a few of these types of ROM.

Mask ROM

Mask ROM (MROM) is ROM whose contents are programmed using photolithography by the manufacturer. It remains read-only forever and can't be overwritten. If you want to update an MROM chip, you have to remove it, throw it away, and insert a brand new chip containing the new content. Photolithography is very expensive, so MROMs are difficult to produce and to upgrade.

Programmable ROM

Programmable ROM (PROM) was a great advance over MROM. Similar to the programmable logic arrays (PLAs) discussed in Chapter 5, PROMs are chips manufactured by photolithography to include a generic circuit with many fuses. The programmer can then selectively blow the fuses to create different structures. While PLAs enable arbitrary digital logic networks to be burned in this way, PROMs instead contain a fixed structure of addresses and words, and allow only the bits composing the words to be burned, to make a ROM. Usually each bit contains 1 when its fuse is intact and changes to 0 if its fuse is blown. Like PLAs, PROMs can never be erased once they're programmed.

Erasable Programmable ROM

Erasable programmable ROM (EPROM) is like PROM, but the chip's data can be erased using ultraviolet light. Then new data can be burned on. This cycle can be repeated many times. Although the erasing process was quite complex, requiring that you take the chip out of the computer and put it in a light box, it was still something you, a skilled end-user customer, could do without needing the computer manufacturer.

Electrically Erasable Programmable ROM

Electrically erasable programmable ROM (EEPROM) is like EPROM in that you can wipe the entire chip and rewrite it, but here you only need to use electricity to erase and reprogram. This removes the need to physically manipulate the ROM; it can remain inside the computer. EEPROM is used today in ROMs that allow their firmware to be upgraded. If you've ever done a firmware update, you'll have seen that it can be done entirely in software, without having to physically touch anything. You wouldn't want to be updating firmware every day, but maybe once per year or whenever a bug fix has been found.

Flash Memory

Flash memory is EEPROM that can be erased and rewritten block-wise, meaning you can selectively wipe and rewrite just one small part, or block, of the memory at a time. This way you can leave most of the ROM intact, unlike with regular EEPROM, where you have to wipe and rewrite an entire chip of ROM at a time, as in a firmware update. Flash memory makes it much easier to rewrite portions of ROM frequently, while the chip is online, making it more feasible for day-to-day storage, functioning almost like RAM in some cases.

Caches

A *cache* is an extra layer in the memory pyramid between the fast registers of the CPU and the slower RAM. It stores copies of the most heavily used memory contents, making them available for quick retrieval. (*Cache* is an archaic word for a store of items such as food, weapons, or pirate treasure.) Without a cache, RAM would connect straight to the CPU, either directly, as discussed in Chapter 7, or using a bus with control (C), address (A), and data (D) lines, as discussed in Chapter 9 and summarized in Figure 10-7.

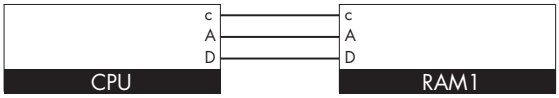


Figure 10-7: A basic CPU, bus, and RAM architecture

The problem with this kind of cacheless architecture is that most programs need to access RAM frequently, but the capacitors that implement DRAM are slower than the flip-flops that implement the CPU's registers.

RAM thus becomes a major bottleneck for system speed. It's no use having a fast, gigahertz CPU if the RAM is running orders of magnitude slower and the CPU has to wait around for each load and store to complete. Adding an SRAM-based cache made from flip-flops between the CPU and RAM, as shown in Figure 10-8, helps avoid these bottlenecks.



Figure 10-8: A basic CPU, bus, and RAM architecture with a cache in between

When the CPU needs to load some data, the cache checks if it has it, and returns it quickly if so. If not, the cache refers to the next memory level down (in Figure 10-8, RAM) and fetches the data from that level. Caching can also occur at *all* levels of the memory hierarchy, from registers to hard disks and jukeboxes (more on the latter in the "Tertiary Memory" section). However, it's most commonly considered at the primary memory level, as we're discussing here, between the registers and the main DRAM memory.

Initial designs began with a single cache, made from SRAM. More recent machines have made use of Moore's law for transistor density to fill silicon with larger caches and more levels of cache. It's common today to have at least three cache levels, called L1, L2, and L3, as in Figure 10-9.



Figure 10-9: A basic CPU, bus, and RAM architecture with L1, L2, and L3 caches

All these cache layers between CPU and DRAM memory are typically made in SRAM, but they have different operations policies that trade off size and speed in their different digital logic implementations. Historically, caches lived on dedicated chips outside the CPU. While lower levels still do this, a major trend is to move bigger and higher cache levels onto the CPU silicon itself.

Understanding the caches of your machines helps you write faster programs. Typically, each level of cache is 10 times faster than the one below it, so when you fill a level you'll see a sudden slowdown in memory access. If you know the cache sizes, you can redesign your code to keep data in use within known cache-level limits to benefit from their speed.

Cache Concepts

Caches are based on the *principle of locality*, which states that only a small amount of memory space is being accessed at any given time, and values in that space are being accessed repeatedly. It's therefore useful to copy recently accessed values and their neighbors from larger, slower memory to smaller, faster memory. There are several different ways to think about "neighbors" and "locality." *Temporal locality* is the property that values tend

to be accessed repeatedly at nearby times. *Sequential locality* is the property that some sequences tend to be re-accessed in the same order multiple times. *Spatial locality* is the property that values nearby in memory tend to be accessed together. These concepts apply to both instructions and data, often arising due to loops and subroutines.

Cache memory is made of many *cache lines*. Each line contains a *block* with copies of several contiguous words from memory, as well as a *tag*, an address or other identifier describing which memory location has been copied into the block. Each line also has a *dirty bit* that tracks whether the CPU has changed the value in the cache, making it different from the equivalent value in memory. Table 10-1 shows a few example cache lines.

Table 10-1: Cache Lines

Tag	Block	Dirty bit
$08F4	01101100 01101100 10011010	1
$2AD5	10010101 11100110 00110110	0

Each cache line shown in the table has a block of three 8-bit words, a tag consisting of the full address from a 16-bit address space, and a dirty bit. The 1 dirty bit for the first line indicates it's been updated, while the 0 dirty bit for the second line indicates it hasn't.

We don't cache individual addresses, but rather lines because it's very cheap to move around larger chunks of memory rather than individual words. By bringing in whole lines around a target word, we exploit spatial locality—data and programs in neighboring locations are likely to be used next. The line prepares for this.

Some cache systems use "hash functions" to choose a location in the cache for storing a piece of data, usually based on the data's address in lower-level memory. A *hash function* is a many-to-one function that maps a big input number to a smaller output number, the *hash value*. It's not usually possible to recover the original value from the hash value. For example, a function that takes the last two hex digits of a hex number is a simple hash function: $hash(9A8E_{16}) = 8E_{16}$. The function that performs a Boolean AND of all binary digits in a number is another hash function: $hash(01101001_2) = 0\&1\&1\&0\&1\&0\&0 = 0$. A commonly used hash function for caches is to compute the value of an address modulo the number of available lines in the cache.

Finding an item in a cache is known as a *hit*. Not finding an item in a cache is known as a *miss*. When a miss occurs, we have to go back to the underlying memory and find the item there instead, usually making a new copy in the cache for future use. The *hit rate* is the ratio of hits to attempts (hits and misses together). This measures the proportion of cache lookups that are successful. The *miss rate* is the ratio of misses to attempts. This measures the proportion of cache lookups that are unsuccessful. The *hit time* is the time required to access requested data if a hit has occurred, and the *miss penalty* is the time required to process a miss.

A cache has only a limited number of lines, and they quickly fill up as we store cached copies of everything that we access from the underlying memory. Once the cache is full, we'll continue to request new addresses. These will initially miss, but temporal locality suggests that these new addresses are more likely to be reused than the older ones in the cache. We should therefore choose lines in the cache to overwrite, discarding their previously cached addresses and replacing them with the new ones. The contents of the overwritten lines are called *victims*.

Once we have a cache structure, we need algorithms, implemented in fast digital logic, to manage it. We need to decide how to best make use of the available lines, and how to create and look up tags. As with most digital logic design, there will be trade-offs between methods that are simple and methods that are fast. The latter tend to require more silicon, making them more complex, error-prone, and expensive. Let's take a look at a few options for using caches.

Cache Read Policies

Reading from a cache is a simpler task than writing to it, so we'll first study some options for cache read algorithms.

Direct Mapped

Direct mapping is the simplest, easiest, and cheapest cache read policy to implement and understand. It's sketched out in Figure 10-10.

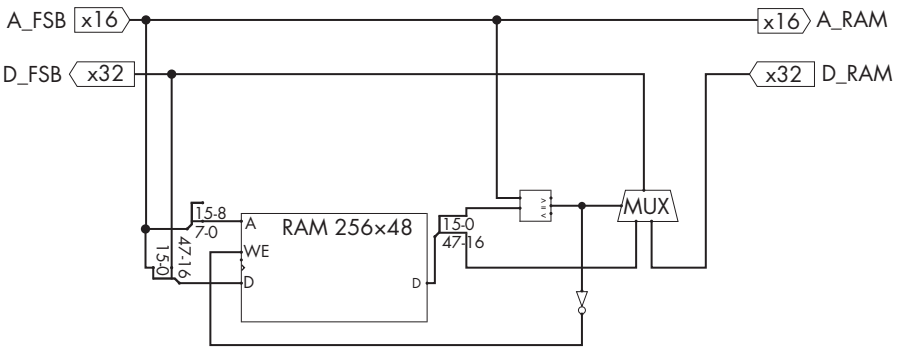


Figure 10-10: A direct mapping cache read policy (showing lookup and caching)

In essence, the line where we store or look for a tag is addressed using a fixed hash of the tag. A line with this tag will only ever be stored at a single location. If multiple lines compete for the location, the new one will replace the older one. For example, suppose we load from address $67AB_{16}$. We might compute $hash(67AB_{16}) = 4_{16}$, which means that this address and its contents will be cached in line 4_{16}, victimizing anything that was previously on this line.

The drawback is that direct mapping can't keep multiple in-use addresses in cache if they share the same hash. Suppose our program has a tight loop that reads and writes the two alternating addresses $67AB_{16}$ and $12C9_{16}$

many times. The problem here is that $hash(67AB_{16}) = hash(12C0_{16}) = 4_{16}$. Both addresses will continually fight and victimize one another, overwriting line 4_{16}, even if no other addresses or cache lines are being used in the loop at all. In such a case, the cache will give no benefit at all, as every attempt will miss.

Fully Associative

To fix the problem with direct mapping, we'd like to have addresses use different cache lines depending on how in-use our lines are, so that we victimize lines that are the least used, as sketched out in the *fully associative cache* of Figure 10-11.

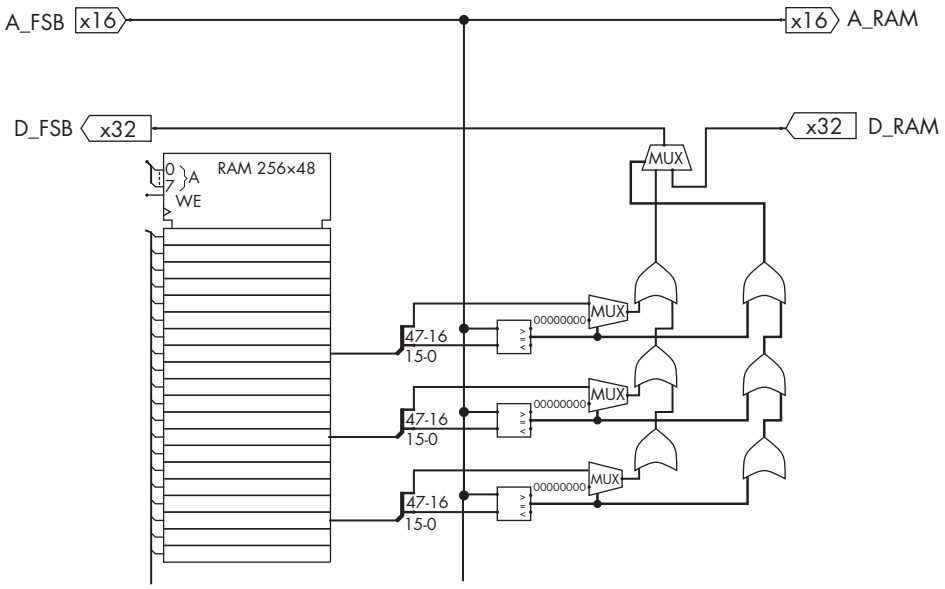


Figure 10-11: A fully associative cache sketch

Here, each line of cache RAM is given its own digital logic block, including a comparator, multiplexer, and OR arrays. Only three such blocks are shown for illustration purposes, but for a 256-line cache, for example, there would be 256 such blocks, all running in parallel.

We want to be able to store a tag, block, and dirty bit on *any* available line and be able to find it quickly. Caching is the easy part here: we just create some digital logic to count how much use each line is getting and to pick out the line with the lowest count.

The cache lookup is the harder part. In direct mapping, we just computed the same hash function as we used for caching, to tell us at which line to find a desired address. Now it could be anywhere in the cache, so we need to add lots of extra digital logic to check each of the lines' tags for a match with the desired one and activate the matching line if it exists. Doing this in parallel (which is the only realistic way to make this fast enough to be useful)

requires *N* copies of this matching digital logic, one for each of the *N* lines of cache, making it a much larger and more energy-consuming beast.

Set Associative

Set associative cache reading is an attempt to get the best of both of the above methods. Here we partition the *N*-line cache into several smaller sets of lines. We use hashing on addresses to hash to a set number, rather than a line number. During caching we find the set number from this hash, similar to the direct mapping approach, then choose as the victim the line within this set that has the least usage, similar to the fully associative approach. During lookup we again find the set number from the hash, then we use parallel matching checks on all items in just the one set to quickly find the matching line.

This approach means we only have to activate the comparators within a single set, rather than the entire cache, but we still avoid the direct-mapped problem of tight loops sharing hash values. In practice, this is often found to be a nice balance.

Cache Write Policies

Caches become a bit more complicated when we do stores because a store changes the state of the memory. Suppose we've recently loaded an integer 17 from address $540A_{16}$ and cached a copy during the load. We want to increment this integer to 18 and store the result back at $540A_{16}$. Due to the locality principles, it's likely that we'll continue to both load and store from $540A_{16}$ in the near future, so rather than store 18 directly in $540A_{16}$, it may be faster to store it only in the cache line that's currently caching $540A_{16}$. This means that all the future loads and stores can just hit the cache and don't need to go to main memory.

The problem is that eventually this line will be victimized and we'll lose all the changes we've made to the value; the main memory still contains the old value of 17. To avoid this, at some point we need to copy the modified value back to main memory. The dirty bit shown earlier in Table 10-1 tracks whether this needs doing. It's set to 0 if the value in the line is the same as the value in memory, or to 1 if the value in the line has been updated but the value in memory hasn't. Algorithms called *cache write policies* use this dirty bit to manage the copying back to memory. Let's look at two different approaches: write-back and write-through.

Write-Back

Write-back is the simpler cache writing method: it copies the contents of the cache block back to RAM only when the line is victimized. This is relatively slow, however, because victimization occurs only when an instruction is in a rush to get executed. We get told to start writing back only once the victimization has been announced, and the victimizing instruction will now have to wait for us to do a slow RAM access before it can overwrite our victim line.

Write-Through

Write-through is a potentially faster alternative to write-back, although it uses more resources. In write-through, we don't wait until our line is victimized to copy our line's block back to RAM; rather, we do it multiple times, continually, in the background, using digital logic attached to the cache line and bus. This logic acts similarly to an application like SyncThing or Dropbox, continually looking out for any changes in the cached version and copying them back to the main version in RAM. This doesn't create extra work for the CPU, as the extra digital logic is located on the cache itself. It does, however, lead to more traffic on the bus, as we're sending these updates many more times than with the write-back approach.

Advanced Cache Architectures

Consider how caches should interact with the advanced CPU developments of Chapter 8. Pipelined CPUs need to care a lot about cache misses, as they form another possible hazard. An efficient pipeline may be timed to assume that memory accesses will be cached, and if there's a miss they'll need to stall or otherwise handle this hazard.

You saw in Chapter 8 how branch prediction attempts to guess the flow of a program to enable pipelines and out-of-order execution to go more smoothly. This can be used in conjunction with caching to *preemptively* fetch and store data—that is, before the actual load and store instructions are reached. These instructions take much longer to execute than in-CPU operations, so it's useful to initiate them early. CPUs can look ahead in the program to try to guess which parts of main memory are likely to be needed many instructions down the line, and start caching them in advance so the CPU fetches will be faster.

As mentioned, each layer of the cache—L1, L2, and L3—provides roughly a tenfold speedup over the layer below it, so the potential gain from preemptively moving data higher up in the memory hierarchy isn't trivial. The caches can always be rolled back and the CPU stalled if preemption gets it wrong. It's not the end of the world if we bring the wrong data into the cache: the cache is a big place, and it's okay to change what's in it.

Due to the row-column structure of DRAM addressing, it's faster to read multiple items in a single DRAM row all at once rather than individually. (Once a row is activated, it's almost free to read many columns versus a single one.) Hence, modern DRAM controllers will typically work in harmony with the cache to move large DRAM rows into cache lines.

Cache writes can unnecessarily slow down a system if we know in advance that the data won't need to be read again soon. In this case, writing to the cache and then transferring to main memory can be slower than just writing directly to main memory. Modern CPUs may provide special instructions for cacheless writing, which canny programmers and compiler writers can use to make programs faster.

It's been found empirically that L1 caches work more smoothly if they're split into two separate, parallel caches, one for instructions and one for data.

This can occur in Harvard architectures, where instructions and data are already separated in RAM, but also in von Neumann architectures, where instructions and data can be distinguished by which part of the CU is requesting them (instructions are requested during the fetch stage, while data is requested during the execute stage). This separation occurs only at L1, with lower cache levels sharing instructions and data, as in Figure 10-12.

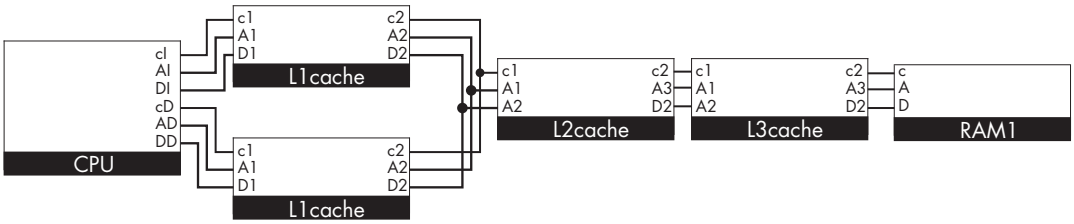


Figure 10-12: A basic CPU, bus, and RAM architecture with separate L1 caches for instructions and data, and shared L2 and L3 caches

Separating the instructions and data at the L1 level appears to be effective because both data and programs exhibit spatial locality individually, but with little locality between them. Also, instructions aren't usually overwritten, while data often is, so separating out the instructions can simplify the cache write process.

Secondary and Offline Memory

Secondary memory is memory that can quickly be brought into addressed memory space via I/O. Data items in secondary memory don't have addresses in the primary memory address space. Rather, they're accessed via I/O, usually via an I/O module that *does* sit in the primary address space and relays requests to the secondary storage. Secondary storage is sometimes called *online storage* to emphasize that it's powered, active, and available whenever the computer is on.

Offline memory is that which can't automatically be loaded into primary memory without *manual* human interventions. Often this includes secondary memory media that are physically ejectable and replaceable, such as tapes, discs, and USB devices. These media are secondary memory when connected to the computer, and offline memory when disconnected. Offline memory is typically used for backup and archival purposes, as well as for transportation. The fastest way to move petabytes of data around the world is still to put it on a truck as offline memory and drive it to its destination.

Secondary and offline memory should really nowadays be measured in bits and SI units—for example, describing an "8.8 terabit hard disk" instead of a "1 tebibyte hard disk." This is because they aren't part of primary memory address space and so aren't addressed using primary memory's word or byte addresses. The concept of bytes is even less relevant here than in modern primary memory. However, as primary memory is still often byte-addressed and measured in bytes, most people still have a better feel for sizes

in bytes rather than bits, so they choose to measure secondary memory in the same units.

Secondary (and offline) memory is usually characterized by requiring some mechanical motion to look up data, rather than being random access. This includes scrolling through tape or spinning discs made from various materials. We'll look at some details of these technologies next.

Tapes

Tapes are one-dimensional data stores that must be scrolled left or right to locate a required datum. You can think of human-written paper scrolls, like the Torah, as the original tapes. Tapes aren't random access because a reading device has a position at one point in the tape, and it takes longer to move the tape (or the reader) to access a far-away location than a nearby location. Fast algorithms using tape storage need to take this structure into account and optimize memory access to reduce large address jumps.

Punch Cards

Punch cards are the original computational secondary storage, as used in the Jacquard loom and Analytical Engine (seen in Figure 1-11). They continued to be used in IBM Hollerith machines, and were used to store and read programs for early electronic machines of the 1960s. Occasional industrial use continued even into the 1980s, and allegedly at least one UK council may still be using them today. In punch cards, binary digits of data are represented by the presence or absence of holes punched or not punched at a series of physical locations on a card or piece of paper. The holes are usually about the size made by the desktop hole punchers you buy to file your paper documents into ring binders.

Cards are 2D, having rows and columns. Typically each row stores one word, with their row numbers acting as addresses (in a secondary address space, not primary RAM addresses). Conceptually, and sometimes physically, decks of cards are *chained* together to make what is really a 2D tape.

Punched Tape

Punched tape is an alternative to punch cards. Such tapes were used by the British Post Office, formed the inspiration for the Turing Machine, and were also used in the Colossus, as seen in Figure 1-22. Depending on your point of view, tape is conceptually simpler than cards because it's just a single 1D row of bits; or it's more complex than cards because you have to worry more about aligning and reading words, which on cards are easily presented as rows.

Magnetic Tape

Magnetic tape was developed in the 1920s for analog audio recording in studios, commercialized for home use as 8-track systems in the 1960s, then used widely in 4-track compact cassettes during the 1980s. Analog magnetic tape was also widely used in the 1980s for home video recordings, following

one of the first modern data standards wars between competing VHS and Betamax formats.

In these systems, a magnetizable material such as iron oxide is formed into a tape structure, and the level of magnetization at each point along the tape is used to store data. Unlike punched paper, magnetic tape is easy to remagnetize and can be rewritten many times.

The same magnetic tapes can be used to store digital information, in various ways. For example, 0s and 1s can be encoded as single cycles of two different audible frequencies—a method that's resilient to the heavy noise added by most tape devices. Algorithms developed for optimal access of punched tape carried over directly to magnetic tapes, as in the 1980s machine of Figure 10-13.



Figure 10-13: A 1980s compact cassette and player/recorder, used for both analog music and digital file storage

Magnetic tape is still in use today for offline storage, specifically for weekly or daily backups of company systems. Tape is cheap and cost-effective for large-scale storage, where access time is less important. Tapes are thus useful for the daily backup task because you want to have lots of old backups kept around for as long as possible. In particular, if someone attacks your company in a more subtle way than just deleting everything—for example, by making a series of small changes to your database—it's useful to have a long series of backups so you can recover the state of the system from different days, weeks, months, years, and even decades. You can buy a new tape for a few dollars every day to get this assurance. Having many tapes around also means they can be kept at many more locations than can hard drives—for example, with a different employee taking one tape home each day so that even if half the staff's houses burn down on the same day, you still have many recent backups around.

The most popular current standard for magnetic tape storage is *Linear Tape Open (LTO)*, shown in Figure 10-14.



Figure 10-14: An IBM Ultrium Linear Tape Open cartridge and drive

LTO is an open source standard that, as of 2020, stored around 36TB on about 1 km of tape in one cartridge that fits in your pocket and takes around 12 hours to write. This is a good size and time for most small businesses; they can back up the whole system overnight onto a single cartridge.

Disks

Audio recording began in the 1870s with wax cylinders, as shown in Figure 10-15.



Figure 10-15: A wax cylinder audio storage device

Here, sound waves enter the acoustic horn and are concentrated to vibrate a needle, etching the sound wave into a spiral around a hot wax cylinder as it rotates and is slowly moved left to right. When the wax cylinder is cool it can then be spun past the needle again to make it vibrate in the same ways, and have its motions amplified by the horn, replaying the sound.

Wax cylinders were used commercially until 1898, when they were replaced by gramophones with discs, rotating at 78 revolutions per minute (Figure 10-16, left). These "78" disks used the same idea of etching the analog sound wave directly into their spiral grooves, and their vinyl descendants—now with electrical amplification—are still in use by DJs today (Figure 10-16, right).



Figure 10-16: A gramophone (left) and a modern Technics SL-1200 turntable (right)

Unlike audio discs, which have a single track spiraling in from the edge to the center, most data disks are truly 2D, as they have many independent *tracks*, each at a fixed radius, as shown in Figure 10-17.

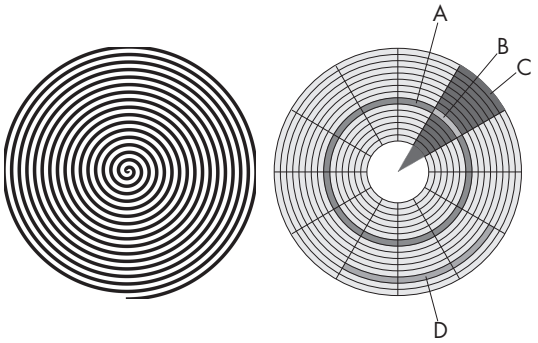


Figure 10-17: The single track of an audio disc (left) and the 2D track of a data disk (right). The latter shows a track (A), sector (B), geometric sector (C), and cluster (D).

Tracks near the edge are larger than those in the center, so they store more data. Tracks are divided into fixed-data-size *sectors* around their circumference. Each sector has an address composed of its track ID and location within the track. In most systems, sectors store their own location in some of their bits so that we can figure out which part of the disk we're looking at. They may also store redundant bits, which compensate for physical damage to the disk, using Shannon's theory of communication. Sectors may be grouped into contiguous *clusters*, which are the smallest unit that can be read or written together.

Data on disks can be accessed in an *almost* random-access manner: individual sectors can be stored or retrieved in any order, not only sequentially, but reads and writes to nearby sectors and tracks will be faster due to the motion of the disk and head. It's easy and fast to read from a series of sectors in order around the same track as they spin past the head. If you want data on the same track but at a different angle from the current sector, you have to wait for the disk to spin around to bring that sector under your head. If you want data from a different track, you have to move your head along

the radius, which is very slow, as it's a physical device. I/O modules controlling spinning disks thus need to consider the *access time*—the time it takes to read or write one sector. Access time is composed of two main factors: *seek time* is the time it takes for the arm to position itself over the track, and *rotational delay* is the time it takes for the desired sector to position itself under the head.

Floppy Disks

Magnetic disks use the same technology as magnetic tape to represent data, but they arrange the magnetizable material into a 2D disk rather than a 1D tape. The disk is read and written by a magnetic head on an arm, like a gramophone needle. *Floppy disks* (Figure 10-18) first appeared in the 1960s. They're so-called because they physically flex.



Figure 10-18: Three generations of floppy disks: 8 inch (1970s), 5 1/4 inch (1980s), and 3 1/2 inch (1990s)

Floppy disks are vulnerable to damage, so they're usually encased in a plastic sheath, as in the figure.

Hard Disks

Hard disks are made of nonflexible materials. They can store higher information densities and spin faster than floppies. These devices usually require sealing the head into a package with the disk, as in Figure 10-19, rather than allowing removable disks, as with floppies.



Figure 10-19: The inside of a magnetic hard drive

Hard *drives* usually contain multiple hard disks packaged together, each with its own head, with a single address space spanning all of them. This can help reduce access times, because the heads can all read and write together. The disks spin at speeds such as 90 to 250 Hz, which causes a layer of air to lift the head off the surface, so the head doesn't physically contact the platter. This means there's no physical wear to the head or the disk. Designers have invested heavily in technology to automatically and rapidly park the head if the unit is in physical danger, such as being struck or pushed. Without this, the head would crash into the disk and destroy it during such an incident.

Optical Discs

Optical discs are modern-day version of the Babylonian clay tablets seen in Figure 1-5. Like those tablets, they're solid objects with small cavities—known as pits—made in them to represent data, as shown in Figure 10-20. Like punch cards, they use binary encoding, so each location either contains a pit or doesn't contain a pit. The pits are read using a laser, and their nanometer scales are comparable with the wavelengths of this laser light.

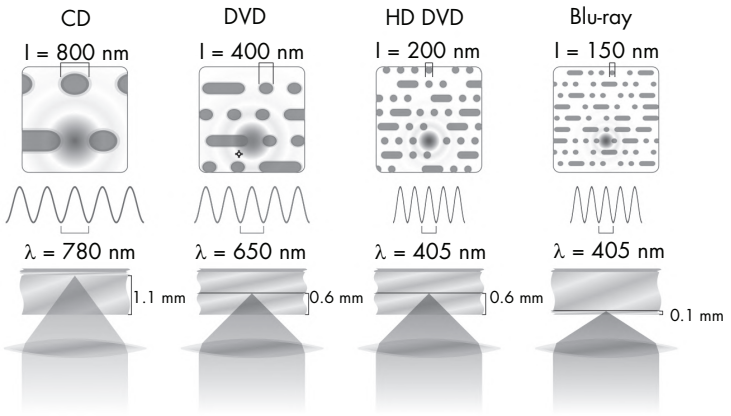


Figure 10-20: Four generations of optical storage

LaserDisc (1978) was the first optical disc, having a 12-inch diameter like a vinyl album and marketed for home video. *Compact discs*, or *CDs* (1982), used roughly 800 nm pits, read by a laser head, to store up to 700 MB of audio data. CDs started seeing use for general rather than audio data storage in 1988 with the *CD-ROM* specification. Like CDs, these became read-only after initially creating the pits on their surfaces. *CD-R* was a version that simplified the recording process, allowing home users to "burn" their own CD-ROMs, again only once. These were used in the late 1990s for copying audio music collections, first using CD audio representations and then using bulk MP3 storage. They were usually blue on the burnable side and gold on top. Their "burning" was a physical process involving lasers and heat; this is the origin of modern slang "burning" now used for writing to other types of

ROM, such as flash or FPGA. *CD-RW* was an improved CD-ROM that could be rewritten several times.

Digital Versatile Disc (DVD) (1995), was an order of magnitude improvement, reducing pit size to 400 nm to achieve disc capacity of up to 4.7GB using the same size physical disc as CDs. DVDs were initially used for video but soon also for general data. As with CDs, write-once DVD-R and rewritable DVD-RW were also developed. *Blu-ray* (like its short-lived competitor, HD-DVD) reduced the pit size again, this time to 150 nm, allowing storage up to 25GB on the same size disc. As these pits are smaller, they require shorter-wavelength blue rather than infrared or red laser light to read them, hence the name.

Solid-State Drives

For secondary storage, most current computers have moved from hard drives to *solid-state drives (SSDs)*. These are manufactured to have the same form factors and I/O interfaces, and similar capacities, as hard drives, but with no moving parts. This makes them faster, more reliable, lower power, quieter, smaller, and less prone to breakage when dropped. As there are no moving parts, they can be truly random access. SSDs are flash memory, as we've previously reviewed.

The same flash memory technology is also used as offline storage, where SSD drives are easily removable, such as when connected to I/O via USB (known as USB sticks) or SD (known as SD cards).

Tertiary Memory

Tertiary memory is a recently proposed level in the memory hierarchy. It lies below secondary memory but above offline memory, and has been created to describe memories that used to be offline—requiring humans to physically load and eject media such as discs and tapes—but is now automated by mechanical processes. For example, automated Blu-ray and LTO tape jukeboxes as in Figure 10-21 form tertiary memory.



Figure 10-21: A robotic tape jukebox in a data center

In the figure, a robot arm is used—as in 1950s vinyl record jukeboxes—to pick up tapes and place them into the reader and storage containers.

Similar robotic systems can be built around Blu-ray discs. Mobile robots driving baskets of hard disks around can now also be considered tertiary memory.

Data Centers

When you put thousands, or tens or hundreds of thousands, of secondary and tertiary memories together in a warehouse-sized building, you get a *data center*. Search engines, social networks, online retailers, media streamers, and governments all now need to store and access data at this scale. A typical data center will contain many different layers of the lower levels of the memory hierarchy. For example, tapes take longer to fast-forward and rewind than disks, so these are more likely to be found as long-term backup systems than serving the latest social media posts. Once you access something from a slower backup system, it will then be cached somewhere higher up the memory hierarchy, such as on an SSD drive, making for faster retrieval next time.

Data centers may be built with extreme security and resilience in mind. For example, HSBC's literal "data mine" is widely believed to store backups of all its global financial data in a former UK coal mine. You can tell it's a data center because there are huge air ducts rising out of the ground to disperse all the heat from the computers. The mine is thought to be robust to nuclear, chemical, and biological attack. In the event of a nuclear war, the rest of humanity may be bombed back to computing with Ishango bones, but the bank will still be able to come after your mortgage repayments.

Summary

Memory architecture is driven by economics: you can buy big, slow, cheap memory; small, fast, expensive memory; or some mixture of both. Empirically, most programs show spatial, sequential, and temporal locality, in which different small parts of memory tend to be in heavy, repeated use at different times. Memory architectures are thus designed in hierarchies that fit both the economics and usage patterns, including caches between layers to promote currently in-use memory to higher levels. Primary memory is that which is addressed directly by the CPU, using the bus, while secondary memory is connected via I/O. Secondary memory often takes the form of spinning disks, which can be disconnected and replaced, becoming offline memory if humans are involved or tertiary memory if the process is automated by robotics.

Exercises

Your Computer's Memory

1. Try to find the sizes and speeds for each type of memory in your own computer, including caches, RAM, and secondary storage. If you can open up your computer, look inside, locate them, and

find their makes and model numbers, then look up their datasheets online. Most operating systems have utilities that will display useful information about their memory; for example, Linux will show caches with `lscpu` or `cat /proc/cpuinfo`, RAM with `free -h`, and secondary memory with `lsblk`.

Building a Static RAM in LogiSim

1. Build the static random-access memory (SRAM) presented in Figure 6-22 in LogiSim. It should be able to store and read 2-bit words at the four memory locations.
2. Extend your LogiSim SRAM to have longer words and more addresses.

Challenging

1. Make four copies of your SRAM, representing multiple RAM chips. Each one will have the same address space, starting from address zero. Design a memory controller module that converts addresses from a larger global address space—having two extra bits—to sections of particular RAM chips and these local addresses within them.
2. Try attaching this system to the Manchester Baby model in place of its previous LogiSim RAM.

More Challenging

1. Design and build a direct-mapped cache in LogiSim and link it to your LogiSim RAM from the previous task. (This won't speed up that RAM, as it's already fast SRAM, but it could then enable that SRAM to be replaced by a larger and cheaper, but slower, DRAM.)
2. Try to build the other types of cache too, if you're feeling brave. Use the sketches provided in this chapter as starting points.

Further Reading

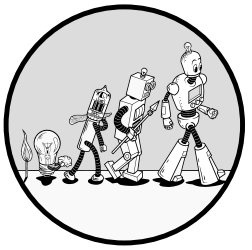
For a definitive recent classic on memory, see U. Drepper, "What Every Programmer Should Know About Memory," November 21, 2007, *https://people.freebsd.org/~lstewart/articles/cpumemory.pdf*. In fact, this resource contains far more than any normal human should know about memory.

PART III

EXAMPLE ARCHITECTURES

11

RETRO ARCHITECTURES



Now that you've made it through the theory, let's have some fun. Part III will consolidate your theoretical knowledge by programming on a series of real, emulated architectures. It's possible to skip around these chapters depending on your interests, but they present systems roughly in order of their complexity and history, so seeing and programming the earlier systems may help you understand the later ones.

We previously studied the Analytical Engine and Manchester Baby, and in this chapter we'll progress to 1980s 8-bit and then 16-bit systems. Modern embedded systems are somewhat similar to these retro systems, so we'll work with them in the next chapter. We'll then look at 1990s desktop PCs, followed by modern smart and parallel architectures. At each step we'll introduce new features that have stuck around into modern designs.

The basic structure of the classical CPU didn't change very much from 1836 to 1990. The design served us well, from the Analytical Engine through what's now known as the 1980s golden age of architectures. In this chapter, we'll look at two designs from this golden age: the famous 8-bit 6502, as used in the Commodore 64, Nintendo Entertainment System (NES), and

BBC Micro; and the 16-bit 68000, which defined the 16-bit generation of machines, including the Commodore Amiga and Sega Megadrive. We'll study these as relatively simple examples of classical CPUs, before things got complicated. These examples should help you to consolidate what you've learned in the previous chapters, so refer back to them if you need to look anything up as you go.

Programming in the 1980s Golden Age

Programming in the 1980s was dominated by architecture. The 1980s hardware market was highly heterogeneous, with many competing companies designing and producing different, incompatible machines. Figure 11-1 shows just a few of the different machines that came out over the course of the decade.

Instead of downloading apps, you could buy magazines full of printed assembly code that you would type out to run simple games and applications. Without modern operating systems, this code could read and write the machine's entire memory space, so you could see exactly what was going on in your machine and be at one with its architecture.

Computer design companies such as Commodore could produce their own custom ROMs at much lower cost using programmable ROMS or PLAs than if they had to do their own photolithography, and these technologies were a major enabler of the multitude of home computer systems. In today's language, these ROMs were basic input-output systems (BIOSes), collections of subroutines that, for example, print ASCII text to the screen; draw points, lines, and triangles; and make sounds. The programmer could also perform these tasks directly via I/O—that is, by loading and storing directly to I/O module addresses—but subroutines were provided for convenience to automate the process. You would call a subroutine on a ROM chip by putting the necessary arguments into CPU registers and then doing a jump to the subroutine address in the ROM.

ROM and RAM were equally important, and they worked together. RAM was a scarce resource for user data and user programs that made many calls to the subroutines in ROM. In addition to knowing ROM subroutine addresses by heart, programmers and communities often had conventions for favorite regions of RAM to use for different tasks, so they would generally know their way around the memory map of their whole computer.

Because of these conventions, users had much more direct access to their computers. The number of addresses was quite small: 32,768 (32 k_2B) or 65,536 (64 k_2B), and this meant you could find where variables like the number of lives in a game were stored, and then go inside the memory to edit them. Directly overwriting memory like this was called a *poke*, and successful pokes were collected onto *cheat disks* and passed around to modify games.

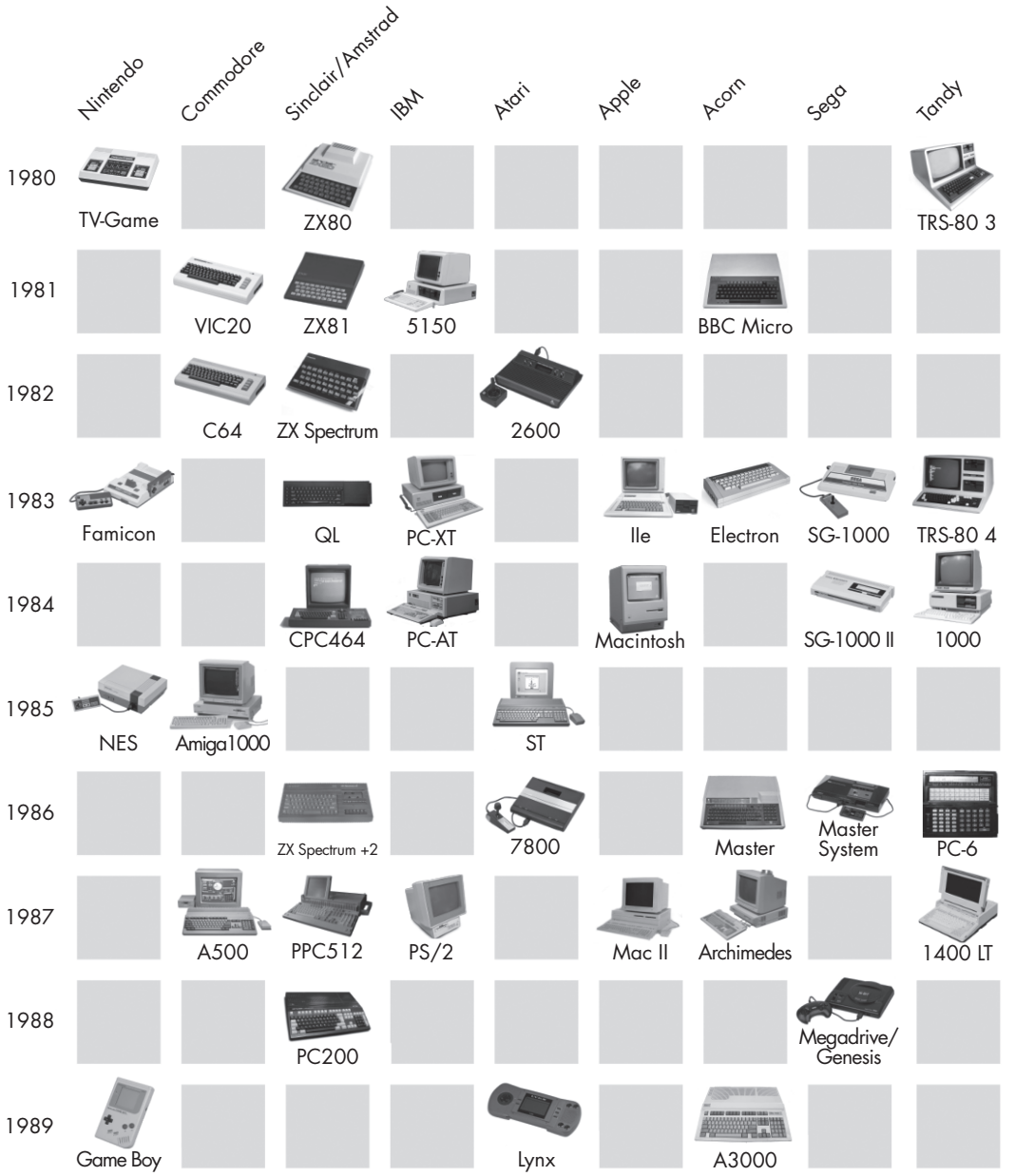


Figure 11-1: Different computers from the 1980s golden age

8-Bit Era

The early 1980s was the 8-bit era: this was the time of the Commodore 64 and Atari 2600, games consoles like the Sega Master System and Nintendo NES, and the British machines BBC Micro and ZX Spectrum.

Some of these machines shared some subcomponents; for example, the 6502 was used in both the Commodore 64 and BBC Micro, and the Spectrum's Z80 chip could be added to the BBC Micro as a second processor, so you could have friends sharing programs at this level. But the machines would have different graphics and sound chips containing different functions at different addresses that weren't compatible, and typically each machine would have its own friends, user groups, and magazines form around it.

Computer graphics and music of this era looked (see Figure 11-2) and sounded *like computers* because they reflected their architecture, creating a computer culture that has been lost today. You could actually feel the 8-bit-ness of an 8-bit game in a way that you don't see 64-bit-ness in contemporary games.

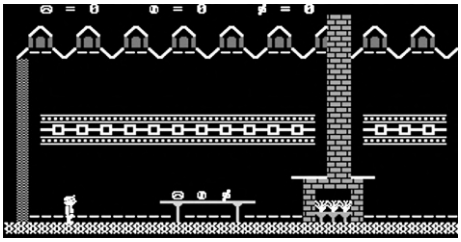


Figure 11-2: An example of typical 8-bit game graphics

Even just playing—rather than writing—games in the golden age could be subliminally educational about architecture. Games were usually written specifically to play with and explore the architecture, to push it to its limits, and to show off programming skills. For example, the 8-bit architecture encouraged games to use certain sizes of sprites and certain layouts of levels. You could animate *Space Invaders* easily by overwriting the area of memory where the *A* character was defined, replacing it with an 8×8 pixel space invader, then just print the *A* character on the screen to move it around, without needing any graphics commands. (The downside of this was that when you listed your program to debug it afterward, all the *A*s had also changed into space invaders.)

16-Bit Era

The late 1980s introduced 16-bit machines and continued this style of assembly programming, but with the extra bits and more advanced I/O modules enabling a move to *sampling* of images and sounds rather than their pure computer generation, as on 8-bit machines. These developments gave rise to the distinctive 16-bit aesthetics of sprite-based games like *Sonic* and *Mario* (Figure 11-3), and sample-based music by artists such as The Prodigy and the soundtracks of games such as *Streets of Rage 2*.

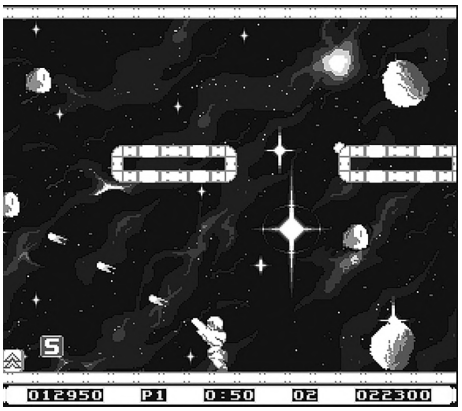


Figure 11-3: An example of typical 16-bit game graphics

Popular machines included the Commodore Amiga, Atari ST, Sega Megadrive, and Nintendo SNES. High-performance programs such as games and demos were still mostly written in assembly, with full access to memory, but they would make heavier use of calls to additional graphics and sound hardware.

Companies continued to produce 16-bit machines into the early 1990s, including for many now-classic games. But by this time most programmers had shifted to the C language, which could compile into assembly code for multiple machines, making it easier to port software between them. Programmers came to rely more on heavyweight operating systems, also accessed primarily via C libraries. Together, C and operating systems acted to wrap and hide architecture, presenting higher-level and more portable interfaces to the machines, but ending the 1980s golden age of architecture-based programming.

Good times! Let's relive them here by learning to program on two classic systems from the period, the 8-bit 6502-based Commodore 64 (C64) and the 16-bit 68000-based Commodore Amiga. For each system, we'll first study its CPU in isolation, then its wider computer design. In the exercises, we'll write assembly programs for a C64 animated text demo and a simple Amiga game.

Working with the MOS 6502 8-Bit CPU

MOS Technology's MOS 6502 was an 8-bit processor, designed in 1975 by Chuck Peddle. *MOS* stands for *metal-oxide semiconductor*, as in the MOS field-effect transistors, or MOSFETs, used by the company. The 6502 was used in many of the classic 8-bit micros of the 1980s: the Commodore 64, NES, Atari 2600, Apple II, and BBC Micro; it was also used in first-generation arcade machines such as *Asteroids*.

Here we'll study the 6502 using the same steps as for the Analytical Engine and Manchester Baby. We'll first examine its structures, including registers, the arithmetic logic unit (ALU), the decoder, and the control unit (CU). We'll then look at its instruction set, including instructions for memory access, arithmetic, and control flow.

Internal Subcomponents

The 6502 had 3,000 transistors and wires connecting them. The layout of these components was designed and drawn by hand on transparent sheets, with pens and masking tape, and then made directly into chips using photolithography.

NOTE *The term* taping out *is still used to refer to the equivalent modern computerized process of finalizing photolithography mask designs. For chip designers, a tape-out marks the end of their work and handover to a fab plant. Like "shipping" for software companies, taping out can be a reason to have a large party, lasting until the chips arrive in the mail and fail to work.*

Physically, the 6502 appears as a plastic-packaged integrated circuit (IC) about 2 cm long, with 40 pins, as seen in Figure 9-2. Eight of these are data pins, labeled D0 through D7. These pins read and write 8-bit words of data to and from memory, and they define the CPU as an 8-bit machine. The 6502 uses a 16-bit address space accessed by writing 16-bit addresses on the 16 address pins, A0 through A15. This enables up to 64 k_2B to be addressed. The R/W is the control line that specifies whether we want to read or write to the address. The package also has pins for ground and supply voltage, a clock, and an IRQ (interrupt request) line. The clock sets the speed of the CPU, usually to around 1 to 3 MHz.

The actual silicon chip is much smaller than the outer package, about 5 mm^2. Figure 11-4 shows a photograph of the chip under a microscope (known as a *die shot*).

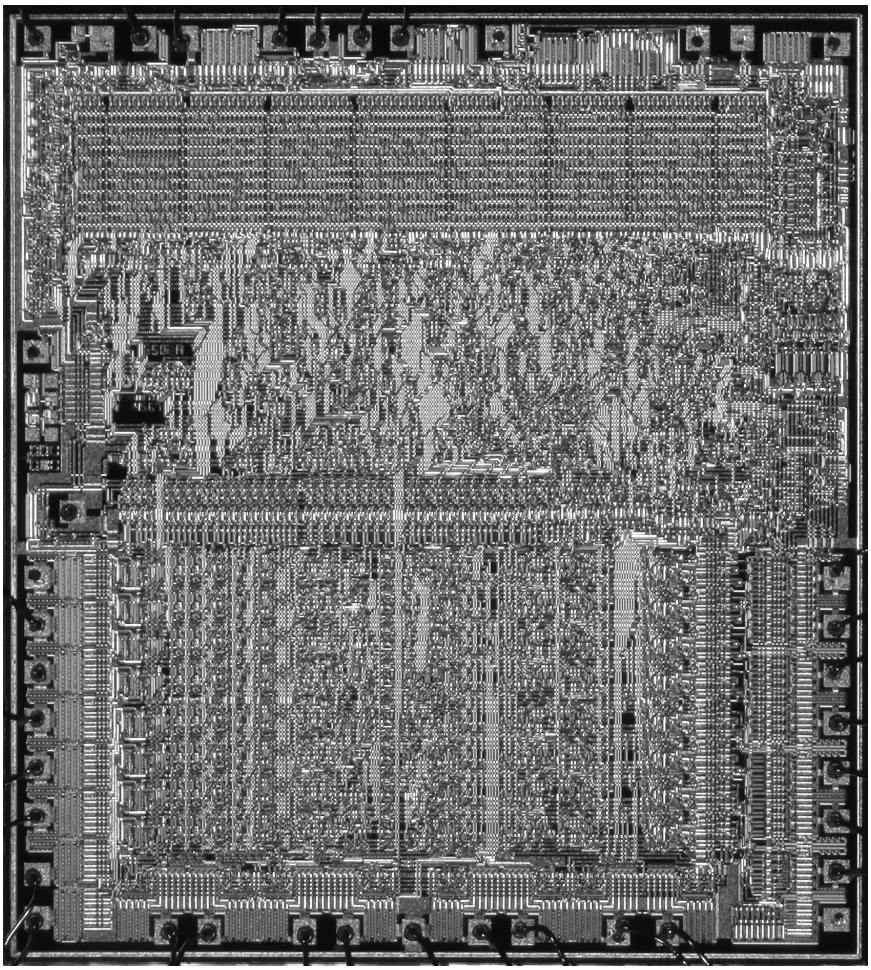


Figure 11-4: A 6502 chip microscope photograph

Details of this chip's design were lost for several decades, but they were recently fully reverse engineered at the transistor level in a heroic effort by the Visual 6502 project (*http://visual6502.org*). The workers on this project exposed the silicon by applying acid to dissolve some of the plastic casing. They then took die shots of the chip to reverse engineer its circuit diagram.

The circuit contains only transistors and copper wires, but some very skilled chip-reading people have learned to look at these and mentally chunk them into logic gates. From there, they were chunked into well-known simple machines. This painstaking process, guided by the surviving block diagram shown in Figure 11-5, enabled the whole architecture to be reverse engineered and reconstructed.

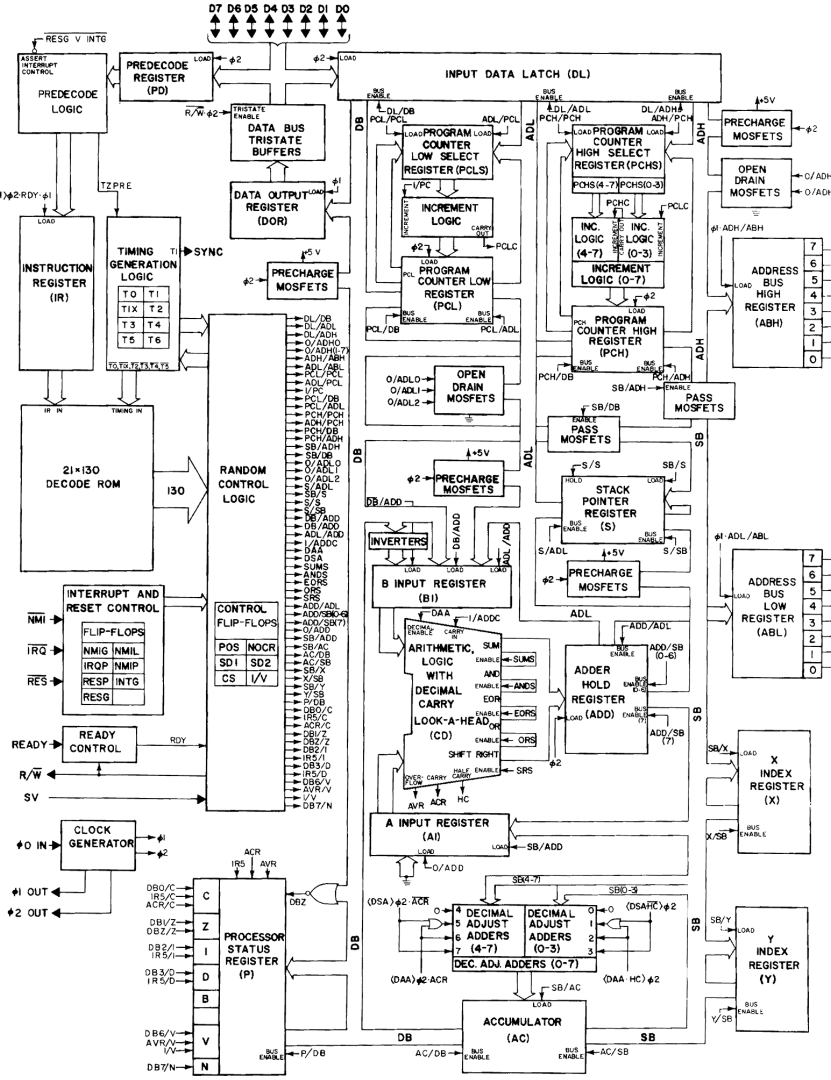
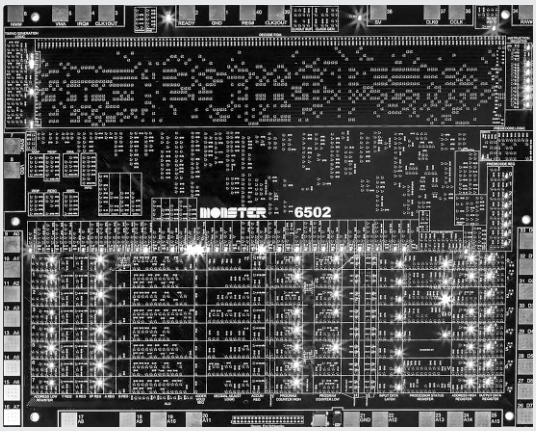


Figure 11-5: The original 6502 block diagram

The circuit in Figure 11-5 shows some recognizable subcomponents that are common to most classic-design chips. Each component is a digital logic simple machine. We'll examine each of the main subcomponents in turn.

THE MONSTER 6502

As a result of the Visual 6502 project, the 6502 is now being manufactured again for use as a cheap, embedded processor—for example, in Internet of Things (IoT) devices—as well as for education, such as our present study. The design was also used by Eric Schlaepfer and Evil Mad Scientist Laboratories to create a fully functioning—but slower than the original—MOnSter 6502 rebuild from big transistors rather than ICs, shown here.



User Registers

In Figure 11-4 the registers and ALU are the regular area in the lower half of the chip. The 8 bits are stacked vertically, as in Babbage's machines. There are three 8-bit user registers: two general-purpose ones called X and Y, and an accumulator called A.

The X and Y registers are intended to be usable together to represent 16-bit addresses, with the first 8 of the 16 bits stored in X and the second 8 bits stored in Y. It's quite hard work to manipulate the two halves separately, so the architecture often provides methods to manipulate the two 8-bit halves of 16-bit addresses together.

It's common to imagine and visualize 8-bit memory as divided into 256 pages of 256 bytes each. For example, 8-bit hex editors may display one page of memory on the screen at a time, like the pages of a book. Viewed this way, one of the two bytes is the page number and the other says what line on the page is used.

NOTE *It's common 6502 programming practice to use the 256 addresses in page 0 of memory as if they were additional registers. This is slower than using actual registers, so A, X, and Y are preferred in the first instance.*

Internal Registers

Recall that the program counter keeps track of the current line number. Here, the program counter stores a 2 byte address. It's automatically incremented by the CU after executing each instruction, unless it's a flow control instruction. It can be accessed as 2 individual bytes (PCH, PCL) storing the high and low parts of the 16-bit address. On power-up, the 6502 copies the content of addresses FFFC and FFFD (usually ROM, and pointing to a ROM subroutine) into the program counter to tell it where to begin executing.

Here, the stack pointer is a single byte, and it's assumed to refer to lines on page 1 of memory; note this is the second page, after page 0. In most programming styles, the stack isn't accessed directly by the programmer, but rather is used internally by the subroutine instructions to push and pop the address of the calling line. However, it can also be accessed directly using instructions (`PHA` and `PLA`) that push and pop the contents of the accumulator to and from the stack.

The instruction register holds a copy of the current instruction; in the 6502, opcodes are 8 bits long and may require 0, 1, or 2 bytes of operand. Because the data bus is 8-bit, fetching instructions usually requires several steps; the opcode and operand need to be copied in one byte at a time. This is one of the reasons why 8-bit machines are slow: machines with larger word lengths could fetch whole instructions, including opcodes and operands, as single words.

The status register holds 8 bits of flags that can be tested and used by control flow instructions. These are set in the ALU, which we'll turn to next.

Arithmetic Logic Unit

In the 6502, the ALU is physically built around the registers so that the 8 bits flow horizontally, as in Babbage's machines. The registers-ALU area looks a lot like Babbage's Difference Engine, and contains a similar parallel propagation of bits and carries. If you miniaturized Babbage's metal machine onto a chip, this is pretty much what it would look like; only the scale has changed.

The ALU provides integer addition and subtraction simple machines, activated by instructions (`ADC`, `SBC`), along with dedicated increment and decrement (`INC`, `DEC`). There are bitshifts and bitwise Boolean instructions (`ASL`,

ASR; AND, ORA, EOR). There are no multiplication or division instructions—these must be constructed in software from what's available. There is also no floating point.

Figure 11-6 shows the ALU's assigned meanings of the bits in the status register, which are written as side effects of its operations.

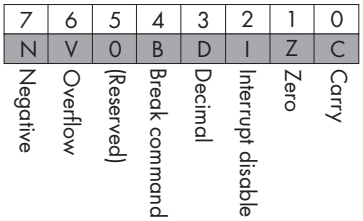


Figure 11-6: The status register flags

The ALU's operations include flagging if the result was zero (Z) or negative (N), if there was an overflow (V), and if there was a carry (C).

Decoder

In Figure 11-4, the decoder is visible in the upper one-fifth of the die shot as a semi-regular binary structure. It looks like a load of binary numbers stored in an array, which is pretty much what it is. Opcodes are 8 bits, meaning that 256 distinct instructions are possible. Each opcode is decoded and used to activate a control line.

Control Unit

In Figure 11-4, the CU forms the middle region of the chip. It appears visually as a highly irregular region. This is because every operation is different, so it's implemented with entirely different circuitry. The 6502 CU often needs to do more work than later 16-bit machines, because the 6502 operates with 16-bit address and sometimes 16- or 24-bit instructions, which the CU has to break up into 8-bit chunks and marshal over the 8-bit bus.

Programmer Interface

Figure 11-7 shows the complete instruction set for the 6502.

	Low nibble															
High nibble	0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	0F
0	BRK i	ORA (zp,x)				ORA zp	ASL zp		PHP i	ORA #	ASL A			ORA a	ASL a	
10	BPL r	ORA (zp),y				ORA zp,x	ASL zp,x		CLC i	ORA a,y				ORA a,x	ASL a,x	
20	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp		PLP i	AND #	ROL A		BIT a	AND a	ROL a	
30	BMI r	AND (zp),y				AND zp,x	ROL zp,x		SEC i	AND a,y				AND a,x	ROL a,x	
40	RTI i	EOR (zp,x)				EOR zp	LSR zp		PHA i	EOR #	LSR A		JMP a	EOR a	LSR a	
50	BVC r	EOR (zp),y				EOR zp,x	LSR zp,x		CLI i	EOR a,y				EOR a,x	LSR a,x	
60	RTS i	ADC (zp,x)				ADC zp	ROR zp		PLA i	ADC #	ROR A		JMP (a)	ADC a	ROR a	
70	BVS r	ADC (zp),y				ADC zp,x	ROR zp,x		SEI i	ADC a,y				ADC a,x	ROR a,x	
80		STA (zp,x)			STY zp	STA zp	STX zp		DEY i		TXA i		STY a	STA a	STX a	
90	BCC r	STA (zp),y			STY zp,x	STA zp,x	STX zp,y		TYA i	STA a,y	TXS i			STA a,x		
A0	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp		TAY i	LDA #	TAX i		LDY a	LDA a	LDX a	
B0	BCS r	LDA (zp),y			LDY zp,x	LDA zp,x	LDX zp,y		CLV i	LDA a,y	TSX i		LDY a,x	LDA a,x	LDX a,y	
C0	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp		INY i	CMP #	DEX i		CPY a	CMP a	DEC a	
D0	BNE r	CMP (zp),y				CMP zp,x	DEC zp,x		CLD i	CMP a,y				CMP a,x	DEC a,x	
E0	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp		INX i	SBC #	NOP i		CPX a	SBC a	INC a	
F0	BEQ r	SBC (zp),y				SBC zp,x	INC zp,x		SED i	SBC a,y				SBC a,x	INC a,x	

Figure 11-7: The complete 6502 instruction set. For full definitions of these instructions, see https://en.wikibooks.org/wiki/6502_Assembly.

Because opcodes are 8-bit, there's space for 256 instructions; notice, though, that the instruction set architecture contains a few less, so there are some gaps in the table.

Load and Store

Loading (`LD`) to and storing (`ST`) from the three user registers (X, Y, and A) is done using instructions such as:

```
LDA #$00  ; load to accumulator the constant 8-bit hex integer 00
STA $0200 ; store accumulator contents to 16-bit hex address 0200
LDX $0200 ; load contents of address 0200 to register X
STX $0201 ; store contents of X into address 0201
LDY #$03  ; load 8-bit constant hex 03 to register Y
STY $0202 ; store contents of Y to address 0202
```

Offset addressing enables the value of a user register to be used as an offset to a given address. This is useful for iterating over arrays. For example:

```
LDX #$01
STA $0200,X ; store the value of A at memory location $0201
```

Indirect addressing allows us to specify an address that in turns holds another address where we actually want to load or store:

```
LDA ($C000)  ; load to A from the address stored at address C000
```

Indirection and offsetting can be used together, such as:

```
LDA ($01),Y
```

Zero-paging is the 6502 convention that page 0 of memory is intended to function similarly to 256 additional registers. This requires specifying and moving around only 1 byte of address, as in:

```
LDA $12   ; single byte address assumed to be from page 0
```

This is faster than moving 2 bytes around individually.

Arithmetic

The `ADC` instruction means "add data with carry." It adds the integer contents of its address operand, and the carry bit from the status register, to the accumulator. The following program should end with hex value $0A_{16}$ (decimal 10) in the accumulator:

```
CLC       ; clear content of carry flag in status register
LDA #$07  ; load constant 07 to accumulator
STA $0200 ; store content of accumulator to address 0200
LDA #$03  ; load constant 03 to accumulator
ADC $0200 ; add with carry the content of 0200 into accumulator
```

`CLC` clears the carry flag; it's important to do this before any new addition, unless you want the carry from a previous operation to get added in as well.

To add two 16-bit integers, we can make use of the carry status flag state, instead of clearing it. Each `ADC` reads and writes it, so we can split a 16-bit addition into a pair of two 8-bit additions with a carry. Here, the two inputs, `num1` and `num2`, and the output, `result`, are each split into low and high bytes:

```
CLC
LDA num1_low
ADC num2_low
STA result_low
LDA num1_high
ADC num2_high
STA result_high
```

Similarly, `SBC` is "subtract with carry," so the following computes 7 − 3, resulting in the value 4 in the accumulator:

```
SEC       ; set carry flag to 1 (needed to init subtraction)
LDA #$03  ; load constant 3 to accumulator
STA $0200 ; store constant 3 to address 0200
LDA #$07  ; load constant 7 to accumulator
SBC $0200 ; subtract content of 0200 from accumulator
```

We can increment (INC or IN) and decrement (DEC or DE) both address and register contents with instructions such as:

```
LDX #$02
LDY #$04
INX
DEY
LDA #$07
STA $0200
INC $0200
DEC $0200
```

Here again, # denotes that the operand is a constant, with the other operands being addresses.

Jump and Branches

JMP is the jump instruction. The following program continually increments register X, which will overflow after FF_{16}, going back to 00_{16}:

```
LDX #$02
mylabel:
  INX
  JMP mylabel
```

Instead of specifying the line number to jump to—as a BASIC programmer of the era might—this notation first marks the destination line with a *label*—in this case, mylabel—then specifies the name of this label in the jump instruction. The label line doesn't compile to machine code; it's ignored when first seen by the assembler. But when the assembler sees the label again in the jump instruction, it replaces it with the address of the instruction following the label.

Conditional branching can be done in two stages. First, comparison instructions check if some condition is true and store the result in the status register. Then, branch instructions consult the status register to decide when to branch. For example, the following uses register X to count down from 5 to 2 then halt, by comparing X to 2 (CPX) and branching if the comparison isn't equal (BNE):

```
LDX #$05
mylabel:
  DEX
  CPX #$02
  BNE mylabel
```

You can also branch (`B`) if the comparison was equal (`BEQ`), negative (on minus, `BMI`), or positive (on plus, `BPL`). Or if the carry (`C`) or overflow (`V`) flag is clear (`C`) or set (`S`): `BCC`, `BVC`, `BCS`, `BVS`, respectively.

Subroutines

`JSR` and `RTS` jump to and return from a subroutine, respectively. For example, the following program uses a common convention of placing arguments for a subroutine into addresses at the start of memory, which are then picked up by the subroutine code. `BRK` is "break," roughly the 6502's halt instruction (actually an interrupt). It's needed to prevent the main program execution overrunning into the code of the subroutine after it.

```
LDA #$5    ; load first argument to accumulator
STA $0001  ; put it in address 1 for sub to pick up
LDA #$4    ; load second argument to accumulator
STA $0002  ; put it in address 2 for sub to pick up
JSR mysub  ; call the subroutine
STA $0200  ; use subroutine's result, is in accumulator
BRK        ; halt
mysub:
  LDA #$00   ; reset the accumulator
  CLC        ; reset the carry
  ADC $0001  ; add in the first argument
  ADC $0002  ; add in the second argument
  RTS        ; return from subroutine
```

In the exercises, you'll see how to run the above and similar examples on an emulated standalone 6502. A 6502 by itself isn't very exciting, though. We need a computer design to add memory and I/O to the CPU, so now let's zoom out from the 6502 and look at a complete computer design, the Commodore 64, based upon it.

8-Bit Computer Design with the Commodore 64

The 6502-based Commodore 64, or C64, was and still is the highest-selling computer model of all time. Released in 1982, it defined the 8-bit home computing market in most of the world by combining gaming features with the potential for business and creative applications. Commodore was so named because its founder, the colorful Holocaust survivor Jack Tramiel, originally wanted "General Computers," like "General Electric," but "General" was taken. Commodore is a lower, second-choice rank below general. The C64 board was shown previously in Figure 9-1. Its name comes from the fact that it used the full 64 k_2B of available memory from its 16-bit address space with 8-bit words (2^{16} addresses × 8 bits = 64 k_2B), unlike some other 6502-based machines.

Understanding the Architecture

MOS produced several variants of the 6502 and assigned different model numbers to each. As with 7400 logic chips, "6502" is thus ambiguous, sometimes used to mean the original, numbered CPU design, and other times referring to all members of the family, which each have related numbers. The 6502 family member used in the Commodore 64 is more precisely known as the 6510.

In addition to a full 64 k_2B of actual RAM, the C64 also added devices, I/O modules, and their own ROMs containing libraries of subroutines for talking to them (what we now call a BIOS). It's this configuration that differentiates the C64 from other 6502-based machines as a programming platform.

The physical board layout is connected as in the block diagram of Figure 11-8. The bus—consisting of 16-bit addressing and 8-bit data—dominates this diagram and connects the CPU, RAM, ROM, and I/O.

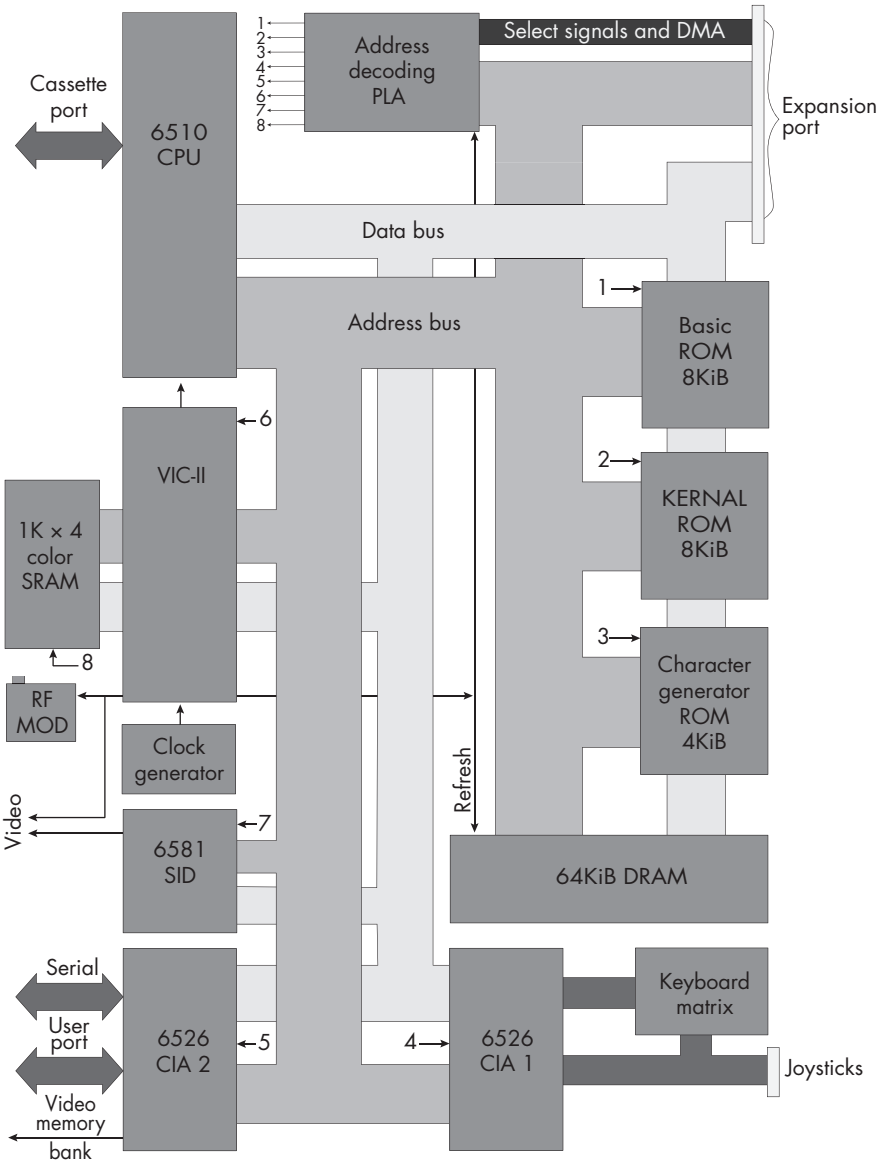


Figure 11-8: A C64 block diagram

Figure 11-9 shows the memory map for the C64.

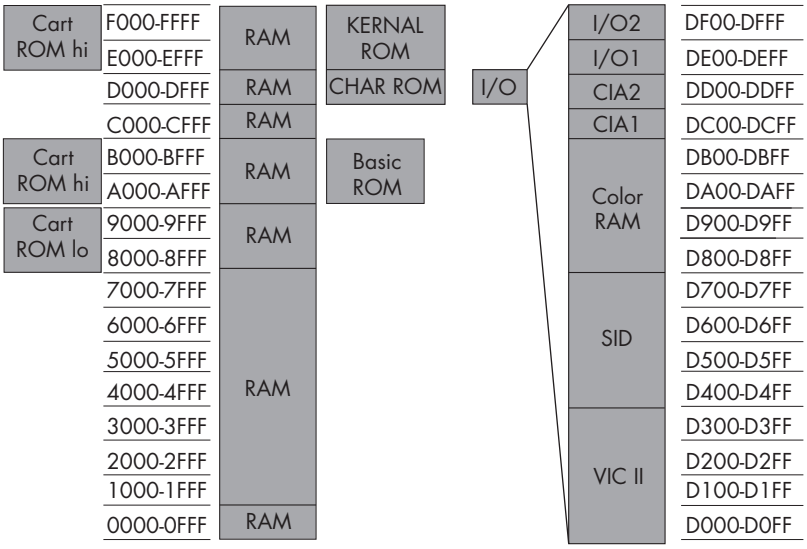
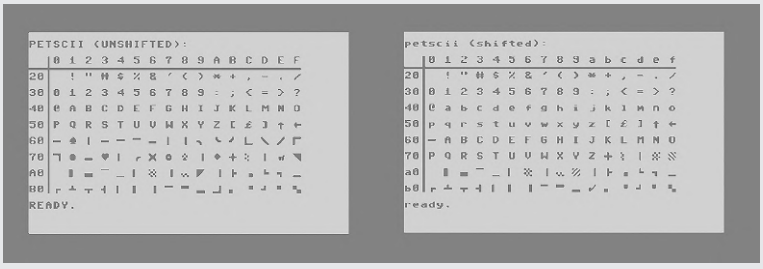


Figure 11-9: A C64 memory map

In this memory map, the RAM, ROM, and I/O are each assigned address ranges within the 16-bit address space. The I/O address space is broken down into ranges used by individual I/O modules and chips. (Because address space was a scarce resource at this time, the C64 enables the programmer to temporarily disconnect the ROMs from it and mount additional RAM in their place.)

PETSCII CHARACTERS

The Commodore 64 extended ASCII differently from Unicode, into a now dead branch of evolution called PETSCII, by using the first digit as a "shift" character and defining a second set of ASCII-like symbols. It also defined new visual symbols in the unshifted sector in place of control codes, including the C64's iconic playing card symbols and bitmap graphic elements used for drawing and shown here.



Programming the C64

Programming the C64 is done in 6502 assembly as discussed in the "Programmer Interface" section on page 255, but with added interactions with the particular ROMs and I/O modules mounted in the address space. The ROMs contain libraries of Commodore's own subroutines (known as "KERNAL," with an A). I/O includes a memory-mapped screen display that can be switched between character and pixel modes. Character mode allows PETSCII characters to be drawn at screen locations by writing their codes directly into this memory space. The state of the keyboard can be read by reading its memory-mapped space, but ROM subroutines are provided to simplify this process and decode its state to PETSCII character codes.

The following program illustrates these structures. It displays a scrolling message on a colored screen, and exits when the A key is pressed.

```
screenbeg = $0400           ; const, beginning of screen memorymap
screenend = $07E7           ; const, end of screen memorymap
screenpos = $8000           ; variable, current position in screen
main:
    LDA #$02                ; black color code
    STA $D020               ; I/O border color
    STA $D021               ; I/O background color
    STA screenpos           ; screen position
loop:                       ; main game loop, once per frame
    JSR $E544               ; ROM routine, clears screen
    JSR drawframe           ; most of the work is done here
    JSR check_keyboard
    INC screenpos           ; increment current screen position
    JMP loop                ; do the loop, forever
drawframe:
    LDX #$00                ; regX tracks idx of char in the string
    LDY screenpos           ; regY keeps scrolling screen position
    CPY #$20                ; compare Y with constant 20
    BCS resetscreenpos      ; branch if Y>20 (stored in carry bit)
drawmsgloop:                ; drop through to here if not branching
    LDA msg,X               ; load the xth char of the message
    BEQ return              ; exit when zero char (end of string)
    AND #$3F                ; convert ASCII to PETSCII
    STA screenbeg,Y         ; VDU: write char in A to memorymap offset Y
    INX                     ; increment idx of char in message
    INY                     ; increment location on screen
    CPY #$20                ; are we trying to write offscreen?
    BCS wraparound_y        ; if so, shift offset by screen width
    JMP drawmsgloop         ; loop (until all chars are done)
resetscreenpos:
    LDY #$00
    STY screenpos           ; reset the screenpos to 0
    JMP drawmsgloop
```

```
wraparound_y:               ; if Y trying to write off screen, wrap
    TYA                     ; transfer Y to accumulator
    SBC #$20                ; subtract with carry
    TAY                     ; transfer accumulator to Y
    JMP drawmsgloop
check_keyboard:
    JSR $FF9F               ; ROM SCANKEY IO, writes keybdmatrix to 00CB
    JSR $FFE4               ; ROM GETIN, convert matrix to keycode in acc
    CMP #65                 ; compare accumulator to ASCII 'A'
    BNE return
    BRK                     ; if 'A' pressed, quit
return:
    RTS
msg:
    .byte "HELLO C64!\0"    ; this is data, not an instruction
```

This creates a scrolling text result, as in Figure 11-10.



Figure 11-10: The hello C64 result. The text scrolls across the screen.

The program can be used as the starting point for writing a game, as it includes all of the basic game elements: a loop, display, keyboard read, and state update.

CHIPTUNES

In the 8-bit era, the sound chip was a genuine synthesizer, an actual musical instrument made in hardware and placed inside the computer.

The simplest way to generate tones is to use square waves. This is how a non-musician architect would go about building a sound chip, such as the Texas Instruments SN76489. Square waves alternate at a given frequency (musical pitch) between a digital 0 and 1, so they can be made entirely from digital logic rather than requiring the analog voltages that would be needed to make other waveforms. Limiting chips to square waves gave devices of the era their characteristic, primitive 8-bit sound.

As Commodore had bought MOS, they used MOS's latest tone generator, the 6581 Sound Interface Device (SID), in the C64. SID was far superior to previous sound chips. It was designed as a real musical instrument, by a musical synthesizer designer. It added analog sawtooth and sine waves into the mix, and revolutionized 8-bit audio by adding analog filters to these waves. Filters emphasize or mute bands of harmonics in a musical signal. Both square and sawtooth waves have infinite harmonics, which provide good raw material for filters to act upon. Filters can be swept over notes in many different ways to create many effects, and this gave the C64 its large musical palette.

SID contains the analog device and an I/O module that interfaces it to the address space, so it attaches to the bus. In the C64, it's controlled by writing parameters such as frequencies, volumes, and filter cutoffs to its assigned address space, D400 to D7FF, as in the following example, which plays a square wave on channel 1:

```
main:
    LDA #$0F
    STA $D418  ; I/O SID volume
    LDA #$BE   ; attack duration = B, decay duration = E
    STA $D405  ; I/O SID ch1 attack and decay byte
    LDA #$F8   ; sustain level = F, release duration = 8
    STA $D406  ; I/O SID ch1 sustain and release byte
    LDA #$11   ; frequency high byte = 11
    STA $D401  ; I/O SID ch1 frequency high byte
    LDA #$25   ; frequency low byte = 25
    STA $D400  ; I/O SID ch1 frequency low byte
    LDA #$11   ; id for square wave waveform
    STA $D404  ; I/O SID ch1 ctl register
loop:
    JMP loop
```

After SID's release, the great 8-bit "chiptune" composers such as Rob Hubbard found highly creative ways to hack it to play samples and to appear to have many more voices than the three it had in hardware. SID presented a limited and constrained palette, encouraging minimalist, mathematical aesthetics. Hubbard was influenced by Philip Glass, Jean-Michel Jarre, and Kraftwerk. More recently, music producers in the 2010s, such as Max Martin and Dr. Luke, have used SID for its retro gaming sound.

Working with the Motorola 68000 16-Bit CPU

The 16-bit era is somewhat misnamed: it should have been the "16/32-bit era." This is because the defining chip of the era was the Motorola 68000, used in the Commodore Amiga, Atari ST, Apple Macintosh, and Sega Megadrive, as well as in arcade machines such as *Street Fighter II*. The 68000 used 16-bit data words, but also had 32-bit registers and an ALU inside the CPU. The Atari ST's name refers to this hybrid "*S*ixteen/*T*hirty-two" nature of the 68000. Also known as the 68k, the 68000 was released in 1979 and appeared in computers in the later 1980s to define the 16-bit era.

Both the 6502 and 68000 descended from the earlier Motorola 6800, in separate branches of evolution. Their names reflect this, and they share some structures and instructions. This means that learning the 68000 is often an extension of what we learned about the 6502. If you're unsure of how to do something in the 68000, you can often make a good guess based on the 6502 equivalent.

Internal Subcomponents

Figure 11-11 shows a die shot of the Motorola 68000. In the figure, you can see the same basic structure as in the 6502, with the registers and ALU at the bottom, control logic in the center, and decoder near the top.

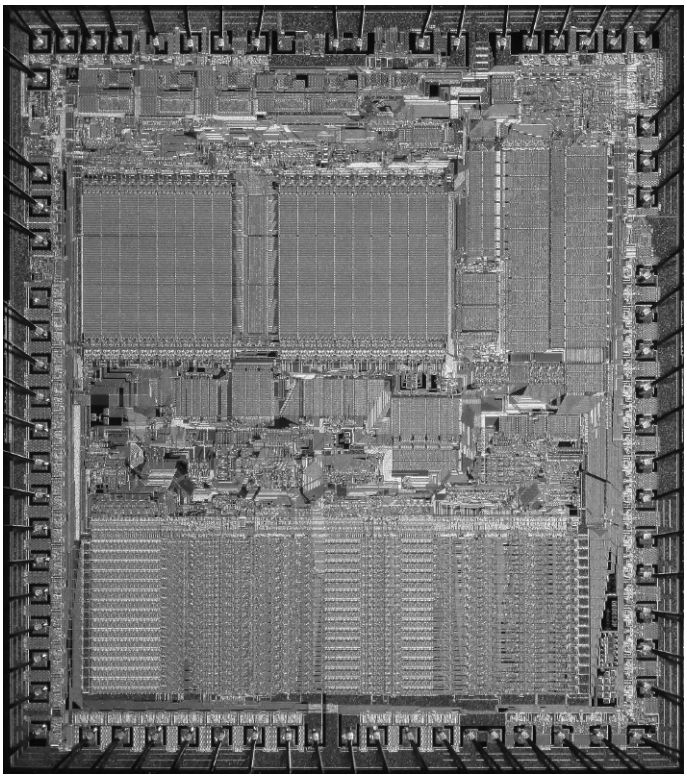


Figure 11-11: A 68000 die shot

You can see that the registers and ALU section now has more repeated rows due to having more than 8 bits. Unlike the 6502, the digital logic is now too small to see when the whole CPU is shown on a printed page.

There are 16 user registers, all 32-bit, of which 8 are called D0 to D7 for "data registers," and the others are called A0 to A7 for "address registers." A7 is used as the stack pointer. There's a 16-bit status register containing similar status bits to the 6502, and some extra information.

The bus has 16 data lines and 24 address lines. The addresses, however, refer to locations of bytes rather than 16-bit words, so there are 2^{24} addressable bytes, which is 16 M_2B of addressable memory. The 24-bit addresses are written as six hex characters, such as $DFF102_{16}$.

The 68000 has a two-stage pipeline that fetches the next instruction while simultaneously decoding and executing the current one.

Programmer Interface

As we've done for other machines, having seen the structure of the 68000 we'll now examine the instruction set that it enables—via memory access, arithmetic, and flow control—and that you can use to write your own programs. The 16-bit era saw a widespread shift from the use of upper- to lowercase characters in programming, which we'll respect from here onward.

Data Movement

A single `move` instruction is used for load, store, and register data transfers:

```
move.l d0, d1         ; copy from register d0 to register d1
move.l #$1a2, d1      ; copy hex constant $1a2 to register d1
move.l $0a3ff24, d1   ; load longword from address 0a3ff24 to d1
move.l d1, $0a3ff24   ; store longword from d1 to address 0a3ff24
```

The `l` here stands for "longword" and moves 32 bits at a time. This is fast between registers. When accessing memory, the 32 bits must be split up by the CPU and sent over the 16-bit bus in two steps, sequenced by the CU.

If you only want to move 16-bit words (`w`) or 8-bit bytes (`b`) around, you can use variants of `move`:

```
move.b d0, d1
move.w $0a3ff24, d1
```

Indirect addressing is specified using parentheses:

```
move.l ($0a3ff24), d1   ; load content from addr stored at addr 0a3ff24, to d1
```

Offset addressing includes the following:

```
move.l (pc, 2), d1      ; load content from program counter plus 2
move.l (a1, a2), d1     ; load content from addr formed as sum of regs a1+a2
move.l (a1, a2, 2), d1  ; load content from addr formed as sum of regs a1+a2+2
```

A more complicated and unusual 68000 addressing mode combines indirect addressing with register incrementation; this is useful for iterating over data stored in contiguous addresses:

```
move.l (a1)+, d1        ; load content from addr stored in register a1, to d1,
                        ; then increment a1 by number of bytes in a longword
move.l -(a1), d1        ; decrement a1 by number of bytes in a longword,
                        ; then load content from addr stored in register a1
```

For C programmers: this is roughly what `*(a++)` and `*(--a)` would compile into. Pushing and popping the stack doesn't need dedicated instructions because it can be done using this mode with the stack pointer register:

```
move.w (sp)+, d0        ; push from register d0 to stack
move.w d0, -(sp)        ; pop from stack to register d0
```

Load effective addresses (`lea`) is a related 68000 instruction that can load the address of indirections. For example:

```
lea (pc, 2), a1     ; put address of program counter +2 bytes into a1
lea (a1, 2), a3     ; put address a1+2 into a3
lea (a1, a2, 2), a3 ; put address a1+a2+2 into a3
```

Note that `lea` loads the numerical address itself, rather than the content of the address.

Flow Control

Due to their shared history, jumps, subroutines, and branches on the 68000 are the same as on the 6502. For example:

```
start:
   jsr mysub        ; jump to subroutine

   cmp #2, d0       ; compare values
   beq mylabel      ; branch if equal
   ble start        ; branch if less than or equal
   bne start        ; branch if not equal

mylabel:
   jmp mylabel      ; infinite loop

mysub:
   rts              ; return from subroutine
```

That said, stack logic improved: with the 68000 you can push a series of arguments to the stack, make a jump to a subroutine, and pop them off from inside the subroutine. This allows subroutines to behave like functions with parameters.

Arithmetic

Here are some examples of arithmetic instructions:

```
add.b d0, d4  ; add d0 to d4, store result in d4
sub.w #43, d4 ; subtract constant 43 from d4, store result in d4
muls d0, d4   ; multiply (signed) d0 with d4, store result in d4
mulu d0, d4   ; multiply (unsigned) d0 with d4, store result in d4
divs d0, d4   ; divide (signed) d0 by d4, store result in d4
divu d0, d4   ; divide (unsigned) d0 by d4, store result in d4
and d0, d1    ; bitwise and d0 with d1, store result in d1
asr d0, d1    ; arithmetic shift right d1 by d0 bits, store result in d1
```

The addition and subtraction instructions are similar to those for the 6502. But unlike the 6502, the 68000 can perform multiplication and division in hardware.

16-Bit Computer Design with the Commodore Amiga

Amiga is the feminine of *amigo*, meaning *friend*, and Commodore's 1985 Amiga was intended to have that kind of relationship with its users. Early versions of the Amiga were intended as high-end graphics workstations and marketed to self-described "creatives"—the market now targeted by Apple. However, the now-classic A500 model rapidly became a standard mass-market gaming platform. This became self-fulfilling as both developer and gamer populations increased together. Growth was accelerated by the ease of (illegally) cracking and copying game disks, with bars in many towns around the world hosting "Amiga nights" where they were traded. In Europe, the Amiga was adopted by the "demo scene," a subculture of artistic assembly programmers who met up to compete at pushing the graphics and sound to their limits, not in games but in multimedia demonstrations. These scenes overlapped, with crackers adding demos to the boot sequences of newly cracked games (those with the copy-protection removed). Commodore management ignored all this and tried to push the Amiga in the business market, where it and the company were destroyed by beige-box PCs.

Understanding the Architecture

The classic A500 had 0.5 M_2B of RAM, though it and its successors were upgradable to a few mebibytes. (This was still much smaller than the 16 M_2B addressable by the CPU.) Figure 11-12 shows the A500 mainboard.

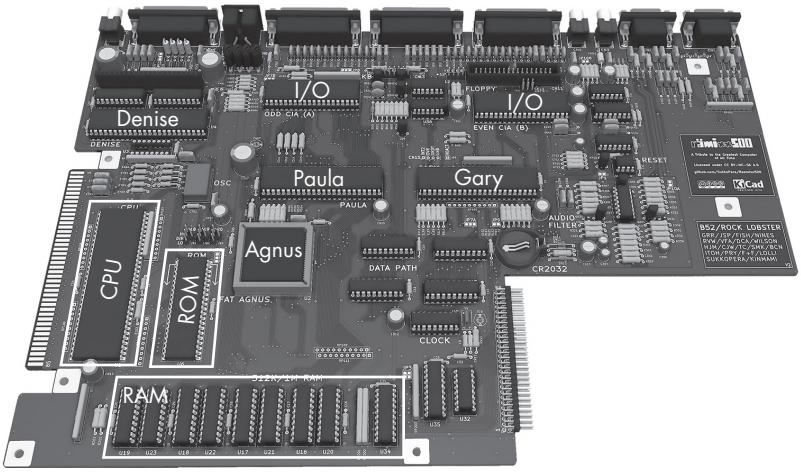


Figure 11-12: An Amiga A500 mainboard

The design was based around four large custom chips, given human names:

Agnus This chip contained a coprocessor ("copper") with its own separate RAM and bus, in addition to the main CPU system. The copper was responsible for graphics. Machine code for the copper could be written as data lines and sent to the copper as data by the main CPU program. (A similar system is used today in GPUs.) Agnus also contained a DMA-based "blitter," used for copying sprites onto video RAM without CPU.

Paula This chip contained a sound device and its I/O module, as well as several other I/O modules, such as for disks and communications ports. It used DMA to read audio samples and other I/O data from RAM without CPU intervention.

Denise This was the VDU chip, reading sprites and bitplanes from RAM, compositing them together under various screen modes, and outputting CRT display controls.

Gary This was a memory controller, translating and routing addresses from the bus to particular chips and addresses within them.

The A500 BIOS (called Kickstart) provides subroutines for accessing I/O, such as graphics and sound. It comes on a chip usually described as a ROM but that is more correctly considered an I/O module. This is because, unlike the C64 BIOS, these subroutines aren't mounted into address space directly. Instead, they're stored on a part of the chip that isn't directly addressed. When a subset (library) of the subroutines is needed, a command is sent to the smaller, addressed part of the chip to copy them into a new location in RAM.

The whole computer was synchronized to the clock rate of the TV CRT scan display, meaning that it (and its games) ran at different speeds in the UK and US due to their different TV standards!

The Amiga was designed as a multimedia machine, and a fundamental requirement, especially for the 16-bit games of the time, was to quickly draw sprites—small images such as game characters—that are overlaid onto backgrounds to build up a scene.

A naive method to draw sprites is to store a primary copy of the sprite at a fixed location in RAM, then write a subroutine in assembly language to copy each pixel one at a time to a parameterized location in video RAM. However, this is very slow because every pixel in the sprite then needs to be loaded into the CPU and written out again to video RAM in sequence.

"Blitting" was a famous use of DMA in the Amiga copper to render sprites more efficiently. The copper could be commanded by the CPU to initiate a complete sprite "blit" by DMA. It would then read the sprite (or blitter object, "bob") pixel by pixel from its location in regular RAM, and copy it into video RAM without any further CPU intervention.

"Hardware sprites" were a second method, in which the primary copy of the sprite was loaded into the VDU at the start of a game. The VDU contained its own dedicated digital logic to implement similar blitting commands internally. There were severe memory limits inside the VDU, allowing only eight hardware sprites, which could often be used for the animation frames of a main character in a game or for the mouse pointer symbol.

For backdrops of 2D games, "playfields" are another hardware acceleration, which allow backdrop images to be stored and scrolled around. Several can be overlaid with transparency masks to create parallax effects.

Programming the Amiga

The following is a short program that displays a spaceship sprite on the screen:

```
custom      equ   $dff000     ; custom chips
bplcon0     equ   $100        ; bitplane control register 0 (misc, control bits)
bplcon1     equ   $102        ; bitplane control register 1 (horizontal, scroll)
bplcon2     equ   $104        ; bitplane control register 2 (priorities, misc)
bpl1mod     equ   $108        ; bitplane modulo
ddfstrt     equ   $092        ; data-fetch start
ddfstop     equ   $094        ; data-fetch stop
diwstrt     equ   $08E        ; display window start
diwstop     equ   $090        ; display window stop
copjmp1     equ   $088        ; copper restart at first location
cop1lc      equ   $080        ; copper list pointer
dmacon      equ   $096        ; DMA controller
sprpt       equ   $120        ; sprite pointer

COLOR00     equ   $180        ; address to store COLOR00 (background)
COLOR01     equ   COLOR00+$02 ; address to store COLOR01 (foreground)
COLOR17     equ   COLOR00+$22 ; etc
COLOR18     equ   COLOR00+$24
COLOR19     equ   COLOR00+$26
```

```
BPL1PTH     equ   $0E0        ; bitplane 1 pointer hi byte
BPL1PTL     equ   BPL1PTH+$02 ; bitplane 1 pointer lo byte
SPR0PTH     equ   sprpt+$00   ; sprite0 pointer, hi byte
SPR0PTL     equ   SPR0PTH+$02 ; sprite0 pointer, lo byte
SPR1PTH     equ   sprpt+$04   ; sprite1 etc
SPR1PTL     equ   SPR1PTH+$02
SPR2PTH     equ   sprpt+$08
SPR2PTL     equ   SPR2PTH+$02
SPR3PTH     equ   sprpt+$0C
SPR3PTL     equ   SPR3PTH+$02
SPR4PTH     equ   sprpt+$10
SPR4PTL     equ   SPR4PTH+$02
SPR5PTH     equ   sprpt+$14
SPR5PTL     equ   SPR5PTH+$02
SPR6PTH     equ   sprpt+$18
SPR6PTL     equ   SPR6PTH+$02
SPR7PTH     equ   sprpt+$1C
SPR7PTL     equ   SPR7PTH+$02

SHIPSPRITE  equ $25000        ; address to store our ship sprite
DUMMYSPRITE equ $30000        ; address to store our dummy sprite
COPPERLIST  equ $20000        ; address to store our copper list
BITPLANE1   equ $21000        ; address to store our bitplane data

; Define bitplane1
        lea     custom,a0               ; a0 := address of custom chips
        move.w  #$1200,bplcon0(a0)      ; 1 bitplane color
        move.w  #$0000,bpl1mod(a0)      ; modulo := 0
        move.w  #$0000,bplcon1(a0)      ; horizontal scroll value := 0
        move.w  #$0024,bplcon2(a0)      ; give sprites priority over playfields
        move.w  #$0038,ddfstrt(a0)      ; data-fetch start
        move.w  #$00D0,ddfstop(a0)      ; data-fetch stop

; Define display window
        move.w  #$3c81,diwstrt(a0)      ; set window start (hi byte = vertical, lo = horiz*2)
        move.w  #$ffc1,diwstop(a0)      ; set window stop (hi byte = vertical, lo = horiz*2)

; Put RGB constants defining colors into the color registers
        move.w  #$000f,COLOR00(a0)      ; set color 00 (background) to blue (00f)
        move.w  #$0000,COLOR01(a0)      ; set color 01 (foreground) to black (000)
        move.w  #$0ff0,COLOR17(a0)      ; Set color 17 to yellow (ff0)
        move.w  #$00ff,COLOR18(a0)      ; Set color 18 to cyan (0ff)
        move.w  #$0f0f,COLOR19(a0)      ; Set color 19 to magenta (f0f)

; Copy copper list data to addresses starting at COPPERLIST
        move.l  #COPPERLIST,a1          ; a1 := copper list destination
        lea     copperl(pc),a2          ; a2 := copper list source
```

```
cloop:
        move.l  (a2),(a1)+              ; copy DMA command
        cmp.l   #$fffffffe,(a2)+        ; end of list?
        bne     cloop                   ; loop until whole list moved

; Copy sprite to addresses starting at SHIPSPRITE
        move.l  #SHIPSPRITE,a1          ; a1 := sprite destination
        lea     sprite(pc),a2           ; a2 := sprite source
sprloop:
        move.l  (a2),(a1)+              ; copy DMA command
        cmp.l   #$00000000,(a2)+        ; end of sprite?
        bne     sprloop                 ; loop until whole sprite moved

; All eight sprites are activated at the same time but we will only use one
; Write a blank sprite to DUMMYSPRITE, so the other sprites can point to it
        move.l  #$00000000,DUMMYSPRITE

; Point copper at our copper list data
        move.l  #COPPERLIST,cop1lc(a0)

gameloop:

; Fill bitplane pixels with foreground color (1-bit plane in fore/background colors)
        move.l  #BITPLANE1,a1           ; a1 := bitplane
        move.w  #1999,d0                ; 2000-1(for dbf) long words = 8000 bytes
floop:
        move.l  #$ffffffff,(a1)+        ; put bit pattern $ffffffff as next row of 16*8 pixels
        dbf     d0,floop                ; decrement, repeat until false

; start DMA, to blit the sprite onto the bitplane
        move.w  d0,copjmp1(a0)          ; force load to copper program counter
        move.w  #$83A0,dmacon(a0)       ; bitplane, copper, and sprite DMA

    ;**your game logic would go here---read keyboard, move sprites**

    jmp gameloop

; Copper list for one bitplane, and eight sprites.  Bitplane is at BITPLANE1
; Sprite 0 is at SHIPSPRITE; other (dummy) sprites are at DUMMYSPRITE
copperl:
        dc.w    BPL1PTH,$0002           ; bitplane 1 pointer := BITPLANE1
        dc.w    BPL1PTL,$1000
        dc.w    SPR0PTH,$0002           ; sprite 0 pointer := SHIPSPRITE
        dc.w    SPR0PTL,$5000
        dc.w    SPR1PTH,$0003           ; sprite 1 pointer := DUMMYSPRITE
        dc.w    SPR1PTL,$0000
        dc.w    SPR2PTH,$0003           ; sprite 2 pointer := DUMMYSPRITE
```

```
        dc.w    SPR2PTL,$0000
        dc.w    SPR3PTH,$0003           ; sprite 3 pointer := DUMMYSPRITE
        dc.w    SPR3PTL,$0000
        dc.w    SPR4PTH,$0003           ; sprite 4 pointer := DUMMYSPRITE
        dc.w    SPR4PTL,$0000
        dc.w    SPR5PTH,$0003           ; sprite 5 pointer := DUMMYSPRITE
        dc.w    SPR5PTL,$0000
        dc.w    SPR6PTH,$0003           ; sprite 6 pointer := DUMMYSPRITE
        dc.w    SPR6PTL,$0000
        dc.w    SPR7PTH,$0003           ; sprite 7 pointer := DUMMYSPRITE
        dc.w    SPR7PTL,$0000
        dc.w    $ffff,$fffe             ; copper list end

; Sprite data. Stores (x,y) screen coordinate and image data
sprite:
        dc.w    $6da0,$7200             ; 6d = y location; a0 = x location; 72-6d = 5 = height)

        dc.w    $0000,$0ff0             ; image data, 5 rows x 16 cols x 2 bit color
        dc.w    $0000,$33cc             ; each line describes one row of 16 pixels
        dc.w    $ffff,$0ff0             ; each pixel is described by a 2-bit color
        dc.w    $0000,$3c3c             ; the low pixel bits form the first word
        dc.w    $0000,$0ff0             ; the high pixel bits form the second word

        dc.w    $0000,$0000             ; ... all zeros marks end of image data
```

Here, the sprite is defined in the data segment at the end. Amiga programs tend to involve a lot of defining constants for use with the many complex ROM I/O subroutines used to call its graphics capabilities. In real life, library files would be included to make the most of these definitions, but they're shown here in full as an illustration of a complete program.

A screenshot of the result is shown in Figure 11-13. Note that the sprite doesn't yet move, but further commands could be added to create a game loop that repeatedly reads the keyboard, updates the sprite location, then does the drawing. In real games, sprites aren't usually defined as data lines in assembly; rather, they're drawn in the famous pixel art program *Deluxe Paint*, then loaded into similar memory areas from files.

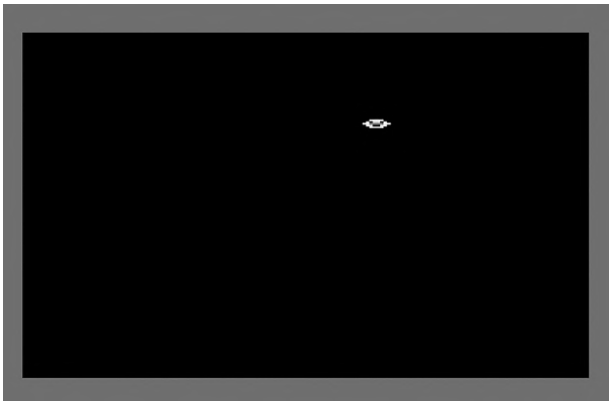


Figure 11-13: The Amiga sprite game result

Retro Peripherals

The 8-bit and 16-bit eras introduced many peripherals that either are still with us today or have had strong influences on modern standards. Let's look at some of the most important ones here to complete our study of retro computing.

Cathode Ray Tube Displays

While the Manchester Baby's Williams tube, seen in the "Historical RAMs" box on page 220, wasn't originally intended as a human display device, its programmers were quick to realize its potential for this use, and they soon began to write deliberately human-readable patterns to some parts of the screen as a form of output, with the rest of the screen storing internal data that appears as random patterns of on and off pixels. In recent decades, hackers have written simple retro arcade games to play on the Baby, displaying *Snake* and *Space Invaders* on parts of its Williams tube as the display.

These green-on-black pixels are the origin of the later cathode ray tube (CRT) green screen, and then color monitors, used as human displays in the retro age, as seen in Figure 1-31.

Programmers grew accustomed to the green color scheme, and in the 1980s often switched their RGB monitors into high-resolution green-on-black modes with a hardware switch to aid their concentration and familiarity. Some claimed that using only the green pixels improved the precision of the display, as the red and blue sub-pixels are some distance away from the green so tend to blur the pixel when used. Today, we still follow this tradition when we put our terminal emulators and text editors such as Vim into green-on-black mode. This classic programming scheme is celebrated in the stylized computer code in the movies *Ghost in the Shell* and *The Matrix*.

To reduce costs, home computers of the golden age were often designed to use consumer television CRTs as RGB monitors. To display to a CRT monitor or TV, an 8-bit machine such as the C64 first needs to read the desired pixel values from video RAM, then arrange for them to be mapped

onto the strength of the CRT beam as it periodically scans the columns and rows of the screen.

CRT monitors produce complex visual halos around each pixel that blur into its neighbors, and pixel art for games of the time was optimized to work with this blurring, which looks completely unlike the result of playing retro games on a modern flat-screen monitor. The arcade game *Asteroids* exploited this effect to the extreme by turning up the brightness for the bullets to the maximum, resulting in the CRT ray functioning as a kind of death ray firing right into the player's eyes—an effect that's impossible to capture in emulation.

User Input

Keyboards in the retro age were typically memory-mapped, with each key wired directly to an address in memory space to look like RAM. There would be a group of addresses together, each mapped to a key, and by loading from one you could determine if the key was up or down.

A mouse of the retro era is shown in Figure 11-14.

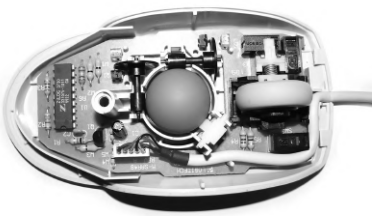


Figure 11-14: A ball mouse teardown

Mice like the one in Figure 11-14 work by physically rolling a thumb-sized rubber ball over your desk, which in turn spins two roller sensors detecting its horizontal and vertical rotations. The sensors convert the rotations into analog then digital signals to send down the wire to your computer.

Serial Port

The serial port was, and still is, a simple communications protocol (formally the RS232 standard) found on retro machines but still very relevant in embedded systems today. The core of a serial port is two wires, called RX and TX, which stand for *receive* and *transmit*, respectively. These use digital voltages over time to transmit 0s and 1s, so there's one wire sending information one way and another wire sending information the other way. A historical serial port has many other wires as well, as in the old days they were used as controls for many things, but nowadays we tend to use only RX and TX. Serial port connectors still have extra, mostly unused, pins from this history, as shown in Figure 11-15.



Figure 11-15: A traditional serial port connector

Serial ports can run at different speeds. They may also use different conventions for error checking, which can add extra redundant bits, and for *stop bits*, which show where the boundaries of characters lie in the stream of 0s and 1s. You have to make sure the device at one end of the wire is using the same speed and conventions as the device at the other end of the wire.

MIDI Interfaces

MIDI (Musical Instrument Device Interface, as seen in Figure 11-16) has been the standard bus for musical keyboards, synthesizers, samplers, and 1980s keytars to communicate real-time, symbolic musical input and output since its standardization in 1983. It's an early example of a bus hierarchy, in which an optional MIDI interface could be connected to the main bus; it also provides a secondary MIDI bus for multiple musical devices to communicate along.

MIDI connections are composed of a pair of unidirectional buses. One is for the manager to send messages to devices, and the other is for devices to send messages to the manager. They're buses in the sense that all devices use the same physical wires and can see all the messages on these wires, and so the devices must look out for which messages are addressed to them to read, and act on only those.



Figure 11-16: A MIDI connector

Each direction's bus has its own connector and runs on three physical wires. (In fact, a standard MIDI connector has five pins, with two spare to help with related work such as supplying "phantom" power to devices.) One wire is 5 V, one is ground, and one is UART (universal asynchronous receiver-transmitter) data. The bus nature of the wire from the manager to the devices is seen in the MIDI specification that all devices have three sockets: "in," "out," and "thru," where "thru" relays all "in" messages to the next device in a daisy-chain wiring scheme; other hardware adapters can merge the "out" messages from several devices onto a single wire, which is a rarer thing to want to do. As a 1980s standard, all messages are 8-bit words (known as "MIDI

bytes"), transmitted similarly to a serial port connection at a standard rate of 31.25Kbps.

MIDI, including recent extensions in MIDI 2.0, is still with us today.

Summary

For readers of a certain age, understanding and programming golden age machines can be a beautiful way to relive their youth and understand what was really going on inside their old machines. But for everyone else, these machines are still valuable to study because they bridge the gap between the simplest electronic computers, such as the Baby, and what you actually have on your desk and in your pocket today. Those modern machines have many more features that can be overwhelming, so by practicing on older machines of increasing power, you can build your confidence. To this end, this chapter studied an 8-bit system, the Commodore 64, and a 16-bit system, the Commodore Amiga. The two are related through their CPUs' common ancestry, meaning they share some instructions and styles. Many of the ideas introduced by these classic systems are still in use today, as we will see in the next chapters.

Exercises

6502 Programming

1. Easy6502 is an open source 6502 emulator that runs in your browser. It's written in JavaScript by Nick Morgan, the author of *JavaScript for Kids* and *JavaScript Crash Course*, also available from No Starch Press. Download Easy6502 with:

```
> git clone https://github.com/charles-fox/easy6502.git
> cd easy6502
```

2. Open the downloaded *emulator.html* in your browser to run Easy6502. Then enter and run the sample 6502 programs from this chapter. The emulator shows the content of the registers on the right.
3. Try writing a 16-bit multiplication subroutine in 6502 assembly using Easy6502.
4. Nick's own tutorial can be found in the downloaded *tutorial.html*. This gives many more 6502 programming details and builds up to writing a retro *Snake*-type game. Try to learn enough to understand how this game works, then try to modify it in some way, either to change the rules of the game or to transform it into another retro game such as *Space Invaders* or *Tetris*. Code built in this emulator can be ported to the C64 or other 6502-based machines, with some extra work to replace the graphics and I/O with calls to their specific designs.

C64 Programming

1. Nowadays, we can do C64 programming and assembling on a modern machine, then just run the resulting executable machine code on a C64 emulator, such as the open source VICE emulator, which can be installed locally. To get started, install the Dasm assembler from *https://dasm-assembler.github.io*.
2. Put your assembly code in a file such as *hello.asm*. Dasm requires the following two lines to be added to the start of the file to tell it to generate an executable for the C64 rather than other machines. They must have exactly eight spaces of indent:

```
        processor 6502          ; define processor family for das
        org $C000               ; memory location for our code
```

3. Assemble your code into a C64 program (*.prg*) with:

```
> dasm hello.asm -ohello.prg
```

4. The *.prg* file can be imported into a C64 emulator, such as the online JavaScript-based emulator at *https://c64emulator.111mb.de*, or VICE. (For SID programs: some emulators, such as the JavaScript one mentioned here, have sound disabled by default, so you need to turn it on.)
5. If you're lucky enough to have access to a real physical C64 and tape drive, you can also try converting your *.prg* files to tape images (*.tap*) and then sound waves (*.wav*) using a program such as *tap2wav.py* available at *https://github.com/Zibri/C64*. Then record the *.wav* to a physical tape to load to the read machine. Try inspecting the *.tap* and *.wav* files to see the 0s and 1s along the way.

Programming a Sprite-Based Game on the Amiga

Assemble and run the spaceship code shown in the "Programming the Amiga" section on page 271 as follows:

1. Download the vasm cross-assembler from *http://sun.hasenbraten.de/vasm*. Build it in Amiga mode with:

```
> make CPU=m68k SYNTAX=mot
```

2. Use vasm to assemble your assembly program with:

```
> ./vasmm68k_mot -kick1hunks -Fhunkexe -o myexe -nosym myprog.asm
```

3. Install *amitools* for Python from *https://pypi.org/project/amitools/*. Create a disk image and write the file to it, and make the disk image bootable with:

```
> xdftool mydisc.adf create
> xdftool mydisc.adf format "title"
> xdftool mydisc.adf write myexe
> xdftool mydisc.adf boot install
> xdftool mydisc.adf makedir S
> echo myexe > STARTUP-SEQUENCE
> xdftool mydisc.adf write STARTUP-SEQUENCE S/
```

4. Download and install the FS-UAE Amiga emulator from *https://fs-uae.net/download*. Run it and boot from your virtual *mydisk.adf* disk image.

More Challenging

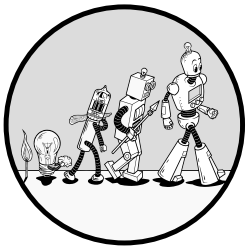
1. Research how to read the Amiga keyboard or joystick, then extend the spaceship example into a simple game using the keyboard to move the sprite around. Research how to add double buffering to remove the flicker as the screen is redrawn during the loop.
2. Building your own 6502-based computer has recently become a popular hobby. Take a look around YouTube and *https://hackaday.com* to find examples of "6502 breadboard computers" and learn how they are made. You might try doing a rebuild of one of these existing designs, or making your own design.

Further Reading

- To read the book that taught a generation of 8-bit programmer kids, see Lisa Watts and Mike Wharton, *Usborne Introduction to Machine Code for Beginners* (London: Usborne, 1983). The book is now freely available online at *https://archive.org/details/machine-code-for-beginners*.
- For a guided tour of the C64 system from 1983, see J. Butterfield, ed., "Commodore 64 Architecture," *Computer!* 32 (January 1983): 208, *https://www.atarimagazines.com/compute/issue32/112_1_COMMODORE_64_ARCHITECTURE.php*.
- For information on 8 bit–era audio programming, see James Vogel and Nevin Scrimshaw, *The Commodore 64 Music Book* (Boston: Birkhauser, 1983), *https://archive.org/details/The_Commodore_64_Music_Book/page/n3/mode/2up*.
- See *https://github.com/emu-russia/breaks* for a 6502 and NES rebuild in LogiSim.

12

EMBEDDED ARCHITECTURES



Computers are now common inside cars, robots, factories, art galleries, and domestic appliances. These environments bring particular constraints and challenges to computation, and architectures designed for them are known as *embedded systems*. The vast majority of manufactured processors—about 98 percent of them—go to embedded systems. This is a huge market, with a value of around $250 billion in the early 2020s, so it's worth your time to study these systems.

This chapter will give you the understanding of embedded systems needed to build your own robots, home automation hacks, electronic musical instruments, or art installations, as well as industrial applications. We'll begin by examining key differences between general-purpose computers and embedded systems, including the structure of typical microcontrollers and their I/O features. We'll then turn to Arduino, the most common embedded system used by computer scientists, and show how to program it in simulation and for real at the assembly language level, where its architecture is clearest to see. Finally, we'll explore some alternatives to Arduino, including Arduinoless AVR, PIC, DSPs, and PLCs.

Design Principles

There are several well-known design principles that distinguish embedded systems from other architectures. Let's walk through them now.

Single Purpose

Unlike PCs, embedded systems are usually purchased and used for a single purpose. An embedded system controls your robot or your washing machine by running a single program, meaning you don't need an operating system to switch between programs, and you don't often—or ever—need to change the program. As a result, embedded devices can be difficult to upgrade. You can occasionally try asking all your users to upgrade the firmware on their TV or music player, but it would be very expensive to promote and explain the concept widely enough for many users to actually do it. Instead, it's common for most users to throw away such devices and buy new ones. Depending on your point of view, this can be a huge waste of Earth's resources or a highly profitable business model.

Reliability

Reliability is often much more of an issue for embedded systems than for general-purpose computing—it can literally be life or death. Consider a heart pacemaker and its embedded systems, which are put inside a human being during surgery. You have to be very sure that it works correctly, as you really don't want to have to open the person up again to fix a bug or turn the device off and on again. Other embedded systems control heavy machinery in factories, signaling systems for public transportation, and nuclear missile launches, all of which have a similarly low tolerance for errors.

Mobility and Power

Embedded systems are usually designed as the computational parts of physical machines, which constrain their physical shape more heavily than for general-purpose computers. It's common for the physical machine to be designed first, and for the embedded system to be designed to fit into whatever space is left. Some embedded systems have mobility concerns, too: if the embedded system has to go on a person, for example, it has to be small and light enough to carry around (and it wouldn't hurt to look good, too).

There are considerations of electricity, especially if the host machine runs on a battery instead of plugging into the wall. Designers have to consider how much power to draw and for how long, and how large the battery has to be. A lot of effort goes into designing embedded processors to use as little energy as possible.

Encapsulation

Because they're intended for a single purpose, embedded systems typically don't need to expose the user to most or any of their functionality, a concept known as *encapsulation*. Instead, the user might get a simple interface, with just a few buttons and some LEDs, or none at all if the system is intended to work without human intervention. Often, the user won't even realize that there's a computer present in their machine.

Careful Debugging

While finished embedded systems are often designed to be very robust, safety-critical, and fault-tolerant, you'll find as a computer scientist that they can feel very brittle during development work. We're used to working with systems that can be quickly and safely hacked around; if something doesn't work, we fix it and run it again until it does. But when you work with embedded systems, a failure can physically destroy a component that may be difficult, expensive, or time-consuming to replace, so you often have to be more careful and organized about how to plan tests.

Microcontrollers

A *microcontroller* (aka a *microcontroller unit*, *MCU*, or μC) is a chip including a CPU that's designed and marketed for embedded applications. A microcontroller may look like the one in Figure 12-1.

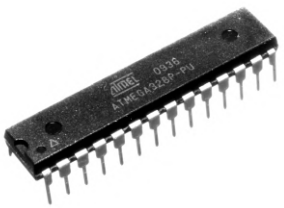


Figure 12-1: An Atmel ATmega328P microcontroller chip

In the next few sections, we'll walk through some common features of microcontrollers.

CPU

Microcontrollers are based around a CPU. The CPU is usually much lower power than a desktop's, in terms of both computational power and energy consumption. They're often 8-bit, behaving quite similarly to retro 8-bit architectures, and they often don't have floating point—as with retro machines, you need to work in either integers or fixed point.

Microcontrollers usually also include memory and I/O components on the same silicon as the CPU. This arrangement removes the need for external buses and reduces the number of pins on the microcontroller. It's easier to build a physical system from a single MCU chip than to require separate chips and bus wiring.

Memory

Because they're intended to run a single, fixed program, microcontrollers typically use a Harvard architecture, with the program stored as firmware in ROM, and RAM used only as working memory for the program's data. Using ROM in this way enables the program to remain in memory when the system is powered off, and to be immediately available when powered on again. Like all CPUs, microcontrollers are designed to fetch from a hard-wired initial address on power-on, and the first instruction will be placed in ROM at that address.

Because it has to fit on the single chip, microcontroller memory is much smaller than in desktop PCs.

Timers and Counters

As many real-world control tasks need to operate based on time and events in the real world, it's common for timers and counters to be included in microcontrollers. They usually appear as extra simple machines, with their own dedicated registers and instructions, in the microcontroller's CPU.

You saw how to make counters from digital logic in Chapter 6. If you connect a wire from the outside world to a counter, you can use the counter to count the number of occurrences of some physical event, such as the number of presses of a button.

A *timer* measures the amount of real time that has elapsed since it was initialized. Real-world time is often called "wall-clock" time in this context, as in the time difference that would be reported by a human looking at a physical clock on the wall. A timer can be made by connecting the electronic clock, as used to control the CPU's cycle, to a counter.

A *watchdog* is a special timer that automatically resets the microcontroller in the case of failure. This is used in systems that have to be reliable in the real world. If something goes wrong, you need a way to reset the system without needing to touch the machine (think of the pacemaker example). The reset is done at the digital logic level and isn't part of the CPU's program.

Embedded I/O

Embedded systems exist to control physical devices, so I/O is particularly important. We often find I/O modules, ports, and some very basic, slow serial communication built into the chip itself. As microcontrollers don't

expose their bus on their external pins, the scarce resource of pin real estate can instead be used to expose I/O connections. Some microcontrollers forgo I/O modules and use direct I/O instructions to talk to these pins—similar to what you saw for the Commodore 64 6510.

I/O isn't only important for real-time execution; it also provides a way to upload programs to embedded systems. Unlike with PCs, it's usually not possible to do the development work on an embedded device, as this would require graphics, a keyboard, an operating system, and a compiler to all run on the low-power device. Instead, we do development work on a desktop, and perhaps test our programs there too using simulation or emulation, before transferring the final binary executable to the embedded device. Microcontrollers have special modes for doing this: usually they can be connected to a desktop via USB, serial port, or other means, then put into "firmware upgrade" mode to copy the executable into their non-volatile program memory via this connection and a software device driver on the desktop machine.

Analog-Digital Conversion

Many microcontrollers need to handle incoming and outgoing analog signals, but inside the controller signals must be digital; this requires conversion at both ends. The necessary converters may be found outside the microcontroller, connected to its pins, or in some cases on the microcontroller silicon itself.

The classic case of analog-digital conversion (ADC) is audio processing. An analog signal from a microphone is sent to a digital processor, which adds effects to the audio before sending the processed analog signal back out to the speakers. This is done by taking a continuous analog signal wave and quantizing it, turning it into a digital signal by sampling it at regular time intervals, as shown in Figure 12-2. You can do this at different resolutions by taking samples more or less frequently.

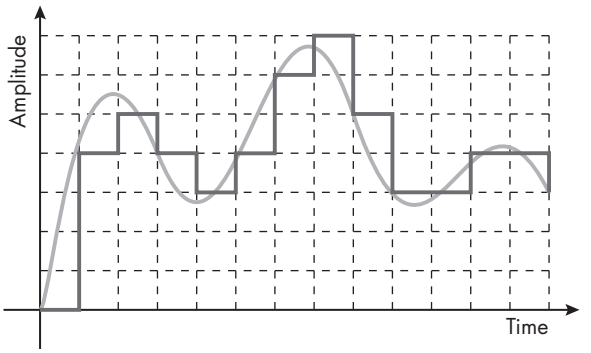


Figure 12-2: Quantizing an analog signal to digital

When converting the other way, digital-analog conversion (DAC), some devices (like the Arduino Due) do true conversion of digital integers to analog voltages. Cheaper ones (such as the Arduino Uno) approximate the conversion with pulse-width modulation (PWM). Here, the output is only ever

0 V or 5 V. If 3 V is asked for, the output oscillates rapidly between 0 V and 5 V, spending three-fifths of its time at 5 V and two-fifths at 0 V, to give a temporal average of 3 V. For some applications this creates no noticeable difference, but for others it can play havoc with the output.

Embedded Serial Ports

The serial ports seen in the previous chapter continue to be used heavily today in embedded systems, due to their simplicity and stability. In the projects you do here, you're more likely to see this convention in a virtualized form, as you don't often see a physical serial port on a modern computer these days. Instead, you can use something like USB to emulate the old-fashioned serial port protocols. Similarly, the *Zigbee* wireless protocol acts as a virtual serial port running over a specific radio frequency; it's used by embedded devices such as programmable light bulbs and transportation and agriculture sensor networks.

Inter-Integrated Circuit Bus

The *Inter-Integrated Circuit bus*, pronounced "eye-two-see" (written as I^2C and sometimes pronounced "eye-squared-see"), is a standard for connecting chips together. It's very common in robotics. The standard is owned and licensed by NXP (formerly Phillips).

I^2C communication is done on just two wires: data (SDA) and clock (SDL), as shown in Figure 12-3.

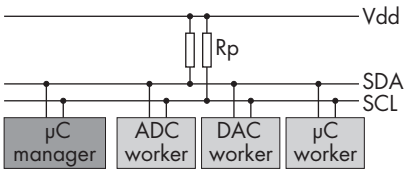


Figure 12-3: The I^2C architecture

I^2C can use 5 V or 3.3 V as its high voltage, and runs in various speed modes, from 100Kbps to 3Mbps. There may be multiple devices on the bus, each having a 7-bit licensed device address. One node must take on the role of manager by generating the clock and initiating communications. The other nodes are workers, which reply to the manager. Basic message collision avoidance is implemented by the rule "only talk if the bus is free."

In practice, I^2C devices can be accessed via a standard FTDI (Future Technology Devices International Ltd) chip, which provides a hardware and software interface to it, usually via a serial connection (which itself is usually via a USB port). Examples of an I^2C device (an inertial measurement unit sensor) and an FTDI for interfacing to it are shown in Figure 12-4.

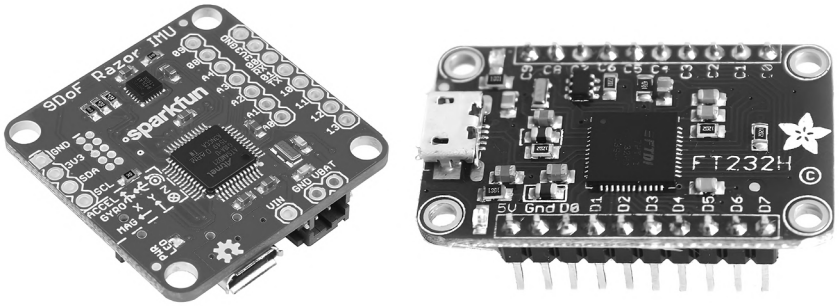


Figure 12-4: An I^2C device (left) and FTDI interface (right)

No extra device drivers are needed for transport—the FTDI behaves as a serial port to the user.

Controller Area Network Bus

A *vehicle bus* is a specialized internal communications network that interconnects components inside a vehicle such as an automobile, train, ship, aircraft, or robot. A *controller area network (CAN)* bus is a type of vehicle bus that has a single public serial channel shared by all devices. CAN has no standard connector, as it isn't intended for use by consumers, but rather for the internals of vehicles. Usually its wires are soldered directly into the printed circuit boards (PCBs) of the many devices in the vehicle. If you remove the plastic covering in front of the passenger seat in a car, you'll usually find a wiring loom, which will include accessible CAN wires. Check your vehicle's service documents to locate and connect to them.

CAN usually has four internal wires, which use differential voltages to protect against the strong external electromagnetic fields expected in vehicles, especially around electric motors and engines.

CAN security is a current concern. Because it's a bus, all devices can read and write to it. This may create problems when safety-critical devices, such as antilock brakes, are connected to the same bus as non-critical devices, such as media players. The concern is that security in media and similar devices is typically less rigorous than in the safety equipment. A hacker could take control of a non-critical device and use it to send malicious commands to critical devices, or deny service to them by filling the bus with junk messages. For autonomous vehicles where steering and acceleration are also managed via the CAN bus, the consequences could be particularly severe.

Now that you've seen the general concepts that go into an embedded system, let's explore how they show up in practice in the best-known example, the Arduino.

Arduino

Arduino, shown in Figure 12-5, is the standard embedded system for hackers, makers, and robotics researchers because it packages a microcontroller onto a PCB together with all the power management and I/O that you need to program it from a PC—you can just connect it to your desktop via USB and start programming it, without having to worry about analog power supplies or setting up its USB I/O system by yourself.

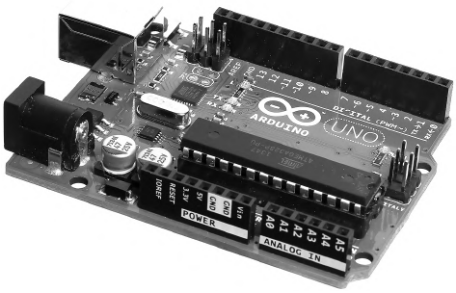


Figure 12-5: An Arduino board. The ATmega328P microcontroller is the large chip near the bottom right.

The Arduino PCB is an open source hardware design based around a microcontroller from the closed source Atmel AVR family, usually the ATmega328 model. The microcontroller is surrounded by additional hardware that makes powering and interfacing with it both easy and standard. These components were the traditional barrier to computer scientists programming microcontrollers, as they have to be made up on breadboards or PCBs for every project, requiring analog electronics skills. The cleverness of the Arduino design was to select and standardize a single set of these components that are generally useful for many applications, and to manufacture them cheaply in bulk so that end users don't have to worry about them anymore. Arduino comes with open source software to easily assemble and transfer programs via USB into its firmware. (There's also a C-like language and compiler, but as this book is about the architectural level, we'll here study only Arduino's assembly-level programming.)

You can program Arduino by itself—for example, to read numbers sent to it over USB from your desktop, do arithmetic on them, and send the results to your desktop. However, Arduino is usually used to interface with other electronic sensors and actuators, starting with LEDs and switches. Typically you lay these out on a breadboard, then connect wires from the breadboard to your Arduino, as in Figure 12-6.

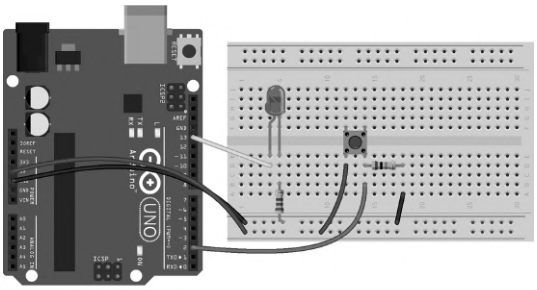


Figure 12-6: An I/O circuit connecting an LED and button to an Arduino using a breadboard and wires

No soldering is required, as components and wires can be pushed into place in both the breadboard and the Arduino's connectors.

The ATmega328 Microcontroller

The classic Arduino microcontroller, the Atmel AVR ATmega328, shown in Figure 12-1, behaves somewhat like an old-style 8-bit system such as a 6502. There are 32 8-bit user registers (more than the three of the 6502). There's an arithmetic logic unit (ALU) that includes integer multiplication and division, but no floating point. Similar to the 6502, there's an 8-bit status register, containing bits telling you the result of arithmetic calculations to allow branching. The instruction set architecture (ISA) includes indirect addressing and a hardware stack. It's usually clocked around 20 MHz.

The pinout, shown in Figure 12-7, is different from a typical CPU, as there's no external bus.

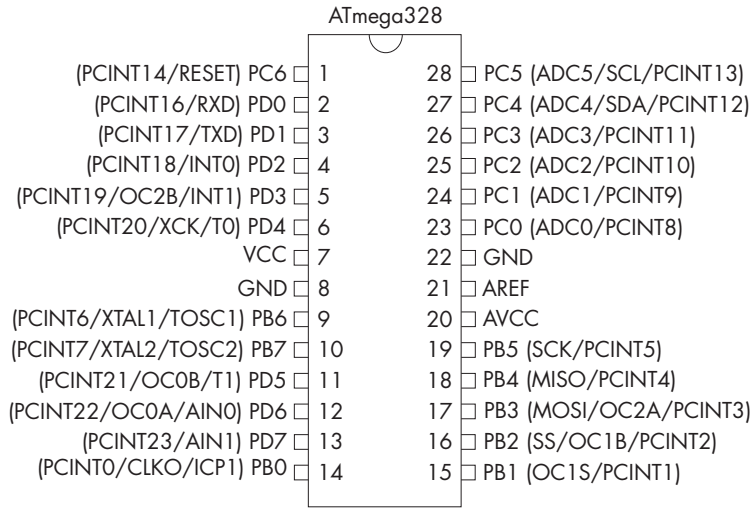


Figure 12-7: The pinout of the ATmega328 (note the lack of A and D bus pins)

Instead of an external bus, 14 I/O pins are directly exposed. Pins increase the size of the chip's package, so they're a scarce resource. The I/O pins are each configurable to function as either input or output. Their configuration is set and stored using dedicated data direction registers (DDRs).

The die shot (Figure 12-8) reveals that the microcontroller contains more than just a CPU.

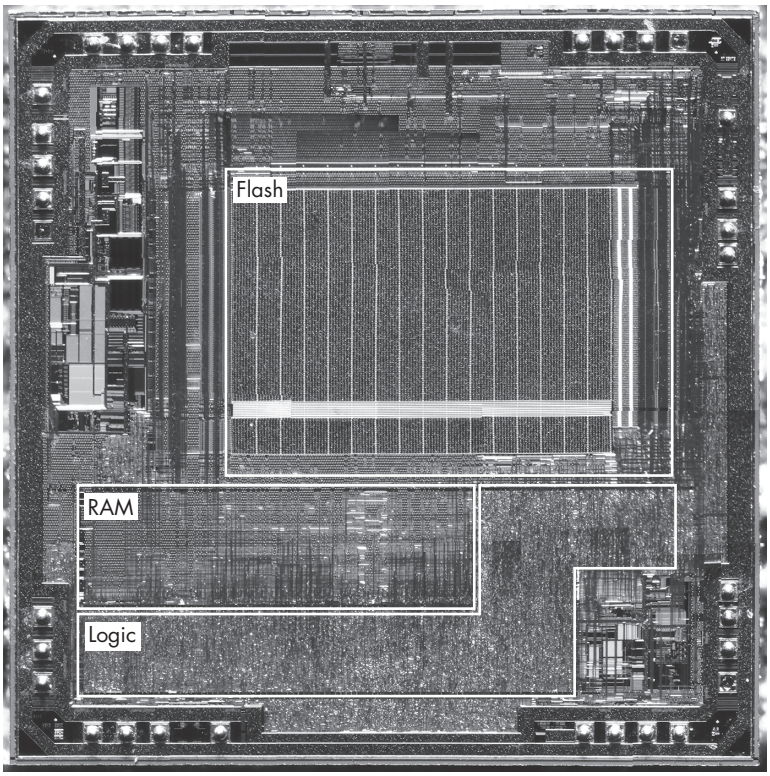


Figure 12-8: A die shot of the ATmega328

In addition to a CPU, the chip has 2 k_2B of SRAM, 32 k_2B of flash memory, and a 1 k_2B EEPROM, all on the silicon. In this sense, the chip is really more comparable to an entire retro computer than to only a CPU.

The Arduino uses a Harvard architecture. The program you send to the board is programmed into the flash memory using software on your host PC, while the RAM is used for data. (The EEPROM is user-writable, and is provided for applications where small configuration-style data needs to be stored during power-off.) The Harvard architecture uses two separate buses: an 8-bit bus for data and a 16-bit bus for programs. There's no external memory or bus; the memory is all on the chip.

The microcontroller contains serial port pins and I/O modules. On power-on, the microcontroller first runs a small internal ROM program that checks its serial port. If there's data waiting on the serial port, it's assumed to be a new user program, which is then loaded into flash. The program counter is set to start the user program.

The Rest of the Arduino Board

While it's possible to program the ATmega directly from its serial port pins, most desktop machines don't have physical serial ports anymore, so it's easier for users to use a virtual serial port running over USB. The Arduino board includes a USB connector and a dedicated chip (actually another, smaller microcontroller) that reads the USB wires and translates them into serial port signals to pass to the ATmega's pins.

Most of the analog electronics on the board are used for power management. The microcontroller requires only a simple 5 V power supply. If a single, stable 5 V is provided, then no further electronics would be needed. However, Arduino is designed to work in several different use cases. In particular, it can be powered by a battery or can take power via a USB cable. The extra components regulate these supplies, protecting the board from peaks and dips, and enabling it to switch between them. (It could otherwise be very bad to allow current from elsewhere to travel back up the USB cable into the connected desktop computer.)

An I^2C bus enables extra extensions to plug into the Arduino. You can get other physical boards ("shields") that plug into ports on the I^2C bus, in a nice, physically stackable way.

As it's an open source platform, Arduino has been modified by many designers. For example, Ruggeduino is a hardened (and thus more expensive) version that includes extra safeguards to prevent you from blowing it up in stupid ways. There are also official variants from the Arduino team. The Due is a version with real DACs replacing PWM, the Mega and Giga have larger PCBs to enable more connections, and the Nano has a smaller footprint. Some variants use different microcontrollers, providing more computing power and different instruction sets for those who need or prefer them.

Programming Arduino

Like all CPUs, Atmel AVRs execute machine code from an instruction set, which you can program by assembling from a human-readable assembly language. The Arduino assembler isn't very different from the other assemblers we've seen so far. You can write, edit, and assemble this assembly code on your desktop PC. The classic "Hello, world!" program for Arduino is to turn on its built-in LED on pin 13:

```
.global main
main:
  ldi r16,0b00100000   ; load bits describing eight AVR PB pins into r16
  out 0x04,r16         ; set AVR pin PB5 (Arduino pin 13) to output mode
  out 0x05,r16         ; set output on AVR pin PB5 (Arduino pin 13) to ON
.global loop
loop:
  jmp loop
```

The global `main` label gets called automatically when the Arduino is powered on. The `ldi` instruction is "load immediate," and it loads a constant into a register. This particular constant contains 8 bits, one for each of the AVR's eight digital I/O pins (labeled PB0 through PB7 in Figure 12-7). They're all set to 0, except for pin PB5 (counting right to left, from PB0, along the binary digits), which is set to 1. The AVR's PB5 pin is wired to the Arduino PCB's pin 13 and thus to the LED. The first `out` instruction copies the bits from r16 to 0x04, the data direction register, to configure the pins for I/O. This sets PB5 to act as an output and the other seven pins to act as inputs. The second `out` writes the same bits from r16 to 0x05, the "PortB" register, which sets values to output on the eight PB pins. This writes the 1 to PB5 and thus sends a high voltage to turn on the LED on Arduino pin 13.

Unlike many CPU programs, the `loop` label and jump are important because they keep the program running forever. Without these, the LED would light only for a fraction of a second, then go off as the program ends. Embedded programs usually need to run forever like this.

A more complex version of the program makes the LED blink on and off:

```
#define DDRB 0x04
#define PINB 0x03
.global main
main:
  sbi   DDRB, 5         ; set bit IO; port b 5th pin (make pin 13 an output)
blink:
  sbi   PINB, 5         ; set bit IO; to toggle PINB
  ldi   r25, hi8(1000)  ; 1,000 ms delay as argument, hi byte
  ldi   r24, lo8(1000)  ; 1,000 ms delay as argument, lo byte
  call  delay_ms
  jmp   blink
delay_ms:               ; delay about (r25:r24)*ms. Clobbers r30, and r31
  ldi   r31, hi8(4000)
  ldi   r30, lo8(4000)
innerloop:
  sbiw    r30, 1        ; subtract immediate value from word
  brne    innerloop     ; branch if not equal to zero status flag
  sbiw    r24, 1
  brne    delay_ms
  ret
```

To make the code easier to read, I have here defined DDRB and PINB to represent the data direction register and PortB register. One millisecond is about 16,000 cycles at 16 MHz. The inner loop takes four cycles, so we repeat it 3,000 times.

NOTE *The AVR also has some 16-bit instructions that operate on pairs of 8-bit registers together, as in the 6502.*

Other CPU-Based Embedded Systems

Arduino isn't the only CPU-based embedded system in town. Let's look at some alternatives that you might encounter.

Atmel AVR Without the Arduino

Arduino is designed for computer scientists, not engineers. You wouldn't normally sell a product based on a full Arduino board. Rather, you would create a custom PCB containing the AVR chip plus only the electronics that are needed, both from the Arduino board and from your own design.

As an intermediate step, you can use a breadboard without the Arduino to mount the AVR and other electronics, as in Figure 12-9.

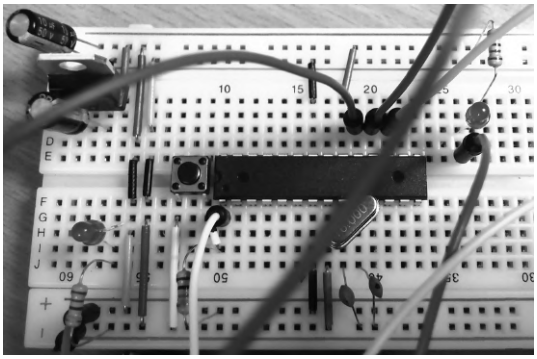


Figure 12-9: A breadboard implementation using an AVR microcontroller

Once you're happy that your design is working, you turn it into a PCB design using a program such as *KiCAD*, submit it to a PCB manufacturing company using their website, and receive your PCB in a few days through the mail. You don't have to solder things yourself nowadays, the PCB manufacturing companies have robots that do it for you.

PIC Microcontrollers

PIC is another series of microcontrollers, similar to but different from the AVR series. As with Arduinoless AVR, PICs require breadboards, PCB design, and serial ports.

PICs are designed by the American company Microchip, who bought out their competitor Atmel in 2016. PICs are found in many consumer and industrial embedded systems. There are a number of PICs to choose from; you decide which to buy based on your needs in terms of speed, power, cost, and physical size. Because of the wide range of options, PICs are more popular than Arduino/AVR in production engineering. The flexibility allows a selected PIC version to be closely matched to its application needs.

Digital Signal Processors

Digital signal processors (DSPs) are a specialized class of microcontroller designed for handling real-time signals, such as audio. Embedded systems working with such signals have particular requirements, as they're fundamentally working with long—effectively endless—real-time streams of identically formatted data that have to be processed repeatedly in identical ways. This means there isn't much branching; instead, the data flows through a smooth pipeline from one stage to the next, always being processed in the same way.

For example, guitarists often buy and use digital effects boxes that connect between their guitar and amplifier to modify the sound (by adding compression, distortion, delay, or reverb, for example). These boxes are embedded systems containing one or more DSPs, such as the chip shown in Figure 12-10.

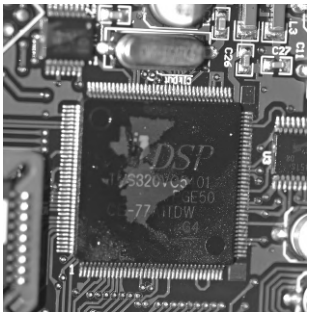


Figure 12-10: A DSP chip inside a guitar digital effects unit

DSPs aren't only for audio signals, though. There are many other types of signals with similar properties, such as video, radar, and data streams from all kinds of medical and scientific monitoring instruments. With sound and sound-like data, you can often mostly get away with representing directly quantized sound waves from the ADC. With video, however, the data is usually so huge that it needs to be compressed during storage and transmission, which means many DSP units are used primarily to perform compression and decompression.

DSPs often use fixed-point number representation (as discussed in Chapter 2) rather than integers or floating point. This is because most signals have clear, fixed upper and lower bounds that can be rescaled to +1.0 and −1.0. For example, musical audio is usually recorded in this way, with any signal outside these bounds being clipped. Fixed point is cheaper and simpler to implement than floating point, but can give similar quality results for these kinds of signals.

DSPs use their available silicon to provide additional instructions dedicated to signal processing. (Recall from Chapter 8 that adding extra domain-specific instructions like this is often considered to be the CISC philosophy.)

For example, special instructions for fast Fourier transforms and convolution are found in DSPs designed for embedded audio use, as these operations form the basis of many standard audio processing algorithms. These usually operate in fixed point. As DSPs are designed to process large streams of data, they sometimes include additional instructions that load and store chunks of data larger than single words. Such instructions may trigger a sequence of transfers over the bus from a series of neighboring memory locations to a group of registers. Similarly, I/O instructions may trigger a series of ADCs to and from these groups of registers.

As with standard microcontrollers, DSPs use a Harvard architecture, so that firmware can be placed in ROM during manufacture, then left alone to run forever.

Embedded Systems with No CPU

The embedded systems we've seen so far have been microcontroller-based, meaning they're still based around a CPU that executes programs of machine code instructions. But there are also other, simpler styles of embedded systems where there's no CPU, no program, and no instruction set. There's only hardware that you lay out to compute what you want to be computed, using digital logic circuits. These systems include PLCs and FPGAs.

Programmable Logic Controllers

Programmable logic controllers (PLCs) are a type of embedded system designed to perform simple computations to control machinery in industrial environments, with very high reliability. They're usually found in factories with dust, chemicals, bits of food, high and low temperatures, and other extreme conditions that make life hard for normal chips. The idea is to install something durable that can operate continuously for 20 years, without ever going down. Systems have to be almost indestructible, utterly reliable, and as simple as possible to avoid any kind of bugs slipping in. In this context of industrial automation, embedded systems are sometimes known as *supervisory control and data acquisition (SCADA)* systems.

You'll see PLCs in these kinds of environments, usually packaged in what are called DIN modules and mounted on standard DIN rails, as in Figure 12-11.

In your house—often in a basement or under the stairs—you might have a DIN-style module that acts as a fuse box or circuit breaker (aka a residual current device, or RCD) for your whole house. Again, it's robust engineering that's designed not to fail under any normal operating circumstances. The DIN design was standardized in the 1970s and is still with us today.

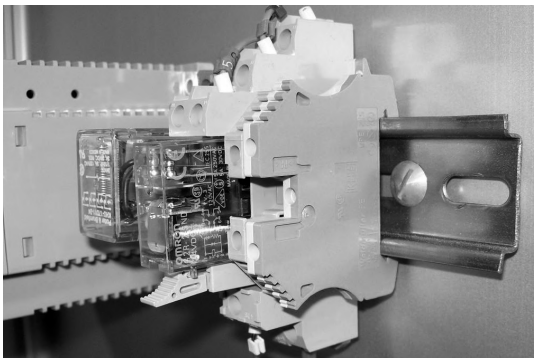


Figure 12-11: DIN modules mounted on a DIN rail

A PLC doesn't run a program in the sense of a series of instructions; instead, its function is usually specified using a visual system called *ladder logic*, as shown in Figure 12-12.

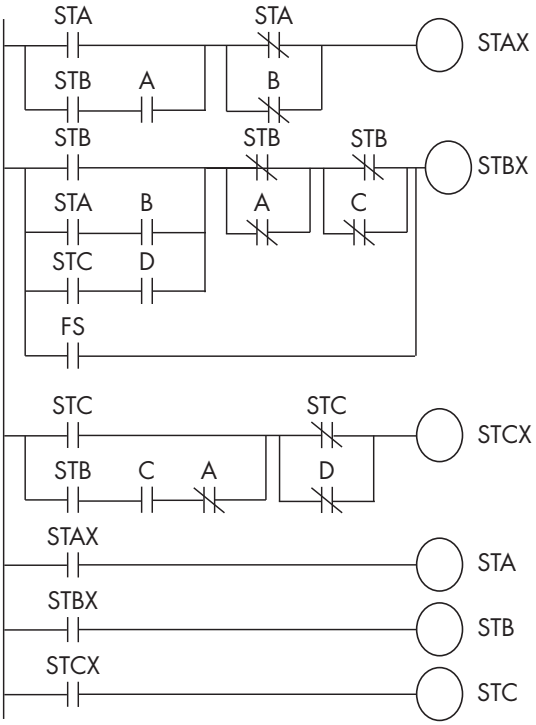


Figure 12-12: An example of a ladder logic configuration for an embedded device

In essence, ladder logic is a set of if-then rules that say that if one input is high, then connect a wire to another wire. There's no program starting at the top and working through a list of instructions; each unit follows the logic of the rules around it. It's derived from the old days where you'd make computers from physical electromechanical relays.

Ladder logic is so simple that even engineers can do it. But the simplicity also means that both formal methods and intuitive inspection can be used to verify that systems do exactly what they're supposed to do. You wouldn't want all the complexity of an operating system and modern programming languages and compilers when even tiny bugs in any of that could cause your nuclear fuel rods to move to the wrong place; everything has to be absolutely reliable and understandable.

PLCs are simple, completely transparent, and verifiable. It may surprise you to learn that the engineers programming this equipment often make more than most computer scientist programmers, but they're also getting paid to take responsibility for safety. The program might exist as part of a nuclear power station where, even if the program is very simple, it has to be very correct. You don't have to design PLCs by directly configuring the ladder logic anymore; there are now compilers and assemblers that will turn C code into these configurations for you. Doing this, of course, requires trusting the compiler and assembler programs, as well as your own code.

EMBEDDED SECURITY

SCADA systems should never be connected to the public internet. A famous leading question used in security audits asks SCADA managers, "How would you connect to the system in an emergency, when all the staff are away working at home and they need to take over control remotely before the nuclear fuel rod goes critical?" A worrying number of managers will then proudly explain that they do in fact have such a connection, which of course can be exploited by hackers.

Even when there's no internet connection and the system is separated from the network by an "air gap," it is possible to gain access. The 2010 Stuxnet worm was distributed on USB sticks left around international academic conferences. It replicated itself from USB stick to USB stick around the world until it reached the Iranian nuclear weapons fuel enrichment centrifuge embedded systems. Stuxnet then affected only their specific model and configuration of PLC, altering its behavior very subtly and almost undetectably to change the timing of the centrifuges, destroying them and preventing enrichment of the fuel.

Embedded FPGAs

The FPGA (field programmable gate array) chips discussed in Chapter 5 can be used to actualize any digital logic designs—not only those intended for use in or with CPUs. This can include PLC-like structures and many other digital logic network designs.

As embedded systems perform single functions, a CPU design capable of running arbitrary programs of instructions may be both overkill and inefficient. Instead, the particular sequence of arithmetic or other transformations can be implemented directly as a sequence of simple machines, pipelined together in an FPGA. This can include, for example, placing multiple adders and multipliers connected in the specific sequence needed to

implement your signal processing algorithm. In addition to reducing the CPU-style paraphernalia needed, this can also make systems run very fast, as all these arithmetic operations can happen in parallel.

Hardware description languages can be especially useful for creating such designs. For example, they enable the arithmetic steps to be expressed in a C-like language before being automatically compiled to the appropriate digital logic.

UBIQUITOUS VS. MINDFUL COMPUTING

Ubicomp, or Ubiquitous Computing, is an embedded design philosophy founded by Marc Wiser in the 1980s Xerox PARC (the same place where the mouse and graphical desktop were invented). Its core idea, as outlined by Marc Weiser, is that "the purpose of a computer is to help you do something else. The best computer is a quiet, invisible servant. The more you can do by intuition the smarter you are; the computer should extend your unconscious. Technology should create calm."

Ubicomp shows up in products such as Amazon's Alexa. It sits invisibly in your house, and when you want something, you say it out loud and it gets done for you. There's no need for you to sit down at a computer and think about how to do it. Ubicomp ideas have also returned in recent fields such as "pervasive computing" and the Internet of Things.

There has recently been a counter-movement against Ubicomp, which we might call *mindful computing*. Its adherents have decided that users don't want decisions made for them by an uncertain, non-understood corporate cloud. They're freaked out about losing control to these machines. Mindful computing therefore does the opposite, deliberately drawing attention to the technology and forcing users to think about and understand the machines they're using.

According to a Ubicomp philosophy, light switches might disappear as machines automatically predict when the lights should turn themselves on and off without your input. According to mindful computing, the light switches should remain, and the user should devote their full conscious attention to becoming at one with the light switch as they touch it.

Summary

Embedded architectures form the vast bulk of the world's computers, yet by their nature they're often invisible to most users. Their applications exist at the border between computing and engineering, but their architectures can be quite similar to those of retro computers, and they provide an interesting place for fans of that style of computing to work today. Most embedded systems are based on microcontrollers, which are chips that combine a low-power CPU with onboard memory, I/O, and other useful features. Arduino is a standard embedded platform that wraps most of the engineering needed for computer scientists to get started interfacing to hardware such as robots, factories, cars, and art installations.

Exercises

Simulated Arduino Programming

1. Use the open source Wokwi Arduino emulator to run the example Arduino programs shown in this chapter. To use the assembler, go to the blink.S tab at *https://wokwi.com/arduino/projects/290348681199092237*, or find the offline version at *https://github.com/arcostasi/avr8js-electron-playground*.
2. Remember that Arduino's I/O pins are each configurable to act as inputs or outputs. If a pin isn't reading or writing as expected, check that you've put it into the right mode first.
3. Instead of the nested delay loops used in the LED blinking program, a prettier and more energy-efficient way to program blinking lights is to use the AVR's built-in timer. Research what registers and commands are needed to do it this way, and implement this alternative version.

Challenging

1. There are many affordable Arduino starter kits available; obtain one and try to run the example programs on the real hardware. When working with real LEDs, remember that as diodes they're directional and must be connected the right way around and always in parallel with a resistor; otherwise, they'll explode!
2. Most kits come with some additional sample programs written in Arduino C; try to reproduce their functionality using your own handwritten AVR assembly. (If stuck, try compiling the Arduino C into assembly and inspecting that to get ideas.)
3. If you prefer command line tools to the Arduino IDE, AVRA is the AVR assembler, and AVRDUDE is the AVR downloader/uploader.

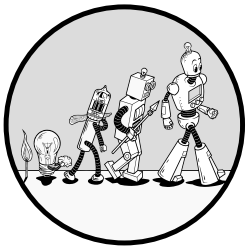
Further Reading

- For a famous parable about engineers' and computer scientists' differing opinions on embedded design, see Do-While Jones, "The Breakfast Food Cooker." Various versions can be found around the internet, dating from 1990.
- For a full reference for the AVR instruction set, see Atmel, "AVR-Instruction-Set-Manual," 2016, *https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf*.
- For an Arduino PCB design explanation and CAD files, see the "Arduino from Scratch" series, *https://rheingoldheavy.com/arduino-from-scratch-series*.
- For information on programming PIC microcontrollers, see "PIC Programming in Assembly," *https://groups.csail.mit.edu/lbr/stack/pic/pic-prog-assembly.pdf*.

- To get your hands dirty with a CAN bus, see Jared Reabow, "How to Hack and Upgrade Your Car Using CAN Bus and Arduino," *https://www.instructables.com/How-to-Hack-and-Upgrade-Your-Car-Using-CAN-Bus/*. The tutorial includes instructions for making a *Back to the Future*–style date and time display.
- Hackaday (*https://www.hackaday.com*) is a well-known website for embedded project ideas.

13

DESKTOP ARCHITECTURES



"A computer on every desk" was Bill Gates's ambition during the 32-bit era of the 1990s, and while the current trend is toward the Internet of Things and the cloud, a personal computer (PC) can still be found on many desks and laps today. The PC isn't a single computer design; rather, it's a set of loose conventions for combining many different components from different manufacturers into computers, based around the x86 family of CPUs.

Thanks to a business-led focus on backward compatibility, modern PCs retain many features from earlier stages of their evolution, so in this chapter we'll study how these conventions came into being and how they've affected x86 architecture and PC computer design. We'll examine x86's CISC philosophy and its Silicon Valley history and instruction set, then look at some computer design elements used to build modern PCs around it.

CISC Design Philosophy

Most desktop computers use CPUs from the x86 family, which are usually described as CISC architectures. We've seen CISC architectures a few times, but let's take a closer look at some of the CISC principles that appear in x86.

In a CISC architecture, you try to do as many big and clever things as you can on a large, complex chip with lots of silicon. You design many different small machines that all do different specialized things; you also provide dedicated instructions for each of them. As you can imagine, this is very hard to design, and you end up having to pay your architects a lot of money—especially when all the new complex features need to be made to play nicely with other innovations, such as pipelining and out-of-order execution (OOOE). Using lots of silicon typically consumes lots of power, so CISC processors often have to be plugged into the wall, with heavy power transformers and large cooling systems such as fans. These requirements are easier to meet in a desktop setting than in embedded and smart-type environments.

A classic aspect of CISC philosophy is having lots of instructions that combine memory access with arithmetic logic unit (ALU) instructions, such as "multiply the contents of a first address by the contents of a second address and store the result in a third address," where the addresses are in RAM. This is, in fact, a compound instruction involving many steps: we need to load both addresses, multiply their values, put the resulting value in a register, and store it in memory again.

CISC also emphasizes implementing new instructions in hardware essentially saying, "Throw more silicon at the problem." For example, if users demand lots of video codec streaming, you can create special instructions that perform the specific mathematical operations used in video codecs, and build lots of new simple machines in digital logic to implement each of them.

A "decode my video" instruction is going to take more than one clock cycle, and accommodating different instructions that take differing amounts of time is a major challenge that arises in CISC architectures. In particular, pipelining and OOOE are harder to get right when instructions have different durations. This problem can be fixed by throwing even more silicon at it: you can create even more complex digital logic in the control unit (CU) to identify these durations and schedule around them.

One supposed advantage of CISC architectures is that the compiler has to do very little work to translate common high-level language statements into assembly; this is because the instruction set architecture (ISA) has dedicated instructions for commands such as "decode my video," which then have a simple one-to-one translation. But these instructions make life harder for compiler writers, who now need to wade through a five-volume set of instructions for *every* backend CPU they target; they're also now expected to make some attempt to optimize their compiler for each particular ISA. It would be much easier for compiler writers to just use one volume of instructions and ignore all the advanced ones. In practice, this means that CISC architectures are more likely to come with compilers written by the same

people who built the CPU, because no one else wants to work to optimize for one particular CPU. These compilers tend to be proprietary and to run faster than the open source versions due to the complexity involved; only those who built the system fully understand all the features.

Another upside is that assembly programs can be short, as every instruction does a lot of work. In the 1980s, this was important: RAM was limited, so shorter programs freed up more RAM for data. It's not so important today.

CISC was invented by an Englishman, Maurice Wilkes, seen previously in Figure 1-19, but was commercialized by Americans. Stereotypical CISC architects and users are business-driven, and CISC is dominant in real-world desktop computing. You're probably using a CISC architecture on your desktop today. If a CISC client asks for a new instruction to speed up their particular multimedia application, then the CISC business will often design and add it for them—for a cost. New features are often bolted on in this way, without necessarily being designed to beautifully fit together with what was there before. The older features will usually be retained, however, in order to avoid breaking other customers' existing systems.

Microprogramming

Building new CPUs in hardware is hard and expensive. A chip mask set costs around $5 million to make, and if you get it wrong anywhere, new masks will be needed. This problem is acute for CISC due to its complex designs. *Microprogramming* is a solution to this problem in which the architecture consists of many simple machines that can be connected and disconnected through basic switches. Instructions are then defined as sequences of connections and disconnections. For example, to add two registers, you first connect one of them to an ALU input, then connect the other register to the other ALU input. Then you connect the ALU to a signal asking it to add, and finally you connect the result in the ALU output to a register.

This idea is reminiscent of the rotating barrel CU in Babbage's Analytical Engine. The barrel has pins that are placed to trigger sequences of the simple machines. If the pins are moved around, different instructions and architectures can be easily created. Modern electronic microprogramming—and hence CISC—is credited to Wilkes, who studied and taught the history of computing and was very open about having picked up the idea from Babbage's mechanical barrel. This is a paradigmatic example of how studying the arc of history can enable major, Turing Award–winning advances in modern architecture.

The electronic version of Babbage's barrel pins is usually firmware, known as *microcode*, inside the CPU, containing a list of connections to make and break in sequence for each instruction. (This isn't ROM in the CPU's address space, it's a non-addressable, separate region inside the CPU itself.) As firmware, it can be electronically reprogrammed at any time. This massively reduces the cost of fixing hardware bugs in the CPU, as they can be

corrected with a firmware update rather than having to return and remanufacture the chip itself.

Microprograms aren't machine code programs; they exist at a lower level, defining the machine that the machine code runs on. The actions of microprograms can be notated using register transfer language (RTL), as in Chapter 7. Modern CISC chips may have many thousands of complex instructions all defined in microcode. You can re-microprogram your CPU to implement a completely different instruction set if you like, such as turning an x86 into a retro 6502! There's now so much reconfigurability that microprograms can behave almost like FPGAs.

Now that we've seen some of the design concepts, let's turn to the history of x86. Doing so will help you make sense of features still present in modern x86s that have accumulated through this history.

x86 History

The x86 architecture has been the most commercially successful and resilient CPU architecture to date, reaching its 45th anniversary in 2023. x86 is a family of CISC architectures whose designs and names derive from the model numbers of the first few generations of Intel processors: 8086, 80286, 80386, and 80486. x86 has persisted across three generations of word lengths: 16-, 32-, and 64-bit architectures. As a commercial product, it has strongly emphasized rigorous backward compatibility with all previous generations, at the cost of adding complexity to the design, including digital logic to ensure historical bugs are kept in order to allow old games that exploit them as features to continue to run. You can still take your executable machine code from the 1970s and run it on a modern x86 and it will "just work." (This is a similar approach to software design in commercial operating systems, which similarly grow to huge, bloated sizes to maintain compatibility for customers at the expense of performance and beauty.) As a result of continually adding new CISC instructions and keeping all the old ones, the latest version of x86—the *amd64* ISA—now includes over 3,000 instructions, documented in a five-volume set of reference books.

Prehistory

The history of x86 design is one of Silicon Valley architecture and politics, and specifically of the companies Intel and AMD. Both companies make processors using the same proprietary instruction set, and they're constantly locked in legal battles with each other, which have now spanned decades.

William Shockley, John Bardeen, and Walter Brattain were awarded the Nobel Prize in Physics in 1956 for their invention of the transistor at Bell Labs, New Jersey. Shockley's family was from Palo Alto, California, though he was born in London. After winning a Nobel Prize, you can live and work wherever you like, so Shockley decided to relocate from New Jersey to Mountain View, California, because he wanted to be near his mother

in Palo Alto. He set up Shockley Semiconductor there to continue his transistor research and commercialization.

By 1957, Shockley had become a difficult person to work with due to a mixture of Nobel laureate hubris and obsession with topics considered fringe by his staff. A group of employees, the so-called "traitorous eight"—including Gordon Moore and Robert Noyce—walked out on Shockley to set a rival firm, Fairchild Semiconductor. This was considered almost blasphemous by the commercial culture of the time, in which it was assumed people would join a big company and be loyal company servants for their whole careers. It has since become the blueprint for Silicon Valley's startup culture, in which it's assumed employees will and should leave big companies to start their own.

Fairchild created the first commercial version of the integrated circuit (chip). Demand for computing at this time was almost entirely from the American military, which used taxpayer money to subsidize research and buy the products of chipmakers to power missiles and planes for the Cold War. These government funds fed the silicon industry, accelerating the growth of Fairchild and also many rival upstarts as Fairchild staff copied the Fairchild model and left to start their own competing chip companies, giving rise to modern Silicon Valley.

In 1968, Fairchild politics led Gordon Moore and Robert Noyce to quit again—this time leaving Fairchild to set up Intel (short for Integrated Electronics). AMD (Advanced Micro Devices) was founded the following year by Jerry Sanders. AMD's early goal was to copy Intel's products and produce them more cheaply as a second source. Before the x86 series proper, Intel produced the 4-bit 4004 in 1971. AMD cloned it shortly afterward in 1975 as the Am9080. Intel preempted this in 1974 with an 8-bit version, the 8080 (3 MHz), which was then also copied by AMD.

16-Bit Classical Era

The first member of the x86 family proper—defined by modern backward compatibility—was Intel's 16-bit, 5 MHz 8086 chip, made in 1978. This was a CISC chip that used microprogramming. x86 is named after its last two digits.

Competition between Intel and AMD became formalized in 1982 by a three-way contract between Intel, AMD, and IBM, whose business at the time was building computers. IBM wanted to buy CPUs for its computers but didn't want to be locked into using a proprietary design from a single company, because such a company could then hold IBM to ransom via the lock-in and increase its prices. As a huge company, IBM had enough buying power to play suppliers against one another to get what it really wanted, which was for more than one company to compete to produce the same chips as generic commodities; this would push down the prices and enable IBM to get them cheap in perpetuity. IBM said to Intel, "We want to buy your chips, but we'll buy them only if you sign this contract saying you'll let AMD copy them. If you don't sign, then we won't buy from either of you."

The three companies agreed and thus created the famous Intel-AMD cross-license for both chipmakers to design and sell chips implementing the same x86 ISA.

NOTE *This is a general lesson about computer economics: after a sale, the seller of a hardware or software platform can wield extreme power over the buyer via lock-in. Platform sellers should thus try to initially give away their platforms for free or at large discounts, to get users locked into them, before ramping up their sales terms once they have the buyer over a barrel. But before the buyer selects a platform, it's the buyer who holds all the power and calls the shots. Thus, buyers should negotiate hard to formalize a contract that mitigates the seller's power over them later. Once you hand over the money, you have no power except what was agreed in that contract.*

The IBM deal propelled both chipmakers into the business computing market, enabling them to scale rapidly. After the deal, Intel updated the 8086 with its 80186 (1982; 6 MHz), followed soon after by the 80286 (1982; 8 MHz), which added protected mode for OS support for the first time. AMD then quickly cloned the 80286 as its Am286 (1982; 8 MHz). These 16-bit devices were appearing in the early 1980s as high-end business machines, at the same time that the 8-bit golden age was arriving in homes.

32-Bit Clone Wars Era

The 32-bit era began with Intel's 386 (1985; 16 MHz), which introduced the 32-bit instruction set x86 IA-32. Throughout this era, we saw continual antagonism and legal action between the two big chipmakers; this was made more entertaining by the entry of additional competitors Cyrix and Via, who also made x86 clones. Table 13-1 summarizes these developments.

Table 13-1: 32-Bit Era x86 Developments

Year	Maker	Architecture	Features
1985	Intel	386	16 MHz
1989	Intel	486	50 MHz, pipelined, FPU
1991	AMD	Am386	Clone of 386
1993	Intel	Pentium	75 MHz, superscalar
1993	AMD	Am486	Clone of 486 (last clone)
1995	Intel	P5	150 MHz, MMX SIMD "Pentium MMX"
1995	Intel	P6 (i686)	200 MHz, SSE SIMD, OOOE, "Pentium Pro"
1996	AMD	K5	133 MHz, Pentium-like
1995	Cyrix	Cx5x86	140 MHz, Pentium-like
1996	Cyrix	6x86	140 MHz, Pentium-like
1997	AMD	K6	300 MHz, 3D-NOW, rival SIMD
2001	VIA	C3	500 MHz, Pentium-like
2001	AMD	Athlon	2 GHz

Intel was usually the technical leader, creating new technologies such as pipelined designs and extension instructions, with the others copying a year

or two later to bring the price down. At every step, clock speeds reliably got faster, following Moore's law for clock speed. This was the "bland 1990s," where customers assumed they would need to buy a new beige desktop computer every 18 months to keep up with doubling clock speeds.

After the 486, Intel got sick of competitors copying the untrademarkable 86 name, so they switched to the trademarkable brand name "Pentium." This was the dominant chip for some time, but then AMD took the lead by becoming the first to reach 1 GHz speed with its Athlon in 2001.

64-Bit Branding Era

The 64-bit era of x86 arrived in 2000 when AMD formally defined the amd64 ISA, which was adopted by most CISC processors following it. This was a coup: the x86 ISA family had previously always been defined by Intel, with others pegging their own products to them.

Intel attempted to define its own failed 64-bit competitor ISA, called IA-64, but this was released after amd64 and never caught on; today, everyone uses amd64. Intel, however, refuses to acknowledge the name amd64, instead referring to the same ISA as x86_64. Confusingly, you'll see both names used to describe executable software downloads for this ISA, such as in the names of Linux distribution packages.

The 64-bit era is characterized by a separation of marketing terms from the underlying technologies, with the same marketing brand often used to label completely different architectures. Unlike the previous 32-bit Pentium, the branding is no longer attached to specific designs. You're probably used to seeing 64-bit products with brands like Pentium, Celeron, and Xeon. You may also see the numbers 3, 5, 7, and 9 in brand names, as in Core i3, Core i5, and so on. For Intel, these numbers don't mean anything other than suggesting an ordering of which products are better; AMD uses the same numbers to suggest which products are similar to Intel's.

Table 13-2 shows examples of Intel and AMD releases and some of their notable features during the 64-bit era.

Pipelines have varied between around 14 and 20 stages during this period, and OOOE has been used throughout. AMD Piledriver was the first to introduce neural network–based branch prediction hardware.

Clock speeds hit 3.5 GHz around the start of the 64-bit era and have been stuck there ever since, due to the end of Moore's law for clock speed. However, Moore's law for transistor size continued to hold, and it became common to define machines by their transistor scale, in nanometers (nm) per transistor, rather than their clock speed, to show the continued progress. Between 2006 and 2016, Intel used a "tick-tock" cycle, in which their new products alternated between new digital logic designs (tock) and the use of new transistor technologies to make the same design smaller and faster (tick). *Boosts* are a feature first added in Nehalem, which *temporarily* increase the clock speed beyond the usual 3.5 GHz heat limit for short periods of time at the bottlenecks of intensive computations.

Table 13-2: 64-Bit Era x86 Developments

Year	Maker	Architecture	Transistor size (nm)	Branding
2003	AMD	Hammer (K8)	130	Opteron
2005	AMD	Hammer (K8)	90	Athlon 64 X2
2006	Intel	Core	65	Celeron/Pentium/Xeon
2007	AMD	10h (K10)	65	Opteron
2008	Intel	Nehalem	45	Pentium, Xeon, Core (1st generation)
2011	Intel	Sandy Bridge	32	2nd-generation Core i3/i5/i9; Xeon
2012	AMD	Piledriver	32	Opteron
2013	Intel	Haswell	22	4th-generation Core i3/5/7; Celeron/Pentium/Xeon
2015	Intel	Skylake	14	6th-generation Core i3/5/7; Celeron/Pentium/Xeon; CoreM
2017	Intel	Coffee Lake	14	8th-generation Core i3/5/7; Celeron/Pentium Gold/Xeon
2017	AMD	Zen	14	Ryzen 3/5/7 1000 series
2018	AMD	Zen+	12	Ryzen 3/5/7 2000 series
2019	AMD	Zen2	7	Ryzen 3/5/7 3000 series
2020	AMD	Zen3	7	Ryzen 5/7/9 5000 series
2021	Intel	Cypress Cove	14	11th-generation Core i5/7/9; Xeon
2021	Intel	Golden Cove	7	12th-generation Core i5/7/9; Xeon
2022	AMD	Zen4	5	Ryzen 5/7/9 7000 series

Now that we've seen how x86 evolved, let's look at its instruction set and learn how to program it. This will be a messier experience than for the other architectures we've studied, but hopefully, by understanding the history, you can at least understand why things ended up this way.

Programming x86

x86 is big and ugly; its code is usually generated by compilers rather than written by hand. Still, it's worth your time to study it if you want to better understand what your compiler and computer are doing, or if you want to write compilers or other system software such as operating systems and bootloaders. Because x86 is such a widely used architecture, understanding it is also useful in security applications, such as cracking and defending code, including cheat and anti-cheat systems for games.

As a CISC architecture, x86 often has many variations of each instruction, taking different types of operand, such as constants, registers, and memory locations. Groups of instructions have been added at different points in the architecture's history, and they don't always use the same conventions: for example, integer addition, integer multiplication, and floating-point operations all present very different interfaces to the programmer. You wouldn't design a new CPU from scratch using such different interfaces; this mess is simply how the architecture has grown over time.

This won't be an exhaustive tour of x86 features. Rather, we'll look at a couple of examples to give a flavor of how CISC extensions are created and how they operate.

Registers

Because of the way x86 has evolved over time and its requirement for backward compatibility, its register set has grown into a particular form. There are two general types of register; let's look at each.

General-Purpose Registers

There are eight general-purpose user registers in x86 architecture. Their names reflect their traditional uses. Table 13-3 shows them.

Table 13-3: x86 General-Purpose Registers

Register	Meaning	Use
AX	Accumulator register	Arithmetic operations
BX	Base register	A pointer to data
CX	Counter register	Shift, rotate, and loop instructions
DX	Data register	Arithmetic and I/O operations
SP	Stack pointer register	A pointer to the top of the stack
BP	Stack base pointer register	A pointer to the base of the stack
SI	Source index register	A pointer to a source for data copies
DI	Destination index register	A pointer to a destination for data copies

In the original 16-bit 8086, the general-purpose registers all had 16 bits. To retain partial backward compatibility with the previous 8-bit 8080, the first four—AX, BX, CX, and DX—can also be split into two 8-bit registers, named with H and L for high and low bytes, which can be accessed independently.

IA-32 extended the eight registers to have 32-bits. They can still be accessed as 16- or 8-bit registers as before, to maintain compatibility. To access them in their full 32-bit mode, we add the prefix E (for *extended*) to their names: EAX, EBX, ECX, and so on.

amd64 extended the eight registers again, to 64 bits. As before, the 32-, 16-, and 8-bit versions are left intact for compatibility. To access them in 64-bit mode, we add the prefix R to their names: RAX, RBX, RCX, and so on. amd64 also added eight more 64-bit general-purpose registers, named R8 through R15.

As x86 is defined as the family based on the 16-bit system, and has to retain backward compatibility, a *word* in x86 speak still means 16 bits of data, rather than the full size of the modern registers. *Doubleword* or *dword* means 32 bits, and *quadword* or *qword* means 64 bits.

Figure 13-1 summarizes the evolution of the general-purpose x86 registers.

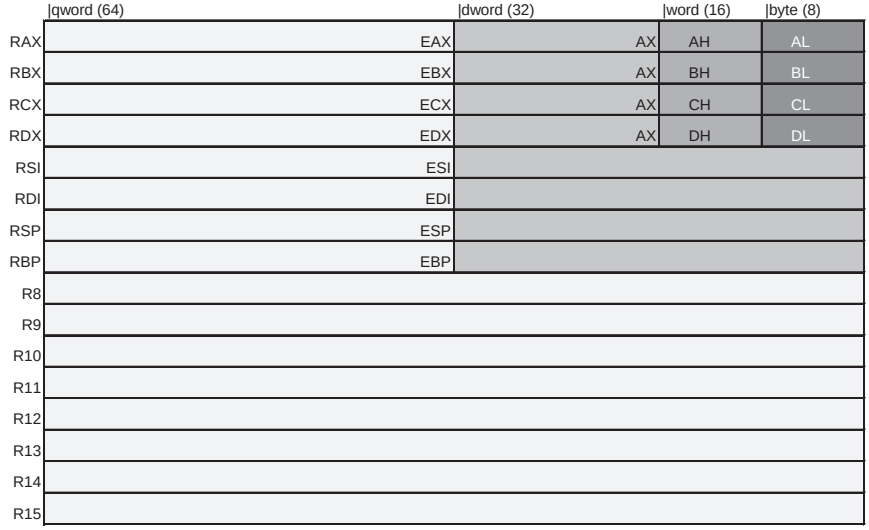


Figure 13-1: The x86 registers. Register names are shown to the left of each register, apart from 8-bit register names, which are shown in the center of the register.

For compatibility with these different word sizes, memory addressing is always done *per byte*, even on a modern amd64. This is in contrast to addressing, say, non-overlapping 64-bit *words* of memory. Words are stored in memory as little-endian bytes.

Internal Registers

The program counter is called the *instruction pointer* in x86 speak, identified as IP, EIP, or RIP when used in its 16-, 32-, or 64-bit form, respectively.

The status register is called FLAGS, EFLAGS, or RFLAGS, again when used in 16-, 32-, or 64-bit form. Its structure is shown in Figure 13-2.

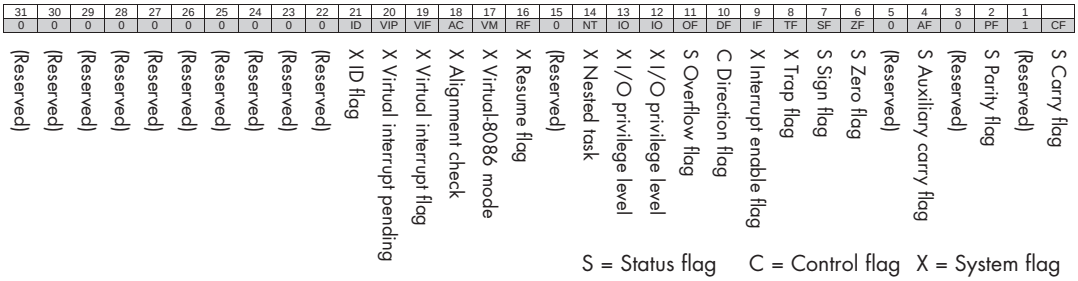


Figure 13-2: The x86 status register (compare with Figure 11-6)

This is very like the 6502's status register, with similar mnemonics. As with the 6502, these flags are set with comparison instructions, then consulted with separate branch instructions. There are also instructions to clear flags. Two important flags, as in other architectures, are the zero flag (ZF) and sign flag (SF).

Netwide Assembler Syntax

Because of its long history, x86 has acquired several different assembly languages with different syntaxes, which all assemble into the same machine code. Here we'll use the *Netwide Assembler (NASM)* style, which is the least worst of them.

x86 instructions usually have two operands. In NASM syntax, the first is usually the destination and sometimes also an input that gets updated to store the result, like an accumulator; the second operand is an input.

Like most assemblers, NASM enables us to label lines of a program with text labels by inserting the label as text, followed by a colon, like so:

```
mylabel:
```

If a label is inserted on line 5, we can jump to or load from line 5 by using its label name rather than the number 5.

Data Movement

To copy constants or register contents between registers and RAM, you can use the same `mov` (move) instruction. This generalizes all of loading, storing, and moving. Several different addressing modes are provided.

Immediate addressing places constants into registers. For example:

```
mov rbx, 123          ; place decimal 123 into register RBX
mov ebx, 4c6h         ; place hex 4c6 into register EBX
mov bh, 01101100b     ; place binary 01101100 into register BH
```

Register addressing copies data from one register to another inside the CPU, such as:

```
mov rax, rbx          ; copy to RAX from RBX
```

Direct addressing loads from and stores to memory through a specified address. Labels can be used in place of numerical addresses, in which context they're known as *variables*. For example:

```
mov rbx, [1000h]      ; load to RBX from hex address 1000
mov [1000h], rbx      ; store to hex address 1000 from RBX
mov rbx, [1000h+20h]  ; load from an address with offset
mov [1000h+20h], rbx  ; store to an address with offset
mov rbx, myvar        ; load a labeled address (address, not its content)
mov rbx, [myvar]      ; load content of a labeled address
mov [myvar], rbx      ; store to a labeled address from RBX
```

Register indirect addressing is notated using square brackets, such as:

```
mov rax, [rdi]         ; copy to RAX, from content of the address in RDI
mov [rdi], rax         ; copy to address in RDI, from RAX
```

In these two instructions, RDI is assumed to contain an address that in turn is used to load or store the value from RAX.

Data Creation

Data locations in RAM can be given names, and can be initialized or uninitialized. To initialize a location with a value and create a name for it, we use commands beginning with `d`, for *define*. For example:

```
mybyte:  db 15          ; define byte
myword:  dw 452         ; define word (2 bytes)
mydword: dd 478569      ; define doubleword (4 bytes)
myqword: dq 100000000   ; define quadword (8 bytes)
```

To name an uninitialized location, we use commands beginning with `r`, for *reserve*:

```
mybyte:  resb 1         ; reserve uninitialized 1 byte
myword:  resw 1         ; reserve uninitialized 1 word
mydword: resw 1         ; reserve uninitialized 1 doubleword
myqword: resw 1         ; reserve uninitialized 1 quadword
```

Note that these aren't x86 instructions, but rather just labeled regions of data, with the directives telling NASM to treat them as such.

To create arrays, we simply allocate a set of consecutive addresses. For example:

```
myarray:  dq 1, 2, 3, 4 ; define 4 quadwords, myarray addresses first element
myzeros:  times 4 dw 0  ; define 4 doublewords all to 0
mywords:  resw 100      ; reserve uninitialized 100 words
mystring: db "hello", "world", 10, 0    ; define a single 12-char ASCII string
```

NASM also provides macro directives, which enable you to define numeric (`equ`) and string (`%define`) constants:

```
SCREEN_WIDTH equ 1920
%define isTrue 1
```

NASM substitutes for these constants' values before doing the assembly. These macro directives aren't part of the x86 instructions set, but NASM provides them for convenience.

Arithmetic and Logic

As x86 instructions are usually designed to take two arguments, most arithmetic is done accumulator-style. There isn't a single accumulator register, but any register can act like one. For example, here we place the value 1 into RBX and add 2 into it, so it ends up storing the result, 3:

```
mov rbx, 1
add rbx, 2
```

As a CISC architecture, variations of arithmetic instructions usually exist that combine loading data from memory with the arithmetic. For example,

here's how to add two numbers from addresses 1000h and 2000h and put the result in RBX:

```
mov rbx, [1000h]
add rbx, [2000h]
```

Note that x86 *doesn't* include the most extreme CISC style of addition, such as `[3000h] := [1000h]+[2000h]`, which combines two loads, one addition, and one store in a single instruction.

Subtraction works similarly to addition:

```
sub ax, 5
```

Incrementing and decrementing 8-, 16-, or 32-bit operands can be done using the `inc` and `dec` instructions:

```
dec ax                ; decrement content of register
inc [mybyte]          ; increment content of variable mybyte
```

To multiply or divide integer operands, x86 provides `mul` and `div` instructions. Unlike addition and subtraction, these always use the A register as the accumulator (hence its name) and act on it with the operand given to the instruction. For example:

```
; 64-bit multiplication
mov rax, 2
mov rbx, 3
mul rbx      ; result 6 is in accumulator RAX
; 16-bit multiplication
mov ax, 20   ; first operand
mov bx, 4    ; second operand
mul bx       ; result is stored in AX
; 8-bit division
mov al, 10   ; dividend
mov bl, 2    ; divisor
div bl       ; result stored in AL
; 16-bit signed division
mov ax, -48  ; dividend is negative, need signed version
cwd          ; extend AX into DX
mov  bx, 5
idiv bx      ; result in AX, remainder in DX
```

In the last of the above examples, the prefix `i` is added to the `div` instruction to indicate that signed integers are used. The `cwd` instruction converts a word to a double by allowing the DX register to be used as an extension of AX in order to accommodate the sign information.

Bitwise logic instructions include `and`, `or`, `not`, and `xor`. For example:

```
and ax, 01h
or  ax, bx
not ax
```

As with addition, the first operand acts as an accumulator so gets overwritten with the result.

Flow Control

NASM provides two types of labels, symbolic and numeric, that can both be used for jumps and branches. Symbolic labels consist of an identifier followed by a colon (`:`). They must be defined only once, as they have global scope. If the label identifier begins with a period (`.`), it's considered local and can be used only in the current file. Here's an infinite loop using a symbolic label and a jump:

```
mylabel:
    jmp mylabel
```

Numeric labels consist of a single digit in the range 0 to 9 followed by a colon. Numeric labels are considered local. They also have limited scope so can be redefined repeatedly. When a numeric label is used as a reference (as an instruction operand, for example), the suffixes `b` (for backward) or `f` (for forward) should be added to the numeric label. For numeric label `1`, the reference `1b` refers to the nearest label `1` defined before the reference, and the reference `1f` refers to the nearest label `1` defined after the reference. For example:

```
main:
    1:                ; define new numeric label
    ; do something
    jmp 1f            ; jump to first numeric label "1" defined
    1:                ; redefine existing label
    ; do something
    jmp 1b            ; jump to last numeric label "1" defined
```

Conditional jumps are performed using pairs of instructions. First, we use the `cmp` instruction to compare two values. It takes two operands to compare and raises appropriate flags in the status register. Next, a conditional jump instruction consults the status register to determine whether or not to make the jump. Some of the available conditional jump types are listed in Table 13-4.

Table 13-4: x86 Conditional Jump Instructions

Instruction	Condition
`je`	Jump if `cmp` is equal
`jne`	Jump if`cmp` is not equal
`jg`	Signed > (greater)
`jge`	Signed >=
`jl`	Signed < (less than)
`jle`	Signed <=
`ja`	Unsigned > (above)
`jae`	Unsigned >=
`jb`	Unsigned < (below)
`jbe`	Unsigned <=
`jc`	Jump if carry (used for unsigned overflow or multi-precision add)
`jo`	Jump if there was signed overflow

To illustrate, this program uses the `cmp` and `je` instructions to make a jump if the compared values are equal:

```
cmp 15, 10
je equal              ; jump to "equal" label if equal
; continue if jump condition is false
cmp 10,10
je equal
equal:
   ; they are equal
```

Subroutines are called and returned from as follows:

```
main:
   call somefunction

somefunction:
    ; some content
    ret
```

The `call` instruction jumps to the subroutine with the given label, and `ret` returns from the subroutine to the calling location.

The Stack

Subroutine calls and returns are implemented internally using a stack. If you're just writing simple calls and returns, as in the example we just looked at, you don't need to see or think about the stack yourself. However, x86 also allows you to access the stack directly to pass arguments or for other purposes. Specifically, registers SS and ESP (or SP) are provided and used

for implementing the stack. The stack is limited to storing only words and doublewords. Here's how it works:

```
; save register values
push ax
push bx
; perform whatever you want with these registers
; restore the value
pop bx
pop ax
```

Here, the contents of registers AX and BX are pushed to the stack, meaning these registers can then be overwritten and used for other purposes, before being restored by the `pop` instructions.

X86 CALLING CONVENTIONS

The x86 architecture has been used with many different calling conventions during its history. Due to the small number of architectural registers, and a historical focus on simplicity and small code size, many x86 calling conventions pass arguments on the stack. The return value (or a pointer to it) is returned in a register. Some conventions use registers for the first few parameters, which may improve performance, especially for short and simple *leaf routines* that are very frequently invoked (these are routines that don't call other routines).

For amd64, there are two current conventions in widespread use, one suggested by System V UNIX designers and the other by Microsoft. They agree that the caller rather than callee should clean up the stack. They both require the first few arguments to be passed in registers, with the later arguments on the stack, right to left, though they disagree on how many and which registers to use. They disagree on which registers are *temporary*—that is, which can be overwritten by the callee during a function call. This is in contrast to those that are *safe*, guaranteed to not be changed by function calls.

BIOS I/O

We can call BIOS routines from ROM to communicate with the screen and keyboard, as on a retro computer. For example:

```
; BIOS Character display
mov ah, 0eh      ; set mode
mov al, 'H'      ; char 'H' to print
int 10h          ; ask BIOS to display letter on screen
; BIOS Character input
mov ah, 00h
int 16h          ; ask BIOS to read a keypress char to AL
; BIOS Graphics   (only works in 16-bit mode)
mov al, 13h      ; desired graphics mode
mov ah, 0        ; set graphics mode
```

```
int 10h          ; ask BIOS to set graphics mode
mov al, 1100b    ; desired pixel RGB color
mov cx, 10       ; desired pixel x coordinate
mov dx, 20       ; desired pixel y coordinate
mov ah, 0ch      ; ask BIOS to light the pixel
int 10h
```

This sets a screen mode, prints an ASCII character to a location on the screen, reads an ASCII character from the keyboard, and sets a pixel color. These are all the basic ingredients you need to make 8 bit–style video games. The `int` instructions here generate interrupt requests, which pass control to the BIOS, and their operands tell the BIOS which of its subroutines is to be run. These subroutines each assume that their arguments have been placed into particular registers such as AH and AL before the interrupt is made.

Floating Point

The x86 floating-point architecture derives from the 8086's old coprocessor, the 8087. This was a separate, optional chip for accelerating numerical computation. Since the 486, the FPU moved into the main x86 architecture, where it has become known as the *x87 extension*.

The x87 extension adds dedicated floating-point registers called ST0 to ST7, which are used as a stack (hence the prefix *ST*); the stack has a maximum of eight elements, with ST0 being the top. New floating-point instructions start with the letter `F` and move data to and from this stack; they instruct the FPU to perform arithmetic using the top items of the stack.

You can push floats to the x87 stack, call arithmetic on them, and pop the result back, such as:

```
a: dw 1.456            ; a word (16-bit) float
b: dd 1.456            ; a doubleword (32-bit) float
c: resq 1              ; reserve for output float
;FP add
fld qword [a]          ; load a (pushed on flt pt stack, st0)
fadd qword [b]         ; floating add b (to st0)
fstp qword [c]         ; store result into c (pop flt pt stack)
;FP multiply
fld qword [a]          ; load a (pushed on flt pt stack, st0)
fmul qword [b]         ; floating multiply by b (to st0)
fstp qword [c]         ; store result into c (pop flt pt stack)
```

Here, when you give an ASCII representation of a float to NASM for any of the word lengths used, NASM knows to convert it to IEEE binary representation for you.

Segmentation

x86 programs can be written as collections of *segments*, which are separate chunks of a program that can be stored in different locations in memory.

For example, if you wish to keep your instructions apart from your data (as in a Harvard architecture), you can do this by using a separate code segment and data segment. A stack segment can also be used to keep the hardware stack data separate from both. Segments all live in the same global address space, but by storing the start address of each segment in a dedicated register, addresses within them can afterward be referred to by just their offset from the segment start. This system was intended as a way for 16-bit CPUs to work with more than 64 k_2B of RAM. It still exists but isn't used much in modern 64-bit x86, because the 64-bit address space is so large anyway. Six *segment registers*, called CS, SS, DS, ES, FS, and GS, are specified to hold the segment start addresses.

If you're using the segment system, the NASM directive `section` specifies code and data segments. In some settings, some assemblers will still look for sections and assume that `section .text` is read-only and that `section .data` is read-write, even though the concepts are no longer used at the amd64 hardware level. A *segmentation fault* will occur if you try to access a segment that the assembler doesn't want you to access.

Backward-Compatible Modes

Part of the x86 standard is that all CPUs have to be backward-compatible with the original 16-bit 8086. This means that when they first power on, they have to start in 16-bit mode and behave exactly like an 8086.

From there, 32-bit x86s have instructions that switch them into 32-bit mode, and 64-bit x86s have further instructions to switch from 32-bit to 64-bit mode. To boot an amd64, you therefore progressively switch up into 32- and then 64-bit mode, replaying the history of its architecture in a fraction of a second.

Now that we have an understanding of the x86 architecture, let's zoom out to consider the PC computer design that uses it as the CPU component.

PC Computer Design

The desktop PC is a different concept from the other computers we've studied: rather than specifying one particular computer design, it's a loose collection of formal and informal standards. The first PCs were designed and defined as such by IBM, beginning in 1981 with the IBM 5150, seen in Figure 11-1; they were then copied by other manufacturers using similar compatible components.

In the 1990s, any computer with an x86 CPU capable of running a Microsoft DOS or Windows operating system was generally considered to be a PC. Microsoft chose what computer design features to support in this software, so it effectively set the standard definition. Other operating systems could also run on many of these machines while making different support choices. Often there are multiple competing standards for computer design features, and it becomes a political as well as technical question which ones get taken up by the PC community.

Programming and using PCs thus feels different than more standardized platforms. For example, games created for a particular machine, such as a Commodore 64, can assume a precise hardware feature set and will run exactly the same on any Commodore 64. This enables the game designer to work as an artist, making the game look and feel exactly as they intend. But a game made for PCs will run differently on different PCs with different features, requiring game designers to create what is really a whole set of similar games, some of which they'll never see themselves and can only guess at how to implement. Similarly, game players may have to get more involved in configuring their hardware and software to customize which version of the game they want to play.

Here we'll look at some specific examples of buses, I/O modules, and devices used in today's desktop PCs. These can often form the bottlenecks in modern PCS—there's little use in having a highly optimized CPU if it has to spend its time waiting on other parts of the system. When you buy a computer, don't just look at CPU speed—think about these supporting structures, too.

The Bus Hierarchy

Like CPUs, buses are continually being improved and replaced, so the PC architecture has used various standard bus hierarchies over time. Buses can be found in a desktop PC at several layers; each layer has different uses and different bandwidths, and is optimized for different purposes. Table 13-5 shows some recent standards with their speeds and typical uses.

Table 13-5: PC Bus Speeds and Uses

Standard	Bandwidth (GBps)	Uses
Gigabit Ethernet	1	Network
USB3	5	Peripherals
SATA3	6	Secondary storage
NVMe	32	Secondary storage
PCI express 5.0 x16	63	Graphics cards

You can see that communication with the outside world via Ethernet is at the slower end, local peripherals and secondary storage are in the middle, and graphics cards have had a lot of work done to make them communicate quickly.

The classic PC hierarchy used two structures called Northbridge and Southbridge—known together as the *chipset*—as the main skeleton of the bus hierarchy. This is shown in Figure 13-3.

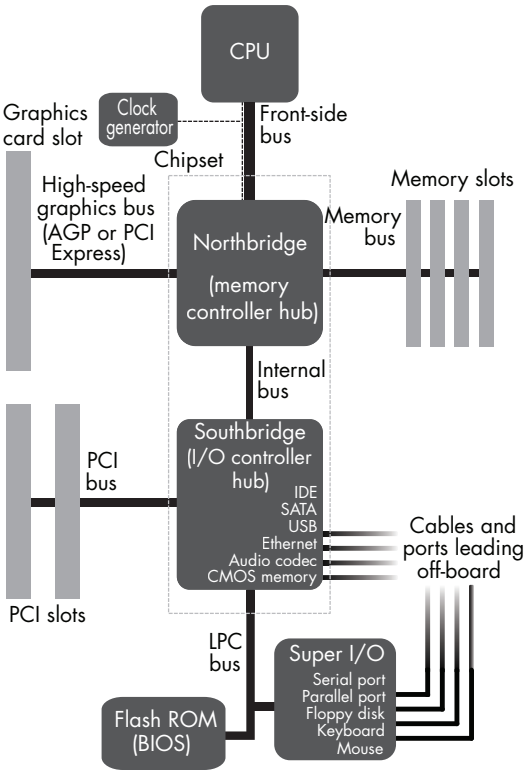


Figure 13-3: The Northbridge-Southbridge bus architecture

Northbridge connects directly to the CPU's FSB (front-side bus) and links it to RAM and to fast I/O modules using the same address space via PCIe bus. It also connects to Southbridge. Northbridge is fast and powerful. It was traditionally constructed on a separate chip from the CPU that also hosted some memory cache levels. More recently, Northbridge has moved onto CPU silicon in many systems.

Southbridge bridges a second time, from Northbridge to slower I/O bus hierarchies. It's still usually located in its own dedicated silicon chip (which is sometimes also called "the chipset" even when Northbridge is located on the CPU chip). Southbridge contains many different standard I/O modules, all printed on the same silicon. Here you'll see structures such as USB controllers, hard disk controllers, and the older PCI (not PCIe) bus.

Figure 2 in the introduction shows the physical layout of this design on a 2010s PC mainboard. In the figure, both Northbridge and Southbridge are covered by large heatsinks, showing that they're major consumers of power and producers of heat, just like the CPU. Compared to retro computers, there are few other chips remaining on the mainboard, because most of their functionality has migrated to either Southbridge, Northbridge, or the CPU. The rest of the mainboard is taken up mostly by physical connectors and analog components used in power management.

With Northbridge now migrated onto the same silicon as the CPU in many cases, it's become harder to identify it on more modern mainboards.

Standardized I/O

A current desktop PC trend is toward standardized I/O. In the bad old days, every device would have its own I/O module, a physical component sitting on the bus. That meant that each device had its own IRQ (interrupt request) line into the processor. You would need a specific I/O-level driver to look after that module, which could be painful to configure.

Bus hierarchies such as USB have now largely solved this problem for PCs. These use a single I/O module, such as a USB controller, which has to be configured only once and uses only a single IRQ. All the devices then connect to this controller using a lower-level bus with its own protocol, which can include communications that inform the controller what the device is. They can easily share the single IRQ allocated to the controller.

Fast Serial Buses

In the golden age, a bus meant a whole load of parallel wires, often in the form of a ribbon cable, as in the left of Figure 13-4.

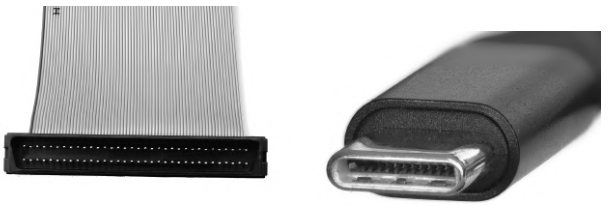


Figure 13-4: A 1980s parallel bus ribbon cable with lots of wires (left) versus a fast serial 2020s connector with fewer wires

It's rare to see ribbon cables nowadays, as most buses are serial, having just one wire for communication plus a few control and power wires, as on the right of Figure 13-4. For example, SATA, SSA-SCSI, USB, and CAN are all serial buses.

This change was prompted by technical problems with parallel buses that arrived once speeds exceeded around 1Gbps. Small differences in delays on out-of-box parallel wires can put signals on different wires out of sync, and resynchronizing their data is very hard. Serial buses, on the other hand, can be made faster and faster as there's no need to sync multiple wires.

Migration Up the Hierarchy

As I/O modules get faster they want to move up the bus hierarchy to be closer to the CPU. Devices that used to hang off standardized buses, such as USB, want to connect directly to Southbridge; devices that used to hang off Southbridge want to get promoted to Northbridge; and devices that used to hang off Northbridge want to get promoted up into system-on-chip (SoC) silicon. At the same time, Northbridge, Southbridge, and standardized buses all want to increase their own speeds, meaning a device wanting

to move from Southbridge to Northbridge, for example, might get overtaken by a new, faster Southbridge that makes its migration unnecessary. Since Moore's law stopped the central CPU clock from getting faster, there's been a big push to move innovation to all of these levels, which perhaps is making it a little more glamorous for the non-CPU architects who work on them.

Migration up the bus hierarchy and onto silicon makes the economics and legal structures of computer design harder to understand. In 8-bit times, different companies could make separate physical chips, such as CPU and I/O modules. Computer manufacturers would buy these chips, then design and build PCBs to integrate them. Nowadays, as more of these structures need to be fabricated together on the same piece of silicon, the CPU and I/O module companies need to share their designs with the computer manufacturer, using software files similar to LogiSim designs. The manufacturer then adds designs to these files to link them together, then sends them to a fabrication company. The units of digital logic design provided by each company are known as *IP (intellectual property) cores* and need to be closely guarded by lawyers and patent agents rather than just bought and sold as physical chips in plastic packages.

Common Buses

Most of the space on mainboards is now taken up by connectors rather than chips, as you saw in Figure 2 of the introduction. The connectors seen in that figure are typical of other parts of the bus hierarchy. We'll examine some of the main ones next.

Peripheral Component Interconnect Express Bus

PCIe (not to be confused with the older PCI) stands for Peripheral Component Interconnect Express and is a general-purpose bus for connecting graphics and other cards. PCIe comes in several flavors, as shown in Figure 13-5; the connectors have physically different widths because they have different numbers of lanes.

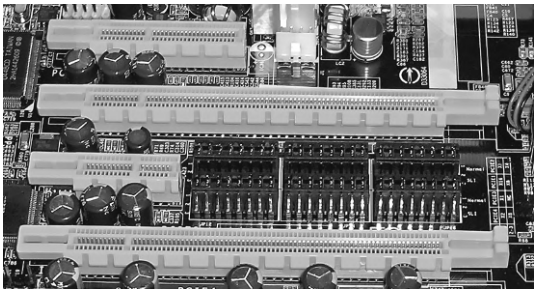


Figure 13-5: Some PCIe bus connectors

You can get various powers of 2 between 1 and 32 lanes, depending on how much data you want to transfer. PCIe also comes in different generations, with speeds going from 250MBps to 2GBps per lane.

Like many modern "buses," PCIe began as an actual bus—in which many nodes share the same set of wires, each with its own address—but has evolved into a mesh network, with nodes now performing some routing to avoid congestion on the bus.

SCSI and SATA Buses

SCSI and SATA are competing buses for mass storage devices (for example, hard disks). The Small Computer System Interface (SCSI, pronounced "scuzzy") is a very ancient, classic, well-tested, reliable, and expensive standard, dating from the 1980s. It pioneered moving compute work for I/O control from CPU into digital logic in the I/O module, freeing up the CPU to work on other tasks more quickly. It's used today in servers. SCSI has been through many versions; the latest update is Serial Storage Architecture (SSA), a serial bus version.

Serial Advanced Technology Attachment (SATA) is cheaper and simpler than SCSI. For these reasons, it's used in most consumer systems rather than SCSI.

Universal Serial Bus

The *Universal Serial "Bus" (USB)* is the one you're probably most familiar with. However, USB isn't a bus at all—it's not even a mesh network. It's actually a point-to-point connector, intended to upgrade the older serial port.

Before USB was invented, whenever you got a new piece of hardware you would spend a day trying to get the device driver working and configuring the IRQ lines. USB now makes all of this instant so you can "plug and play" many devices. USB is designed so that devices can be connected and disconnected while the computer is turned on, and part of its standard defines a generic method for devices to state their type and model over the basic USB protocol itself rather than requiring a device driver. This enables computer software to automatically see what's been plugged in, and in many cases to download and run the appropriate drivers for it without intervention.

USB also defines standards for requesting and sending power down the wires. A USB cable has four wires, two for sending a serial signal and two for power. There are 5 V and a ground in there, so, for example, you can use the same USB cable to charge your mobile phone and exchange data with it.

All of this is done through a centralized USB controller, which is a single I/O module, so you don't have to worry about IRQs anymore. The USB controller itself has an IRQ, but then everything else is hanging off a USB network. There have been different versions of USB, including USB 1 running at 12Mbps and USB 3 running at 5Gbps.

Unlike some point-to-point networks, USB connections have a manager end and a worker end, with the manager in charge of the communications protocol. If you plug a USB memory stick into your computer, your computer is the manager. As the worker, your USB stick can't take over and start sending its own requests to copy data from your computer. This is why USB wires have different endings: one end plugs into the manager that controls

it and the other end goes into the worker, and you can't connect them the other way around.

On-the-go (OTG) is part of the USB protocol that allows a worker device to act as a manager via a physical adapter. Sometimes you do want to connect them the wrong way around. For example, when you connect your smartphone to your computer, you usually want it to be the worker, like a USB stick, with your computer as the manager. But other times you want the phone to be the manager, such as when connecting a memory stick or sound card to it.

Ethernet

Ethernet, in its oldest and simplest form, is a true bus, with multiple PCs in a local area network all writing and reading on public wires. Each message is packaged as a "frame," containing the address (Media Access Control, or MAC, address) of the recipient. Senders must take care to avoid collisions—that is, people talking at the same time—by watching the bus and waiting for a suitable time to transmit. Everyone can see everything on the bus, so it's easy to "sniff" the bus and spy on other users.

Modern networks build non-bus features on top of the basic Ethernet bus structure. For example, rather than connecting all computers in a building to a single shared Ethernet bus, it's now common for each to connect only to a central *switch* using a dedicated Ethernet cable. The switch receives all messages that are sent, but rather than forwarding them, bus-style, to all machines on the network, it forwards them only to the intended destination.

Standard Devices

Your desktop PC wouldn't be complete without some other standard devices. To complete our study of PCs, let's take a quick look at how these have evolved.

Flat-Screen Displays

Modern flat-screen displays are used in mobile phone screens and large-screen TVs and monitors. They're made from transistors and capacitors, laid down like chips by photolithography masks and gas processes. Many rare elements are used to produce the specific red, green, and blue light-emitting pixels, including yttrium, lanthanum, terbium, praseodymium, europium, dysprosium, and gadolinium. Some of these are so rare that they can be mined only in one or two places. Many specific combinations of electronics and elements have been used as display "technologies," including TFT. The latest at the time of writing is organic LED (OLED).

Graphics Cards

In the 1980s, graphics was simple. An area of memory was allocated to represent the array of pixels on the screen. User programs would write to it like any other part of memory. Then a graphics chip would read from it and turn the data into CRT scanning commands to send to the monitor. Now

things are more complicated, as programmers expect graphics hardware to provide commands for complex rendering of 2D and 3D shapes without taking up CPU time.

To respond to this demand, the modern graphics processing unit (GPU) evolved from 1980s visual display units (VDUs). Rather than taking commands to light up pixels, GPUs typically take commands to render 3D triangles with sprite-like textures, and to shade them using complex lighting models.

If you've been playing video games over the last couple of decades, you'll have seen the visual abilities of GPUs evolve with Moore's law, doubling in quality and getting closer to photorealistic, real-time rendering.

The GPU traditionally sits on one of the buses of the mainboard, such as PCI, AGP, or PCIe. GPUs have been the one part of computer architecture that's been getting physically bigger rather than shrinking over the years, starting off as a small chip and now most likely a full card (Figure 13-6).



Figure 13-6: A 2022 Nvidia RTX 3080 GPU

There has, however, also been a recent trend to shrink GPUs back to put on a single chip on the mainboard, or onto the same silicon as the CPU. This is particularly the case in machines where the GPU isn't the main focus, such as generic business PCs where the graphics requirements don't extend much beyond displaying the desktop.

Graphics cards sit on the system bus as I/O modules. Importantly, they can use direct memory access (DMA). For example, an image can be placed in regular RAM, then a single command can be given to the GPU to load it from main RAM into the GPU. This DMA action doesn't go through the CPU, so from the CPU's point of view it's almost instant. (It will, however, slow down if the bus is needed for other things, such as additional DMAs from a webcam into the main RAM.)

Early GPUs were designed to accelerate rendering of the popular OpenGL 3D graphics API by implementing its commands directly in hardware, beginning with a memory-mapped area and a chip that read that area and figured out how to display that memory block on the screen. In the 2000s, in addition to or instead of memory-mapped graphics, optional plug-in graphics cards sat on the system bus as I/O modules and drew graphics in response to compiled and assembled commands of graphics languages such as OpenGL or DirectX, sent to them via the system bus. Graphics cards were labeled and sold as implementing one or more of these language interfaces.

A 3D graphics language usually assumes that 3D objects are composed of many small triangles. Triangles are chosen because their three points always lie in a plane, making the math easier. Their implementations, in

hardware and/or software, usually split into two main parts, known as *shaders*. First, vertex calculations convert the 3D coordinates of each vertex into 2D pixel coordinates. Second, pixel calculations compute the color (shade) of each display pixel.

The latter can be done in many different ways according to different mathematical models of how surfaces and lights interact. Most shaders allow triangles to be translucent (partly transparent), modeled via an alpha channel in their RGBA color, as discussed on page 68. Some shaders allow normal (orthogonal) vectors to be described for each triangle as a hint that they're part of smooth, continuous surfaces.

Figure 13-7 shows the results of three traditional shaders built into early OpenGL implementations, rendering the same triangle mesh approximation to a sphere.

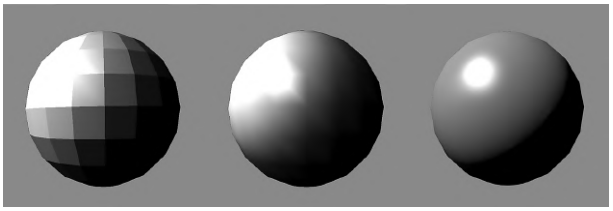


Figure 13-7: Traditional OpenGL shaders: flat (left), Gouraud (center), and Phong (right)

Graphics users demanded more flexibility in shaders. New shading models are often proposed in graphics research, and users wanted them to be quickly available in their own systems. The graphics languages rapidly gained many extension commands in their later versions, to enable particular additional shaders, and graphics card architects struggled to keep up with designing new hardware to implement them and make them compatible with one another. These architects instead began to open up new and simpler shader languages (such as GLSL) to enable these and other arbitrary shaders to be implemented in user programs, and executed on the graphics card—now known as a GPU—via their own ISAs. This allowed programmers—especially game designers and movie studios—to create their own custom shaders to give their creations a more individual feel, as in the examples in Figure 13-8.



Figure 13-8: Custom shaders: water effects from 0 A.D. *(left), "toon" shading (center), and retro CRT emulation (right)*

Today's graphics systems have continued this architectural trend, with GPUs now functioning as highly general parallel processors of their own instruction sets, and the graphics-specific shaders moved into software. Former hardware interfaces including OpenGL and DirectX are now implemented in software, written in the GPU's own assembly and machine code. Such code can now also be generated directly by other graphics tools, such as Wayland compositors and the Vulkan SPIR-V language. The resulting GPU machine code is sent over the bus to the graphics card, where it runs on the GPU. We'll study this code in more detail in Chapter 15.

Sound Cards

Unlike retro sound chips, such as the SID, modern sound cards don't generate signals at all. Instead, they manage the flow of quantized, digital sound wave signals. As a result, computers have lost their characteristic sound effects and musical culture: modern game music can consist of ordinary recordings of orchestras or rock bands rather than any particular "computer music." Like graphics cards, sound cards are always now under OS control, so user programmers are unlikely to see much of their architecture.

A modern sound card is really just a group of digital-to-analog converters (DACs), and indeed it's possible to make your own from any DAC, such as the one found on a Labjack, a software-defined radio, or an Arduino Due. Typically, professional sound cards are optimized for low latency, sound quality, and many channels, while consumer cards are optimized for lower cost. Human hearing has a maximum frequency of around 20 kHz, which requires a 40 kHz sampling rate to be represented accurately. It's common to use 48 kHz to allow some wiggle room and because it's almost a power of 2. Professional systems may use higher rates to reduce the buildup of audible errors from repeated processing.

Sound card hardware typically consists of a ring buffer for each channel, as well as DAC hardware, which reads or writes to and from it. A ring buffer maintains a pointer to the next location to write, and wraps the storage around the ring so space doesn't run out. The buffer size provides a trade-off between latency and dropouts. A small buffer means low latency but risks dropouts. We can also choose the bit depth of the audio.

Sound cards, like graphics cards, connect to the system bus. They're less bandwidth-hungry than video, so they're usually found on a bus hanging off of Southbridge, such as PCI for internal cards or USB or Firewire for external cards.

Sound card I/O protocols vary by manufacturer, and like GPUs, their details may be proprietary and known only to the driver writers inside the company, who then make a software API available. As with GPUs, the hardware or software interfaces are then reverse engineered by open source driver writers, who wrap them in generic software APIs such as *ALSA*.

Keyboards and Mice

Modern keyboards are nothing like the memory-mapped keyboards of the 1980s. They now contain small, embedded computers (see Figure 13-9).

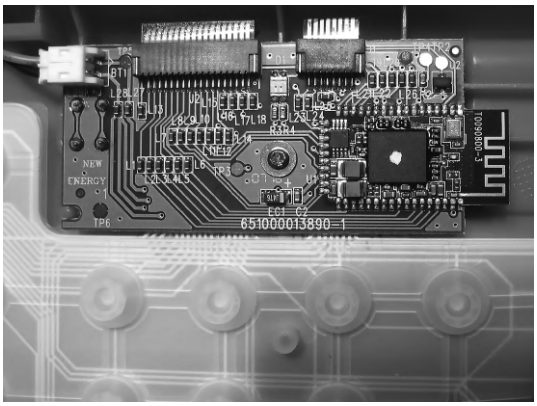


Figure 13-9: The key pressure sensors and embedded system inside a modern keyboard

The keyboard's embedded computer is actually doing a lot of work, similar to a typical Arduino application. It takes the matrix of key presses, converts them to a keycode data representation scheme, and transmits them over a virtual serial port wrapped in USB protocol.

Something similar has happened with mice. A modern optical mouse performs some extremely complicated real-time machine vision processing known as *optic flow* on a dedicated internal embedded system. If you try to implement optic flow in software, you'll find it's hard to do fast. It's still a research area, with recent implementations in software libraries such as OpenCV. In a mouse, however, it's implemented directly as low-level digital electronics, as in Figure 13-10.



Figure 13-10: The inside of an optical mouse

This digital logic is just about simple enough for you to still be able to see the connections. You can see from the overall, fairly homogeneous structure that it's processing a region of 2D space—the image underneath the mouse. It tracks how light and dark areas of this image are moving around and from that infers the movement of the mouse.

There's also usually a USB controller attached to the device. This is actually a complex embedded system—possibly a computer in its own right—and the fact that it's now available for a few dollars in every mouse is very impressive.

THE PC BOOT PROCESS

The term *booting* comes from the paradoxical expression "pulling yourself up by your bootstraps." It means starting with nothing and getting into a complex computer system by having small programs execute that load slightly larger and more powerful programs, in a sequence. On both retro systems and modern PCs, this begins by the CPU fetching an instruction from a hardwired ROM address.

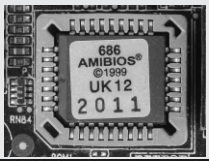
Unlike retro computers, modern PCs aren't made from standard components; instead, they are assembled from many different optional components, such as RAM modules of various types, caches, and I/O extension cards. It's not initially obvious where all these things are, how they should be initialized, or how they should be mounted in the address space. To address this, the modern PC boot process is split into two parts.

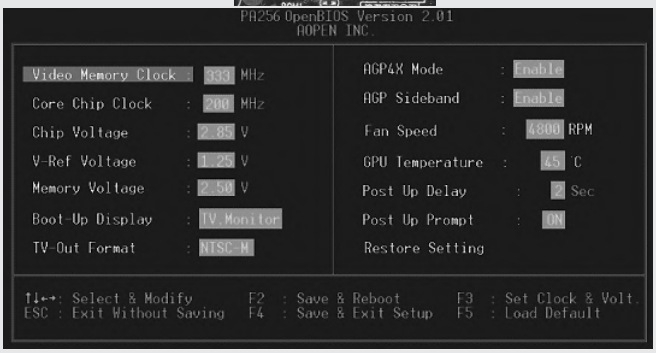
First, a *bootloader* such as *coreboot* is burned into ROM firmware, at the address of the CPU's initial program counter. For x86, this is ffff,fff0_{16}. This is a 16-bit address, because x86 processors always power on in "legacy mode" (Intel calls it "real mode"), which makes them behave like 1980s 16-bit chips for backward compatibility. In this mode, only 1 M_2B of combined ROM and RAM memory is addressable, and the initial program counter address is near the top of it. The bootloader runs from here and is responsible for inspecting, initializing, and assigning addresses to the available hardware. The bootloader doesn't display anything onscreen because there aren't yet any routines available for doing I/O. Because it's invisible, it can be hard to understand all the hard work the bootloader is doing.

Second, after this initialization, the bootloader performs a jump to code in the BIOS. The BIOS, as in a retro computer, contains subroutines for basic I/O such as ASCII character display, keyboard reading, and hard disk access. At this stage, your PC can look and feel much like a retro computer.

(continued)

Usually, the BIOS code jumped to from the bootloader will print a few strings on the screen, such as the name and logo of the BIOS. A PC BIOS ROM and an example of BIOS display I/O capabilities are shown here.





A BIOS will usually first offer the user the chance to "go into the BIOS" by pressing a key, which will call graphical routines for setting configuration options. One of these options is usually to give the name of a storage device whose first data contains the next program to be loaded and jumped to, usually at address $7c00_{16}$. What this program does is up to you—a common first move is to switch the x86 up into 32-, then 64-bit modes.

There was a time when different x86 BIOS manufacturers all made different and incompatible libraries of routines, but they've now converged on two standards. One, PCBIOS, was defined by IBM (who just call it "BIOS") in early x86 PCs. It was cloned by other manufacturers and is still used by many x86 machines today. SeaBIOS is an open source implementation. The other standard, UEFI, is more recent. It assumes more advanced graphics and I/O are available, so its library of routines includes higher resolution and more colorful graphics, and access to additional devices such as USB. TianoCore is an open source implementation.

Summary

No one would design a modern desktop PC to have its current form if they were able to start from scratch. Like many successful commercial, real-world systems, the PC has evolved over time as new features have been requested and bolted on, while existing customers demand backward compatibility. As a result, both the x86 architecture and PC computer design have accumulated layers of legacy features. The CISC philosophy is a good fit for this environment. It's common for multiple competing standards to be supported

within single designs, even including multiple choices for x86 assemblers including but not limited to NASM. Recent x86 has extended beyond the features seen in this chapter by adding parallelization, which we'll examine in Chapter 15. But before this, we'll take a breather by looking at developments in the cleaner, more beautiful world of RISC in the next chapter.

Exercises

Creating a Bootable ISO Image

Here you'll create a simple 16-bit "Hello, world!" program, assemble it with NASM into executable machine code, then store this machine code in an ISO file, an image of the contents of a physical secondary storage device that you can use to boot a real PC or a virtual machine.

1. Create the following *hello16bit.asm* file:

```
bits 16                ; tell NASM we're only using 16-bit x86
org 0x7c00             ; base address for bootloader to place this code
section .data          ; this segment is read-write data
message db 'Hello, World!', 13, 10, 0
section .text          ; this segment is read-only code
entry:
  jmp start
printer:               ; subroutine for printing ASCII strings
  lodsb                ; load SI into AL and increment SI [next char]
  or al, al            ; check if the end of the string
  jz printer_end;
  int 0x10             ; otherwise, call interrupt to print char
  jmp printer          ; loop
printer_end:
  ret                  ; return flow
start:
  mov si, message      ; say what we want to print
  mov ah, 0x0e
  call printer         ; print it
                       ; ** add your own code here ... **
  hlt
times 510-($-$$) db 0  ; zero out rest of 512-byte boot sector
dw 0xaa55              ; code to mark sector as bootable
```

2. Run the following commands:

```
mkdir -p cd/boot
nasm hello16bit.asm -o cd/boot/loader.sys
mkisofs -R -J -c boot/bootcat -b boot/loader.sys -no-emul-boot -o cd.iso cd
```

NOTE *If you're using Microsoft Windows, these commands can be run by installing and using the Windows Subsystem for Linux. If you don't already have NASM, install it from* https://nasm.us. *You may also need to install mkisofs for your system.*

3. If everything worked, you'll now have a *cd.iso* file for booting a physical or virtual x86 machine. This will allow you to run on "bare metal" x86, without an operating system getting in the way.

We'll discuss how to boot into your ISO file in the next exercises. When you do, you should see something like Figure 13-11 on the screen.

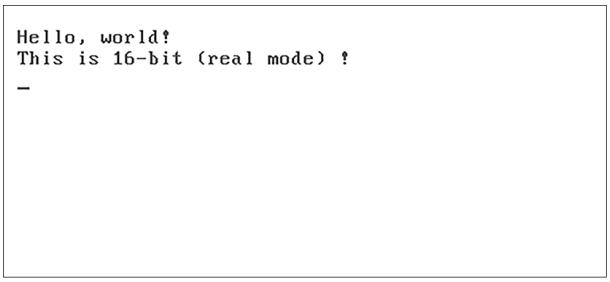


Figure 13-11: The result of booting into a bare metal test program

Before going any further, let's look at what *hello16bit.asm* actually does. In addition to actual x86 instruction mnemonics, a NASM program usually also includes some directives, which are lines that aren't assembled themselves but instead tell NASM to change its behaviors in various ways. The `section` directive tells NASM to change which segment of the output file to write the next assembled instructions to. In some file formats, the number and names of sections are fixed; in others, the user may make up as many as they wish. The Unix object and bin formats all support the standardized section names `.text` (contains executable instructions), `.data` (contains initialized variables), and `.bss` (contains uninitialized variables). The ASCII string includes special ASCII codes 13, 10, and 0 after the human readable letters. What are these? (Hint: See Chapter 2.)

Booting on a Virtual x86

The ISO can be booted on a virtual machine as if it were a physical disk. Follow these steps to try it out using the VirtualBox virtual machine. (Open source Linux users may prefer to use virt-manager at *https://virt-manager.org*.)

1. Visit *https://www.virtualbox.org* for instructions on how to install VirtualBox on your system.
2. Once installed, create a new virtual machine by clicking the **New** icon; use the default settings.
3. Start your virtual machine and "insert" your bootable virtual CD by selecting your *cd.iso* file when asked.

Booting on a Physical x86

The ISO can also be booted on a physical x86 machine if you first "burn" it onto a physical USB stick. Here's how:

1. Use a program such as Etcher (*https://www.balena.io*) for your current operating system to burn the ISO to a USB stick.
2. Once you have a bootable USB stick, you need to tell your PC to boot from it. Your PC is probably currently configured to boot from a hard disk, but it will have some method—which varies by manufacturer—to change to booting from USB as part of its BIOS configuration tools. Editing these settings is called "going into the BIOS." On most machines it's done by holding down a particular key for a few seconds as you turn on the machine. This is often ESC, DEL, F1, F2, F8, F10, or F11, depending on the manufacturer (if it doesn't say which, try running a finger over the whole top row of the keyboard to hit them all). You'll usually see some low-resolution BIOS menus: if you hunt around, there will be some way to specify the boot order and bring USB to the top of it. Some machines may have additional security features that need to be disabled before you can boot from a new device.

Booting to and Programming in 64-Bit Mode

Switching a modern x86 into 32- and 64-bit modes isn't trivial. Due to historical baggage, it requires a couple of screens of instructions and data. How these work is fairly obscure, but luckily it's a standard process that can now be done using the boilerplate code shown here:

```
org 0x7c00          ; base address where this code will be placed (by bootloader)
entry:
    jmp real_to_protected
GDT32:              ; Global Descriptor Table for 32-bit mode
    .Null: equ $ - GDT32
    dq 0            ; defines 32 bits of zeros for the null entry
    .Code: equ $ - GDT32
    dw 0xFFFF       ; segment limit
    dw 0            ; base address
    db 0            ; base address (again)
    db 0b10011010   ; binary flags describing mode
    db 0b11001111   ; binary flags describing mode
    db 0            ; last remaining 8 bits on the base address
    .Data: equ $ - GDT32
    dw 0xFFF        ; --|
    dw 0            ;   | - identical to code segment
    db 0            ; --|
    db 0b10010010
    db 0b11001111
    db 0
```

```
    .Pointer:
    dw $ - GDT32 - 1
    dd GDT32
GDT64:                  ; Global Descriptor Table for 64-bit mode
    .Null: equ $ - GDT64
    dw 0xFFFF
    dw 0
    db 0
    db 0
    db 1
    db 0
    .Code: equ $ - GDT64
    dw 0
    dw 0
    db 0
    db 10011010b        ; binary flags describing mode
    db 10101111b        ; binary flags describing mode
    db 0
    .Data: equ $ - GDT64
    dw 0
    dw 0
    db 0
    db 10010010b        ; binary flags describing mode
    db 00000000b        ; binary flags describing mode
    db 0
    .Pointer:
    dw $ - GDT64 - 1
    dq GDT64
bits 16                 ; tells NASM the following is 16-bit x86 code
real_to_protected:      ; switch from 16 bits to 32 bits
    mov ax, 0x2401
    int 0x15            ; enable a20 gate
    mov ax, 0x3
    int 0x10            ; change video mode
    cli
    lgdt [GDT32.Pointer]
    mov eax, cr0
    or eax, 1
    mov cr0, eax
    jmp GDT32.Code:protected_to_long    ; perform long jump
bits 32                  ; tells NASM the following is 32-bit x86 code
protected_to_long:       ; switch from 32 bits to 64 bits
    mov ax, GDT32.Data
    mov ds, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
```

```
    ; root table - page-map level-4 table (PM4T)
    mov edi, 0x1000     ; starting address of 0x1000
    mov cr3, edi        ; base address of page entry into control register 3
    xor eax, eax        ; set EAX to 0
    mov ecx, 4096
    rep stosd
    mov edi, cr3        ; restore original starting address
    mov dword [edi], 0x2003
    add edi, 0x1000
    mov dword [edi], 0x3003
    add edi, 0x1000
    mov dword [edi], 0x4003
    add edi, 0x1000
    mov ebx, 0x00000003 ; used to identity map the first 2MiB
    mov ecx, 512
    .set_entry:
        mov dword [edi], ebx
        add ebx, 0x1000
        add edi, 8
        loop .set_entry
    mov eax, cr4
    or eax, 1 << 5
    mov cr4, eax
    mov ecx, 0xC0000080 ; magic value actually refers to the EFER MSR
    rdmsr               ; read model-specific register
    or eax, 1 << 8      ; set long-mode bit (bit 8)
    wrmsr               ; write back to model-specific register
    mov eax, cr0
    or eax, 1 << 31 | 1 << 0   ; set PG bit (31st) & PM bit (0th)
    mov cr0, eax
    lgdt [GDT64.Pointer]
    jmp GDT64.Code:real_long_mode
bits 64                 ; tells NASM the following is 64-bit x86 code
printer:                ; subroutine for printing ASCII strings
    printer_loop:
        lodsb
        or al, al
        jz printer_exit
        or rax, 0x0F00
        mov qword [rbx], rax
        add rbx, 2
        jmp printer_loop
    printer_exit:
        ret
real_long_mode:
    cli
    mov ax, GDT64.Data
```

```
    mov ds, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    xor rax, rax         ; clears register rax
    mov rsi, boot_msg    ; say what we want to print
    mov rbx, 0xb8000
    call printer         ; print it
    mov rsi, l_mode      ; say what we want to print
    mov rbx, 0xb80A0
    call printer         ; print it
                         ; ** add your own code here ... **
    hlt
boot_msg db "Hello, world!",0
l_mode db "This is 64-bit (long mode) !",0
times 510 - ($-$$) db 0  ; zero out rest of 512-byte boot sector
dw 0xaa55                ; code to mark sector as bootable
```

If you save this, assemble it, and put it into an ISO as for the 16-bit version, it will boot your real or virtual x86 into 64-bit mode and print another "Hello, world!" message. You can then use the "Hello, world!" program as a starting point, modifying it into your own bootable programs for the following tasks:

1. Write a subroutine that reads integers and converts them into ASCII strings. Extend it to floating point. Use it to print out some numbers along with "Hello, world!"
2. Try porting previous programs from the Analytical Engine and Manchester Baby to run on x86. What's gotten easier or harder to do in modern x86 compared to those systems?
3. Call the BIOS routine to light up pixels on the screen several times to draw a simple shape.

More Challenging

Write a simple game such as *Space Invaders* using the above BIOS calls, on bare metal x86.

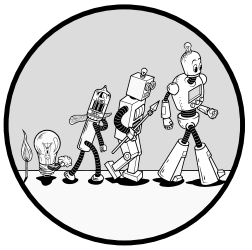
Further Reading

- For the official NASM manual, see "NASM: The Netwide Assembler," *https://www.nasm.us/xdoc/2.13.03/html/nasmdoc0.html*.
- For an overview of x86 history, see P. Lilly, "A Brief History of CPUs: 31 Awesome Years of x86," *Maximum PC*, April 2009, *https://www.pcgamer.com/a-brief-history-of-cpus-31-awesome-years-of-x86*.

- For the full five-volume amd64 reference set, see AMD Technology, *AMD64 Architecture Programmer's Manual Volumes 1–5* (Santa Clara: AMD Technology, 2023), *https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5*.
- For information on 3D graphics programming, see Graham Sellars, *Vulkan Programming Guide* (Boston: Addison-Wesley, 2017).
- For details of how the x86 boot assembly code works, see Gregor Brunmar, "The World of Protected Mode" (*http://www.osdever.net/tutorials/view/the-world-of-protected-mode*), the lame_bootloader GitHub repository (*https://github.com/sedflix/lame_bootloader*), and "Setting Up Long Mode" (*https://wiki.osdev.org/Setting_Up_Long_Mode*).

14

SMART ARCHITECTURES



Smart computing means general-purpose computers built into low-power and/or mobile devices, such as phones, tablets, TVs, routers, and fridges. Unlike embedded systems, they run multiple easily installable and upgradable programs, often known as *apps*. Unlike desktop systems, they need to reduce power consumption, as they often run from batteries. *Reduced instruction set computing (RISC)* is a good fit for these requirements, so RISC architectures are generally found in smart systems. This chapter explores the RISC philosophy, smart devices, and the details of a particular RISC architecture, RISC-V. It's shorter than the desktop chapter because RISC is simpler than CISC by design.

Smart Devices

Early mobile phones were embedded systems whose primary purpose was to function as voice telephones. They had microcontrollers programmed to manage the telephone and user interaction via buttons and simple number-displaying screens. Over time, these microprocessors and interaction devices grew, and the firmware was extended to include features such as contact books, alarm clocks, and simple games like *Snake*.

Modern smartphones now provide such features, and many more, as software apps rather than firmware. They've replaced microcontrollers with fully fledged general-purpose architectures that typically run an operating system such as LineageOS, Replicant, or Android to host apps, much like a desktop PC.

The prefix *smart* emerged to describe these phones, but it's now applied to anything that used to be an embedded system but has been upgraded to a general-purpose computer. For example, smart TVs (the left of Figure 14-1) and smart fridges have moved beyond microcontrollers and firmware to the point where they can easily install and run multiple apps.

The modern consumer internet connection device shown on the right of Figure 14-1 and widely but wrongly known as a "router" is another example of a smart computer. Such devices now usually contain an operating system running many services, including routing, Wi-Fi, firewalling, and a web server (at least to run its configuration page). They should probably be renamed "smart routers" for this reason.

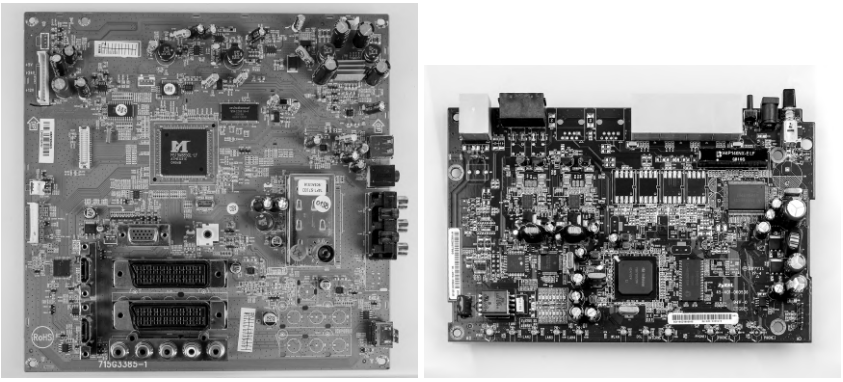


Figure 14-1: The inside of a Toshiba smart TV (left) and a Zyxel router (right)

The "smart home" has been a computing industry ambition for several decades, and refers to a home in which most or all of the usual domestic appliances are upgraded to networked, general-purpose computers. For example, smart washing machines and smart central heating controllers will enable innovative, machine learning–based apps to compete against one another to make the best use of energy, given data from sensors monitoring the condition of clothes to be washed and the temperatures and usages of rooms. Linking these systems will enable automation chains such as your smart fridge predicting that the milk will run out later today, and placing an

automatic order to the local supermarket to deliver more, perhaps via a last-mile delivery robot and a smart drop-off box to receive the delivery. Or your heating system and fridge could be temporarily turned off while the washing machine is turned on, so that it can be powered entirely from the batteries in your parked electric car, charged from your solar panels, without needing to use the power grid.

Architectures for smart devices have similar requirements to embedded systems around reliability and energy usage. However, they also need more computing power than embedded microcontrollers. These requirements are a perfect match for the RISC philosophy, so let's examine this philosophy in more detail.

RISC Philosophy

The RISC concept was invented by an American, David Patterson, but most successfully commercialized by the British. Patterson's quantitative approach to architecture involved a statistical analysis of the instructions actually being used by real-world programs on processors of the 1990s. He found that the more complex instructions were used very rarely, in part because compiler back-end designers didn't know or want to learn how to use them. He determined that roughly 90 percent of the work was being done by roughly 10 percent of the resources. This led him to the central RISC tenet that the silicon occupied by the rarely used instructions would be better put to work making the most popular 10 percent of instructions run very fast, at the expense of removing the other instructions altogether. Patterson and his co-architect, John Hennessy, won the 2017 Turing Award for their research on the use of quantitative methods to guide RISC architecture design.

RISC usually aims for every one of its streamlined set of instructions to be executed in a single CPU cycle. With fewer instructions available, RISC assembly programs are often quite verbose, but each instruction is simple, fast, and low-power to execute. Writing programs in RISC assembly and writing compilers for RISC is easy and fun because the instruction set architectures (ISAs) are small, simple, and understandable.

RISC CPUs themselves aren't necessarily simple, however. While the instruction sets are by definition smaller, designers have found alternative ways to make efficient use of the available silicon. For example, RISC processors typically have many more registers than their CISC counterparts. Extra registers are especially useful in RISC because they help separate memory access from arithmetic. In RISC programming and RISC compilers, it's common to try to bring all relevant variables into registers at the start of a subroutine, then do the entire function's computations in registers, storing only the result back to main memory. This contrasts with CISC, where there can be continual loading and storing throughout the subroutine.

Because part of the RISC philosophy is that every instruction should execute in exactly one clock cycle, instruction-level parallelism through pipelining, branch prediction, and out-of-order execution (OOOE) are massively easier to manage. Every instruction has the same fetch-decode-execute step

duration, and each phase can be triggered regularly in open-loop style. Compare this with CISC architectures, where steps of different instructions can take different durations and must closed-loop trigger each other to say when they've been completed.

RISC was traditionally seen as a very academic philosophy, being beautiful in its design, lean and mean in its execution, and generally resisting the temptation to make a fast buck by bolting on new features to please specific customers by next Thursday. It was often associated with Britain and the British company ARM, though it originated at the University of California, Berkeley—located near to but separated from Silicon Valley. The stereotypical RISC advocate was more interested in beauty and cleverness of design than pragmatism, and for a long time such people were laughed at by more commercial-minded Silicon Valley architects. However, this is now changing: the beauty of RISC is paying off. Most processors now manufactured are RISC. This is largely due to the trend of smart and embedded devices replacing desktops, though serious thoughts are now turning to RISC for cloud servers, too. In 2020, Apple also moved its desktop machines to the RISC-based M1 architecture.

FROM ACORN TO ARM

The British company Acorn used the 6502 in their BBC Micro ("Beeb"), a classic piece of British engineering, brilliantly designed but commercially mistimed and mispositioned. Like much British technology, the Micro was government-funded, in this case via the national broadcaster, the BBC, who wanted a custom-designed, mass-market machine to go with an educational TV series.

Hackers today often play at buying a 6502 and building an 8-bit computer around it on a breadboard, and that was just what the Beeb designers did. Acorn was founded by a bunch of Cambridge people who used that pedigree to convince the establishment BBC to choose their design. The BBC gave a huge specification list for what their computer needed for their TV series. This had a strong education and science—rather than gaming—influence. For example, they didn't include a joystick port but did include options for co-processors and interfacing to maker-style electronics.

One year after the Beeb's release, the Commodore 64 appeared, designed to be "for the masses, not the classes." It had superior graphics and sound for game playing, and came at a much lower price. The C64 filled the full 64 k_2B of addressable RAM, while the Beeb filled only 32 k_2B. The C64 had the glorious SID sound chip, while the Beeb had only basic square waves, white noise, and amplitude envelopes from the poorer SN76489. Commodore had by this point bought out MOS, so it could involve the designers of the 6502 and related chips—including Chuck Peddle—directly in its computer designs to exploit its most advanced features.

The C64 quickly made the Beeb seem overspecified and overpriced. However, Acorn used the Beeb internally to design their first RISC processor, the Acorn RISC Machine (ARM), for their next computer, the Archimedes—a fully 32-bit machine, released in 1987. Archimedes was technically a decade ahead of its time, though temporally and culturally it still belonged to the "16/32-bit era." It was again sadly mismarketed, overspecified, and overpriced—for example, having eight-channel audio compared to the Amiga's four, yet no joystick ports or TV output for games.

Acorn went on to spin out a new company, ARM, to focus on its ARM chip designs. ARM found success, as its chips now power most of the world's smartphones, the M1 used in Apple's tablets and desktops, and many smart devices such as the chips seen in Figure 14-1.

Computing history could have been very different if the BBC had waited an extra year to put a C64 into every UK school. Commodore's business management faced challenges that led to bankruptcy in 1994, but acceptance by the British establishment might have provided the stability needed to survive.

A strange legacy has been that Acorn's OS for Archimedes, RISCOS, is still compatible with the latest ARM ISAs. It was painstakingly hand-coded in ARM assembly to squeeze power out of early 1990s CPUs, so it now runs blindingly fast on modern devices such as the Raspberry Pi. There's been a resurgence of old Acorn user groups around this, in some cases resulting in old friends meeting up for the first time in 30 years.

RISC-V

ARM's RISC ISA designs are heavily patented, but others in the industry now wish for fully open source RISC ISA alternatives. As with the IBM-Intel-AMD license, computer builders want to genericize the ISA and enable multiple implementations to drive down processor prices through competition. A large consortium of major Silicon Valley companies is therefore now backing Patterson's latest and fully open source RISC ISA family design, called RISC-V (V for, and pronounced, "five"), as the next standard for RISC. Note that RISC-V is a family of ISAs, not a hardware implementation of these ISAs. Companies may implement the open source ISAs in proprietary hardware and compete on implementation quality. There are also fully open source implementations by Patterson's group and other members of the RISC-V movement.

Understanding the Architecture

RISC-V is designed as a family of ISAs, rather than a single ISA. The family includes versions suitable for embedded, mobile, desktop, and server machines, including versions for 32-, 64-, and 128-bit ISAs. RISC-V defines a core ISA of instructions that all RISC-V systems need to implement. Like x86, RISC-V uses per-byte, little-endian addressing. RISC-V uses RISC-style instructions that separate memory access from arithmetic logic unit (ALU)

operations. It's not an accumulator architecture, and ALU instructions explicitly specify the output register, so they usually have three arguments.

As we've discussed, the RISC philosophy is to reduce the number of instructions as much as possible. This means that some operations that you would usually expect to find in an ISA are absent if they can be achieved through other means, such as by calling other instructions in slightly odd ways. When such unusual uses of instructions are needed, RISC-V assemblers will sometimes provide *pseudo-instructions* that look like the classic instructions you expect. The assembler then converts them to the underlying, somewhat ugly RISC instructions. To a limited extent, some of the complexity of CISC digital logic is thus moved into the assembler, while keeping the machine code itself clean.

RISC-V registers have standard names and standard conventions for their typical uses, such as how to pass arguments to subroutines by storing them in registers. For example, the integer registers are always called x0 to x32 (and x0 always contains the constant 0). The RISC designers gave the integer registers secondary nickname mnemonics, shown in Table 14-1, to encourage conventions for their use.

Table 14-1: RISC-V Integer Registers

Name	Mnemonic	Intended convention
x0	zero	Value is always zero
x1	ra	Return address for subroutine calls
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–x7	t0–t2	Temporary
x8–x9	s0–s1	Saved
x10–x14	a0–a7	Arguments for subroutine calls
x18–x27	s2–s11	Saved
x28–x31	t3–t6	Temporary

As with instructions and pseudo-instructions, this is designed to keep the underlying architecture very clean and simple, while also providing the ability to think and code in terms of some CISC-like styles, if and only if the programmer wants to do that.

There may be a further 32 optional floating-point registers, also given mnemonics, which you can see in Table 14-2.

The intended calling conventions for registers starting with *t* and *s* are temporary and safe—the same concepts seen but less standardized in the x86 calling conventions from Chapter 13.

Table 14-2: RISC-V Floating-Point Registers

Name	Mnemonic	Intended convention
f0—7	ft0—7	Floating-point temporaries
f8—9	fs0—1	Floating-point saved registers
f10—11	fa0—1	Floating-point arguments/return values
f12—17	fa2—7	Floating-point arguments
f18—27	fs2—11	Floating-point saved registers
f28—31	ft8—11	Floating-point temporaries

As with x86, the user can't access internal registers directly. These internal registers include a program counter and status register.

Programming Core RISC-V

Now we've seen the basic structures, let's use them in some instructions to write RISC-V programs. As with the other architectures we've studied, we'll first introduce data moves and control flow. RISC-V then has various optional extensions, including arithmetic, which we'll also explore.

Data Moves

As a RISC system, data movement involving primary memory is cleanly separated from all other operations, which are performed only on data in registers. Loading from memory to registers, and storing to memory from registers, is done by the following instructions:

```
lw x5, x6, 0          ; load word to x5, from content at address x6+0
sw x5, x6, 0          ; store value from reg x5 to address x6+0
la x6, mylabel        ; load address of mylabel (not its content) to x6
```

Note that like NASM and Arduino assembly, RISC-V assembly writes the destination register first, followed by the inputs. You can see the RISC nature of the `lw` and `sw` instructions in the requirement for all three operands, even when the third operand is a 0 and thus isn't being used, rather than providing a second form of these instructions with only two operands.

For convenience, a pseudo-instruction is provided to load content from a labeled address:

```
lw x5, mylabel        ; load content at address mylabel to register x5
```

This actually gets assembled to two instructions. First, an `la` obtains the address of the label, and then an `lw` loads the content of that address.

Importantly, unlike in CISC, these instructions aren't reused to copy data between registers. This clean separation of memory access instructions from between-register operations is often considered to be a defining feature of RISC. We'll discuss how this is done in the "Arithmetic" section on page 347.

Control Flow

Unconditional jumps come in two forms:

```
j mylabel              ; jump to address mylabel
jr x5                  ; jump to address in x5
```

Conditional jumps include:

```
beq x1, x2, mylabel    ; branch if x1==x2
bne x1, x2, mylabel    ; branch if not x1==x2
blt x1, x2, mylabel    ; branch if x1<x2
bge x1, x2, mylabel    ; branch if x1>=x2
```

Subroutines are called by "jump and link." The "link" here refers to saving the program counter in a register. For example:

```
jal x1, mylabel        ; store current PC in x1 and jump to mylabel
jalr x1, x2, 0         ; store current PC in x1 and jump to address x2+0
```

This is why x1 is nicknamed ra, for return address.

There isn't a return instruction because returning can be done, RISC-style, by reusing the jump instruction, jumping to what was previously saved in x1:

```
jalr x0, x1, 0
```

That said, a `ret` pseudo-instruction may be provided and assembled into the appropriate `jalr` instruction.

Using `ret` enables you to call and return from a single subroutine, with the return address kept in x1. To call nested functions, however, you'll also need a stack. The convention is to have the register x2 (sp) be the stack pointer. Here we push a 4-byte word to the stack:

```
addi sp, sp, -4 ; grow stack
sw   a0, sp, 0  ; store a0 onto stack
```

The `addi` instruction means "add immediate," in this case adding a constant (−4) to the stack pointer. ("Immediate" means that an operand contains the value itself, rather than an address or register containing the value.) Likewise, here we pop from the stack:

```
lw   a0, sp, 0  ; retrieve data to a0 from stack
addi sp, sp, 4  ; shrink stack
```

Note that this is done, RISC-style, by reusing existing instructions. Unlike CISC style, there are no additional stack instructions such as `push` and `pop`. Rather, you have to manage the stack yourself with the reduced instruction set.

Extending RISC-V

RISC-V also defines many plug-ins or libraries of additional instructions that can be implemented optionally. Each of these extensions is given a single-letter code, including:

- **I** Basic integer add/subtract/shift/bitwise logic
- **M** Integer multiplication and division
- **B** Bitwise Booleans
- **F** Single-precision floating point
- **D** Double-precision floating point
- **Q** Quad-precision floating point

To specify a particular ISA, we write "RV" for RISC-V, then the word length, then the extensions used. For example, RV64IMF means "RISC-V, 64-bit, with extensions I, M, and F." This design is intended to allow RISC-V to span all applications from embedded systems (such as RV8I) to high-end scientific computing clusters. Further standard extensions may be proposed as new letters or as arbitrary strings beginning with a Z, and local experimental extensions may be proposed as arbitrary strings beginning with an X.

Arithmetic

Integer arithmetic is performed with three operands. For example:

```
add x6, x7, x8  ; x6 := x7 + x8
sub x6, x7, x8  ; x6 := x7 - x8
mul x6, x7, x8  ; x6 := x7 * x8
div x6, x7, x8  ; x6 := integer of x7 / x8
rem x6, x7, x8  ; x6 := remainder of x7 / x8
```

Bitwise Boolean operations are similar:

```
and x6, x7, x8  ; x6 := x7 bitwise-and x8
or  x6, x7, x8  ; x6 := x7 bitwise-or x8
xor x6, x7, x8  ; x6 := x7 bitwise-xor x8
not x6, x7      ; x6 := bitwise-not x7
```

Unusual cases such as division by zero and overflows are reported in the status register, which can be queried by further instructions.

In a RISC architecture, register-to-register operations belong to arithmetic rather than to data transfer in order to reduce the number of instructions and variations. Thus, there are no extra instructions for placing constants in registers or copying data between registers. Instead, we treat these as addition operations, with the always-zero x0 register used as one of the operands. For example:

```
addi x1, x0, 3   ; load immediate integer 3 to x1
add x2, x1, x0   ; copy x1 to x2
```

The RISC-V assembler developed by the mainline RISC-V project includes alternative pseudo-instructions for these operations, for programmer convenience:

```
li x2, 3         ; load integer 3 into x2
mv x2, x1        ; copy x1 to x2
```

These are assembled behind the scenes into the appropriate addition instructions.

Floating Point

Floating-point instructions begin with `f` and act on the floating-point registers, f0 to f31, such as:

```
fadd f6, f7, f8  ; f6 := f7 + f8
fsub f6, f7, f8  ; f6 := f7 - f8
fmul f6, f7, f8  ; f6 := f7 * f8
fdiv f6, f7, f8  ; f6 := f7 / f8
fsqrt f6, f7     ; f6 := sqrt(f7)
```

There are also instructions to load, store, and compare floats, and convert them to and from integers:

```
flw f1, t0, 0    ; load float word to f1 from address t0+0
fsw t0, f1, 0    ; store float word to address t0+0 from f1
flt.s x6, f1, f2 ; x6 := (f1 < f2)
fcvt.w.s x6, f1  ; convert float f1 to int x6
fcvt.s.w f1, x7  ; convert int x7 to float f1
```

Here the `.s` and `.w` stand for single and word precision. There's also `d` for double. These suffixes are similar to, and perhaps borrowed from, the 68000 discussed in Chapter 11. As with integers, RISC style can be seen in the clear separation of memory access (load and store) from the arithmetic performed.

Different RISC-V Implementations

As I've mentioned, RISC-V is a specification of a family of ISAs of the machine code interface between the programmer and the CPU. It doesn't specify how the instructions should be implemented. Architects are free to design their own implementations of RISC-V as CPUs from digital logic (or from anything else).

So far, there are three major open source hardware implementations of RISC-V. These are fully open source hardware in the sense that anyone can download, edit, and fabricate the files describing the CPU layouts, free of charge. The implementations are:

Berkeley Educational cores These are deliberately simple implementations of some limited RISC-V ISAs for educational use, being easier to understand and modify. They include non-pipelined and simple pipelined versions.

Rocket This is a family of CPU implementations using professional-quality pipelining. It's a family rather than a single CPU because there are versions for different word lengths and for most of the ISA extensions. The Rocket chip generator program can generate a particular chip design and layout for most RISC-V descriptors.

BOOM (Berkeley Out-of-Order Machine) This is a high-performing implementation, using state-of-the-art OOOE. It's a platform for much active research on OOOE and other hardware speedups.

RISC-V Toolchain and Community

RISC-V is more than just an architecture: it's an open source community and ecosystem. For the first time in architecture history, a fully open source architecture and toolchain now exists that allows anyone in the world to download it, hack it, and burn it onto cheap field programmable gate arrays (FPGAs). These kinds of technologies used to be the preserve of a tiny group of professionals working at architecture companies, where they were closely guarded secrets. Now anyone can access and play with the same kinds of tools as those big companies. As a result of this opening up of the toolchain, together with the pressure to develop radically new architectures due to the end of Moore's law(s), there are now over 700 architecture start-up companies in Silicon Valley, and more around the world. Hennessy and Patterson's 2017 Turing Award lecture declared a "new golden age" of architecture for the 2020s, and they encouraged everyone to get involved with this community.

To get involved yourself, you'll need to download the RISC-V community's tools and tutorials. RISC-V development is *de facto* led by Patterson's group at the University of Berkeley, near Silicon Valley. This group has produced a standard set of tools that the community uses to design and build structures from transistor layouts all the way up to full CPUs. RISC-V development is usually done in the program Chisel, and indeed Chisel is developed by many of the same people as RISC-V.

Smart Computer Design

For smart computing applications, it's usually desirable to place a RISC CPU on the same silicon as all the other components, such as memory and I/O, needed to make a complete computer. Such a chip is known as a *system-on-chip (SoC)*. This is a superficially similar idea to embedded microcontrollers, but with substantially larger and more powerful designs. An SoC is then typically mounted on a very small PCB together with only analog electronics for power management and physical I/O connectors.

Computer designers are now taking RISC-V chip designs and putting them onto SoCs and hardware boards. There are several commercial and research systems using silicon fabrications of RISC-V, including the following examples:

HiFive This closed source, commercial RISC-V product, designed by SiFive, was the first practical RISC-V hardware available to the public. It's a $50 Raspberry Pi–style board that uses an OOOE implementation, capable of running Linux for applications, similar to the Raspberry Pi.

Mango Pi This is a RISC-V board with similar small form factor and capabilities to the Raspberry Pi Zero.

lowRISC This is an ongoing project to design and produce a fully open source hardware computer as an SoC, based on a Rocket CPU. To make a full computer, all the other non-CPU components also need to be designed as open source hardware, in particular I/O and devices for communications, such as USB and Ethernet controllers.

ROMA This is the first laptop design based on RISC-V. It was released by Xcalibyte in 2022.

Beyond RISC CPUs, smart computing requirements have also led to developments in memory and I/O. Let's look at some of the most common resulting computer design elements here.

Low-Power DRAM

The DRAM used in phones and other mobile devices is of a special low-power (LP) type known as *LP-DRAM*. LP-DRAM is designed to reduce battery usage at the cost of some loss of speed and convenience. Primarily, this is achieved by turning off the power to large areas of memory that aren't in use. This destroys their volatile contents but greatly reduces power consumption, as no electricity is needed to continually refresh the memory. The main cost is a delay in reactivating these regions when they're needed again. For example, your phone's batteries will last longer on an LP-DRAM machine if you close down all unnecessary apps, because the operating system will then free the memory they were using, which can be turned off to save power.

Like DRAM, LP-DRAM has been through many standards iterations. Various additional innovations beyond power switching include operating on reduced voltages such as 1.8 V, scaling the refresh rate as a function of temperature to reduce unnecessary refresh work, and multiple levels of shutdowns. The latter may be used to differentiate between a user putting their phone away in their pocket for many hours versus temporarily freeing memory from an app while continuing to use the rest of the phone.

Cameras

Camera sensors, as shown in Figure 14-2, are active pixel sensors made from CMOS-like chips.

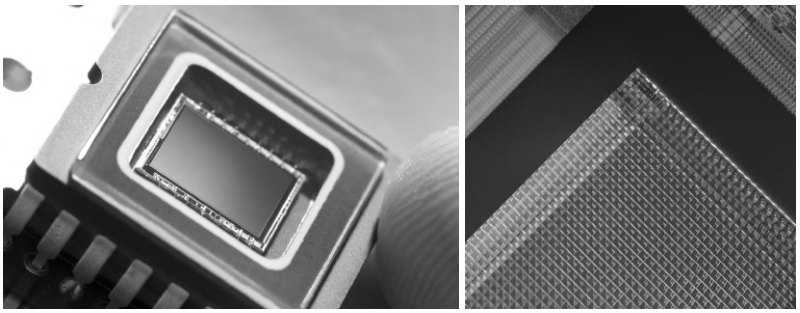


Figure 14-2: A camera sensor (left) and a close-up showing its pixels (right)

Camera sensors are formed from 2D arrays of light sensors (pixels) created through photolithography, like chips. Usually each pixel contains three subpixels for sensing red, green, and blue light, as in displays.

Touchscreens

The touchscreen used in a phone or tablet is produced as a distinct, transparent layer from the display screen placed beneath it. Like chips, touchscreens are produced via photolithography; layers of different materials are laid down in a pixel grid of small half-capacitors, as shown in Figure 14-3.

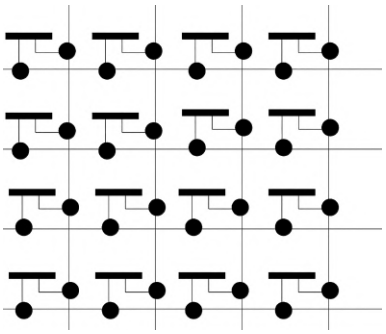


Figure 14-3: A touchscreen made from an array of half-capacitors, with 2D addressing

Human skin acts as the other half of the capacitor when in close proximity to these pixels, making the grid touch-sensitive.

To enable the touchscreen to act as a layer above a visible display, we need to build these half-capacitors and the wires connected to them from a material that's both conductive (a metal) and also transparent to human-visible red, green, and blue light. This is a difficult requirement, because metals generally reflect all frequencies of light. Indium tin oxide (ITO) is a

very unusual compound, based on the rare element indium, that happens to have the desired property, so it's used in most touchscreens.

NOT SO DIFFERENT AFTER ALL?

Unlike CISC, RISC doesn't add extra instructions to make life easier for the programmer, who instead must often make use of sequences of more basic, general-purpose instructions with particular operands. This can make manual RISC assembly programming less interesting than CISC programming, but RISC assemblers can provide pseudo-instructions that function similarly to CISC instructions but are assembled into sequences of multiple RISC instructions. It's also possible to build CISC architectures that do similar but in digital logic, fetching CISC-style instructions but decoding them into *sequences* of RISC instructions that are then executed in RISC style. Such designs look internally quite similar to the CISC microcode structures, suggesting that CISC and RISC don't have to be so different after all.

Summary

Smart devices are replacing embedded systems in many applications as the cost and power consumption of general-purpose computing falls and battery technology improves. RISC architectures are a good fit for smart computing needs, as their simplicity can reduce physical size, cost, and power requirements.

RISC architectures use small sets of simple instructions. They typically make a clean separation between instructions for memory access and for arithmetic. They try to have all instructions execute in the same time in order to simplify execution and enable smoother pipelining and OOOE. RISC assembly code is usually characterized by its appearance as homogeneous-looking lists of instructions with triple operands.

RISC-V is an open source family of RISC ISAs, with both open and closed source digital logic implementations and design toolchains available. RISC-V includes a core instruction set and various optional extension instruction sets, so variants can be used for small, cheap smart devices, and all the way up to higher-power servers.

Exercises

RISC-V Programming

1. Install and run the Jupiter RISC-V simulator from *https://github.com/andrescv/Jupiter*.
2. Enter some simple programs from the examples in this chapter. Jupiter requires a label called `__start` (with two underscores) to be defined and made global; this will be used as the entry point when the program runs. For example:

```
.globl __start
__start:
  li t0,0
```

3. If you have data lines in your program, then by default Jupiter assumes a Harvard-style segmentation, which requires `.data` and `.txt` sections, such as:

```
.globl __start
.data
  mylabel: .word 17
  myfloat: .float 34.56
.text
__start:
  lw x5, mylabel ; load word to register x5, from content at address mylabel
  la x6, mylabel ; load address to x6, of mylabel (not its content)
myloop:
  sw x5, 0(x6)   ; store value from reg x5 to address 0+x6 (= mylabel)
  j myloop
```

4. Save and assemble each program (**Run ▸ Assemble**) and run it in the simulator. You can place breakpoints at any time with the tick-boxes on the left of the lines, and inspect the registers and memory with the GUI on the right. To get back to the code, click **Editor** at the top left.

Challenging

1. Try porting previous programs from the Analytical Engine and Manchester Baby to run on RISC-V. What's easier or harder to do in modern RISC-V compared to those systems? How does it feel compared to x86?
2. Obtain a physical RISC-V board and use its tools and documents to run the same programs on it.

More Challenging

Want to make a real, working CPU in your bedroom? Using RISC-V and Chisel, you can.

1. A full Chisel tutorial can be found at *https://github.com/ucb-bar/chisel-tutorial*. Install Chisel and work through this tutorial.
2. All of the microcircuits used in Rocket and BOOM—including ALU, FPU, and control unit—are available as Chisel libraries. Download and build some of these and experiment with how they work.
3. The Rocket Chip Generator (*https://github.com/chipsalliance/rocket-chip*) is a tool that takes a RISC-V CPU descriptor code such as RV64IMFP as input, and outputs Chisel and Verilog files (or a C++

simulation) for the desired CPU. Install and run Rocket Chip. Study the outputs to find out how the microcircuits you looked at in the previous problem are used in the generated CPU.

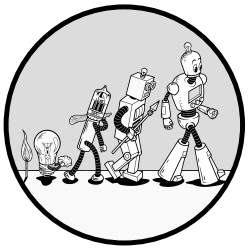
4. Torture (*https://github.com/ucb-bar/riscv-torture*) is a tool provided by the RISC-V community to test correct RISC-V execution in hardware designs, and to help locate errors. Install it, introduce a deliberate error to a Rocket Chip design, and use the tool to study the error.
5. Buy a cheap FPGA board and use the RISC-V documentation and mailing list archives to figure out how to burn your Rocket Chip netlist onto it to make a real physical CPU.
6. Join the RISC-V community discussions at *https://riscv.org* and study the open lowRISC designs at *https://github.com/lowrisc*. Use them to find an interesting piece of work that needs doing and contribute it to the RISC-V community.

Further Reading

- For a detailed RISC-V tutorial, see Edson Borin, *An Introduction to Assembly Programming with RISC-V*, *https://riscv-programming.org/book/riscv-book.html*.
- For the definitive RISC-V manual, see Andrew Waterman and Krste Asanović (eds.), *The RISC-V Instruction Set Manual Volume I: User-Level ISA* (Berkeley: RISC-V Foundation), *https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf*.

15

PARALLEL ARCHITECTURES



As we've discussed, computing is splitting along two paths: low-power systems, forming the Internet of Things; and high-power computing centers, forming the cloud. In previous chapters, we've looked at the low-power IoT side of the split: embedded and smart systems. This chapter will look at the high-power, high-performance systems found in the cloud. Specifically, we'll look at parallelism, the backbone of cloud computing.

The rise of parallelism is related to Moore's two laws. While Moore's law for density says we can still put more and more transistors on chips, Moore's law for clock speed is now over, meaning we can't clock single CPUs faster anymore. The number of fetch-decode-execute cycles per second is no longer increasing, so we need to find new uses for the extra available transistors to try to do more work *within* each cycle, rather than making faster cycles.

For a while, we got away with using the extra silicon to boost classical, serial architectures: we made more and more complex CISC instructions to get more work per instruction; we added more and bigger registers levels

of cache onto the CPU silicon; we replicated structures such as arithmetic logic units (ALUs) to enable simultaneous execution of branches; and we constructed fancier pipelines and out-of-order machines. Together, these techniques have recently delivered double digit–percentage yearly gains in instructions per cycle (IPC) rather than cycles per second. But we may be running out of easy wins in these areas, so we have to think more in terms of digital logic being inherently parallel. Luckily for us, it is.

We've already encountered register- and instruction-level parallelism. Register-level parallelism is the simultaneous per-column execution of digital logic acting on bits of a register. For example, all the bits in a word can be negated at the same time rather than in sequence. Instruction-level parallelism includes pipelining, branch prediction, eager execution, and out-of-order execution (OOOE). These concepts don't appear at the instruction set architecture (ISA) level; they're invisible to the assembly programmer. From the programmer's perspective, they just make serial programs execute faster.

In this chapter, we'll focus on the higher-level parallelisms that are visible in the ISA and therefore may require the attention of the assembly and perhaps also the high level–language programmer. We'll begin by thinking about parallel foundations. Then we'll turn to the two main types of parallelism: *single instruction, multiple data (SIMD)*, as found in modern CPUs and GPUs, and *multiple instruction, multiple data (MIMD)*, as found in multicores and cloud computing centers. Finally, we'll wrap up by considering more radical, instructionless forms of parallelism that might take architecture beyond the concepts of CPUs and programs.

Serial vs. Parallel Thinking

Most of the silicon in a serial computer is used to form memory that sits around doing nothing until it's called on to load from or store to the CPU. In this sense, serial computing is like having 1,000 people send all their work to a single worker, then stand around waiting for the results to come back to them. This effect is known as the *serial bottleneck*.

Parallel computing frees these 1,000 people to all work for themselves. Each becomes an active unit of computation: they pass data directly to one another as needed, and they get massively more work done than if they were standing around waiting for that one worker. Similar gains can occur if we use all the digital logic in a computer to constantly perform computation rather than wait around for the CPU.

Parallel thus seems to be obviously faster and better than serial computing. But at least until the 2010s, computer scientists tended to get stuck in "serial thinking." Most people are at some point taught the concept of programming using a recipe, assuming that only you are in the kitchen and that you're going to perform a sequence of tasks, such as:

```
1. chop vegetables
2. boil water
3. chop chicken
```

```
4. brown chicken
5. add vegetables to pot
6. add chicken to pot
7. simmer pot
8. chop herbs...
```

This is fine on a small scale, but if you're a head chef running a chain of popular restaurants, you'd be in charge of a team of workers, and you'd have to schedule in optimal ways to produce the food more efficiently. The field of operations research is all about optimally scheduling work like this.

How do we take a sequence of instructions like the chicken soup recipe and get everything done in the shortest period of time? There are well-known algorithms to do this. For example, Henry Gantt's charts, like the one shown in Figure 15-1, are used to display and reason about sequences of tasks running in parallel over time.

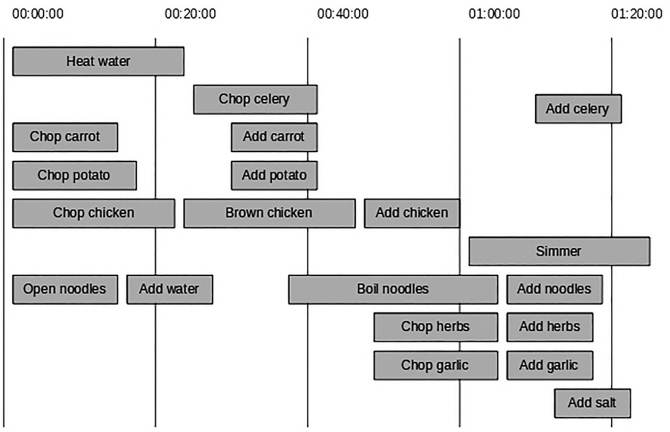


Figure 15-1: A parallel Gantt chart for cooking chicken soup

Simple algorithms exist to generate optimal timings for the tasks, given a list of dependencies—that is, a list of which tasks depend on the completion of which others before they can begin. A critical path can be calculated for the network, which is the sequence of jobs that need to be done on time because they're the bottlenecks.

Bletchley Park made heavy use of this style of computation. Machines weren't the only types of computers used there: it was still the era of human computation, where "computer" was a human job title. Human computers would sit in a computing division (Figure 15-2), all doing parts of computations under a manager allocating and scheduling the work in parallel. These programmer managers thought about how to break down a large mathematical computation into components, distribute the tasks, and collect the results together.



Figure 15-2: A human computing division working in parallel, with a manager (standing) scheduling the work

Given that the management of teams of parallel workers has existed for a long time, and is based on how to design a program of work to efficiently accomplish a task, why do so many programmers basically ignore it and think instead in terms of recipes and serial computing? If the history of computing had been different and started from an operations research perspective rather than from serial algorithms, we might have had a much better foundation. Programming—and perhaps the foundations of computer science—is now having to move toward parallel thinking due to the end of Moore's law for clock speed. For example, today's school children might write their first ever program in Scratch with multiple sprites all running code in parallel. And professional programmers are increasingly having to think in terms of SIMD and MIMD, which we'll study next.

Single Instruction, Multiple Data on CPU

Our first type of parallelism—single instruction, multiple data—means we're going to take a single instruction (for example, "add one") and execute that instruction uniformly on multiple data items at once. We can split SIMD systems into CPU- and GPU-based implementations. Here we'll look at the CPU-based implementation; in the next section, we'll look at the GPU-based one.

Introduction to SIMD

SIMD on CPU is a very CISC-style approach: it involves creating additional instructions and digital logic to perform parallel operations as single instructions. SIMD instructions pack more than one piece of data into a word, then define instructions to apply the same instruction to each piece of data in parallel. For example, on a 64-bit machine, instead of using a 64-bit register to store one big 64-bit integer, we can partition it into four 16-bit chunks that

each hold one 16-bit integer. We can then use instructions that understand this packing and operate on all four chunks at the same time.

In a standard CPU, you might have an `ADD` instruction that adds integers from registers r1 and r2, storing the result in r3. In an SIMD machine, however, you'll have an instruction called something like `SIMD-ADD`, still using the same three registers, but using a different data representation to perform addition on pairs of 16-bit values from the two registers simultaneously; it then stores the output in the third register, packed similarly.

NOTE *SIMD instructions originated in early supercomputers, such as the famous 1960s Cray supercomputers. SIMD was first brought from supercomputers to desktops by Intel via their MMX instructions.*

SIMD can split a 64-bit register into two 32-bit chunks, four 16-bit chunks, or eight 8-bit chunks. The four-way split is especially useful for 3D games. It's common for 3D programmers to represent 3D coordinates using *four*-dimensional vectors, with the fourth dimension serving as a scaling factor to enable *affine transformations*. These are transformations like translations and rotations computed using simple matrix-vector multiplications. The 16-bit precision of the numbers is usually acceptable for games (though maybe not for serious scientific 3D simulations). We're lucky to live in a world whose number of dimensions, when affinated, is a power of 2!

SIMD is also a good fit for images and video, in which pixel colors are often represented by four numbers for RGB and alpha (as discussed in Chapter 2). More generally, for most types of multimedia, including audio, it's common to need to do many copies of the same operations for signal processing, so SIMD can speed this up even when there's no obvious 4D structure.

SIMD instructions can be created for use with any ordinary registers, but they've become more interesting as register sizes have increased to 64 bits. Some architectures also include extra registers that are longer than their word length, known as *vector* registers; these can store 128, 256, or 512 bits, and they're intended primarily for use with SIMD instructions.

Now that we understand the theory of CPU SIMD, let's look at a concrete example of how it's implemented in x86.

SIMD on x86

We saw the names of the various x86 architectures of the 64-bit era in Chapter 13. Beyond the basic amd64 instruction set, most of these architectures have focused on adding extensions using different forms of parallelism. Most of these ideas originated in high-performance computing and high-end servers but have also been introduced to desktop architectures.

The classic CISC approach has been to use the extra transistors to add more simple machines and instructions to the ISA, each intended to do more work than the regular instructions. This has led to thousands of new CISC instructions, added for all manner of special cases such as cryptography, multimedia processing, and machine learning. There have been

disagreements over the standards for these extensions. Everyone implements the same base amd64 ISA, but different manufacturers extend it in different ways to add extensions. They try to get users hooked on their versions and to desert competitors (a well-known strategy called *embrace-extend-extinguish*). This creates headaches for compiler writers who have to create multiple back-ends to optimize for the different extensions.

Most of the new registers and instructions added to x86 during the 64-bit era have been for SIMD. Figure 15-3 shows the complete user register set of modern amd64.

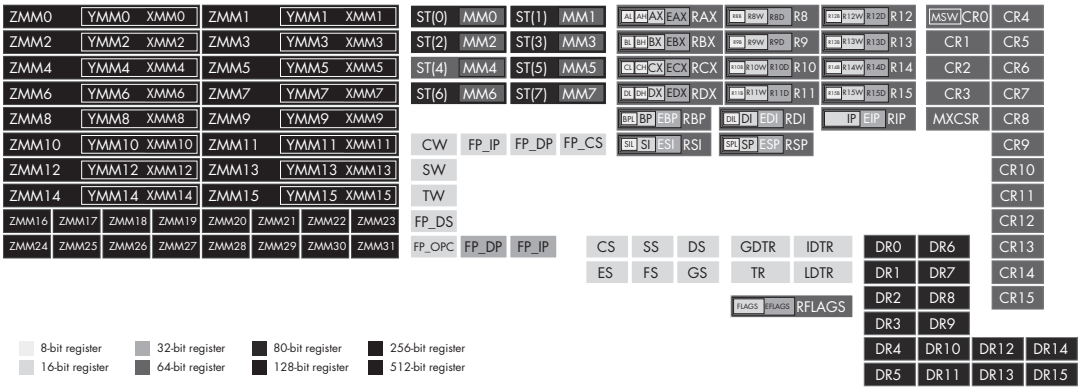


Figure 15-3: The full register set for amd64

The SIMD registers are the ones with "MM" in their names. Notice how new SIMD registers have appeared over time, usually by extending an existing register to have more bits. When extensions are made, x86 backward compatibility requires the original shorter form to still be named and usable, as well as the extended form. This requires many different versions of instructions to be provided.

MMX

MMX was the first x86 SIMD extension. It's never been officially defined what MMX stands for, and indeed this has been a matter of legal trademarking debate between Intel and AMD. Suggestions include "matrix math extensions" and "multimedia extensions."

MMX extended the previous amd64 floating-point registers to 64 bits, similar to how 32-bit registers, such as EAX, were extended to the 64-bit RAX. The new registers have names MM0 through MM7 and still exist on modern machines.

Each MMX register can be used for integer-only SIMD as either a single 64-bit integer, two 32-bit integers, four 16-bit integers, or eight 8-bit integers. Integer SIMD is particularly useful and fast for processing images, including for 2D sprite-based games and video codecs.

MMX instructions begin with `p` for "packed," such as `paddd` for "packed add doubles." New move instructions—`movb`, `movw`, and `movd`—copy arrays of bytes, words, or doubles into single MMX registers. For example, the following defines two arrays of 32-bit doubles: $a = [4, 3]$ and $b = [1, 5]$. It loads a as

packed doubles into MM0 and *b* into MM1. It then packed adds the doubles, leaving [5, 8] in MM0:

```
a:      dd  4, 3
b:      dd  1, 5
main:
        movd    mm0, [a]
        movd    mm1, [b]
        paddd   mm0, mm1
```

MMX adds a *lot* of new instructions, because every arithmetic operation has to exist in each of the packed forms for bytes, words, and doubles.

SSE

Intel's version of x86 SIMD has been extended several times since MMX, as SSE, SSE2, SSE3, SSE4, and SSE4.2 (where SSE stands for streaming SIMD extensions). AMD's latest incompatible competitor is, confusingly, called SSE4a. Unlike MMX, the SSE series provides for floating-point SIMD as well as integers. This makes it particularly useful for accelerating 3D math for games and other physics simulations. (MMX was unsuccessful by the benchmarks of its time, which focused heavily on the 3D game *Quake*.)

Unlike MMX's extension of the old floating-point registers, SSE adds completely new, 128-bit vector registers, called XMM0 through XMM31. The number of these has grown with the SSE versions. They can be split into 8-, 16-, 32-, or 64-bit chunks, with chunks representing either floating points or integers. Each arithmetic operation thus has many instructions depending on these choices.

Most SSE instructions have the letter `p` for "packed" added to their mnemonics, at either the beginning or the end. For example, the top-left of Figure 15-4 shows an SSE compare for equality with the `cmpeqps` instruction. The name comes from `cmpeq`, the standard x86 instruction, plus `ps` to indicate "packed, single-precision."

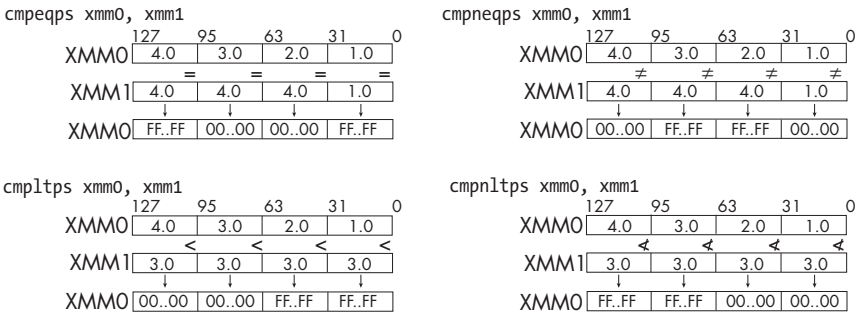


Figure 15-4: The contents of two SSE registers, XMM0 and XMM1, as SSE instructions are carried out, comparing the two sets of data in different ways: equality (top left), inequality (top right), less than (bottom left), and not less than (bottom right)

In the top-right of Figure 15-4, the `compneqps` instruction similarly extends `cmpneq` (compare not equal) to SSE.

The following code shows examples of getting arrays of floats in and out of SSE's XMM registers and performing arithmetic on them:

```
;from en.wikibooks.org/wiki/X86_Assembly/SSE, CC BY 3
section .data
    v1: dd 1.1, 2.2, 3.3, 4.4    ; first set of four numbers
    v2: dd 5.5, 6.6, 7.7, 8.8    ; second set

section .bss
    v3: resd 4    ; result

section .text
    _start:

    movups xmm0, [v1]   ; load v1 into xmm0
    movups xmm1, [v2]   ; load v2 into xmm1

    addps xmm0, xmm1    ; add
    mulps xmm0, xmm1    ; multiply
    subps xmm0, xmm1    ; subtract
    movups [v3], xmm0   ; store result in v3

    ret
```

Here, the `addps` instruction adds the four numbers in XMM1 to the four numbers in XMM0, and stores the result in XMM0. For the first float, the result will be 1.1 + 5.5 = 6.6. The `mulps` instruction multiplies the four numbers in XMM1 with the results from the previous calculation (in XMM0), and stores the result in XMM0. For the first floats, this result will be $5.5 \times 6.6 = 36.3$. The `subps` instruction subtracts the four numbers from `v2` (in XMM1, still unchanged) from the result of the previous calculation (in XMM0). For the first float, its result will be $36.3 - 5.5 = 30.8$.

AVX

Two generations of *Advanced Vector Extensions (AVX)* have added longer vectors than SSE, having 256- and 512-bit lengths. The new 256-bit registers are called YMM0 through YMM31, and the new 512 registers are called ZMM0 through ZMM31.

AVX instructions often have the same names as, and behave similarly to, SSE instructions, but they start with a `v`. For example, to add eight pairs of 32-bit (double) floating-point numbers using AVX-256, we can do this:

```
    v1: dd 0.50, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125, 0.00390625
    v2: dd 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0
    v3: dd 0, 0, 0, 0, 0, 0, 0, 0
```

```
main:
    vmovups ymm0, [v1]
    vmovups ymm1, [v2]
    vaddpd  ymm3, ymm1, ymm2
    vmovups [v3], ymm3
```

Note how the form of AVX arithmetic is different from MMX and SSE, with addition now taking three operands rather than two.

Domain-Specific Instructions Using SIMD

As mentioned earlier, SIMD is usually considered a very CISC approach, as it involves adding lots of new instructions to the ISA. Initially, these arise from the many combinations of packing styles, data types, and arithmetic operations. Going beyond simple replication of arithmetic across the chunks, CISC SIMD has also tended to create further complex instructions. These might include *horizontal SIMD*, which means instructions that *combine* information from multiple chunks in the same register. For example, there are instructions that find the minimum of multiple chunks in a register: `phminposuw` in SSE or `vphminposuw` in AVX.

Horizontal SIMD instructions also sometimes sequence simpler SIMD instructions together. For example, "dot product of packed double-precision floating-point values" (`dppd` on SSE; `vdppd` on AVX) is a single instruction that performs a complete vector dot product, often used in games, 3D simulations, and machine learning. This consists of first SIMD multiplying pairs of chunks, then summing the results horizontally along the register.

Cryptography has been a major source of CISC SIMD extensions. For example, 128-bit AES is the NSA-approved standard for internet encryption. It's computed via four steps: ShiftRows, SubBytes, MixColumns, and AddRoundKey. Intel has added CISC instructions for each of these steps, and also a single mega-instruction that combines them all to perform an entire round (`aesenc` on SSE; `vaesenc` on AVX). If, like most end users, you spend the bulk of your computing time streaming videos over HTTPS, then this CISC approach gives a useful targeted speedup for your use case. But Intel's extensions have been controversial, with Linus Torvalds stating that the NSA and Intel have likely back-doored them in the digital logic, and advising Linux programmers not to use them.

Machine learning—specifically, neural network—operations have been the latest target for SIMD CISC, via Intel's Vector Neural Network Instructions (AVX512-VNNI) and Brain Floating Point (AVX512-BF16) extensions to AVX-512, which arrived in Golden Cove and are marketed together as *DL Boost*. For example, "multiply and add unsigned and signed bytes with saturation" (`vpdpbusds`) performs a full neuron's sigmoid-like activation from its inputs and weights in a single instruction. Some researchers have been able to train neural networks faster than on a GPU using these and similar SIMD CISC instructions, so this is now a competition between CPU and GPU architectures.

Compiler Writers and SIMD

The only compiler writers who understand and care about x86 SIMD are the ones working for Intel and AMD, so proprietary CISC compilers are likely to go faster than open source or third-party compilers (such as gcc) for numerical code on CISC architectures.

Intel has released various C libraries, implemented with its own compilers, that convert high-level numerical code into SIMD instructions. These include Integrated Performance Primitives (IPP), the Math Kernel Library (MKL), and the IPEX PyTorch-to-AVX compiler for neural networks.

Open source compiler writers find it hard to get excited about particular proprietary hardware extensions and CISC, and they generally prefer to spend their valuable, scarce time on more general-purpose work to benefit the wider community, such as generating beautiful RISC code that will be accelerated through methods like pipelining and OOOE.

SIMD ON RISC-V

SIMD instructions are a fundamentally CISCy idea—they add lots of new instructions and digital logic, making the instruction set more complex. However, SIMD extensions have also been proposed for RISC-V, such as P for parallel SIMD instructions and V for vector instructions.

No real-world architecture is purely RISC or CISC nowadays, and there's no law against a primarily RISC-style architecture such as RISC-V adding some CISCy features, especially as RISC-V's extension system makes them completely optional. There have, however, been loud opposing voices in the open source RISC-V community, offended by this potential CISC insertion. Even its founders have published an "SIMD considered harmful" warning.

Good RISC style is rather to make use of extra available silicon to optimize pipelines and OOOE, for example by replicating ALUs, registers, and other components needed to run several branches in parallel. This approach may be made harder by the existence of SIMD instructions, especially the most extreme CISCy, multi-step ones, such as dot products, which do both multiplication and addition. Multicores are generally more acceptable to RISC, and RISC-V has an A extension for atomic memory instructions that provides multicore transactions for them.

SIMD on GPU

SIMD appears on a much larger scale in GPUs. SIMD on CPU gives speedups of 2 to 64 times, based on the number of chunks packed into a word. By contrast, a GPU can scale to thousands of identical instructions running simultaneously across the data.

In Chapter 13, we saw how graphics cards evolved, from providing hardware implementations of graphics commands, to providing their own parallel machine code for non-graphical computing. Initially, this was very hard, geeky work, involving encoding big computational algorithms into shaders

as if they were graphics computations, exploiting the highly parallel 3D rendering hardware, then decoding the resulting images to obtain the computational output.

GPU manufacturers quickly noticed this as a new market and redesigned their shader languages into general-purpose GPU instruction sets for general-purpose SIMD computing. These can be used to implement graphics shaders as before, but now also to implement general, non-graphical SIMD computations. This evolution has occurred rapidly to form GPUs that aren't built for graphics at all, but rather for general higher-power scientific and machine learning computation, especially neural networks. This is why "graphics processing unit" is now a misnomer; a modern GPU is really more of a "general parallel unit."

GPU Architecture

It used to be difficult to discuss GPU architectures in general because they were each developed by different companies according to different, secret designs. However, several of these manufacturers got together to agree on *Khronos* standards, which define ways of thinking about GPU hardware architecture at a level of abstraction common to most of them. This enables most GPUs, and also some other devices, to be viewed as if their hardware was implemented as the standard architecture, so the programmer doesn't have to care about their individual details so much. Programmers can also easily swap one GPU for another, including between manufacturers, as long as the new manufacturer provides software tools to convert programs from Khronos standards to their more specific machine codes.

Khronos defines a hierarchy of named entities. We have a single *host* (the computer), which may have multiple *compute devices* inside (the physical GPU cards or chips). There are multiple *compute units (CUs)* in these, and they each contain *processing elements (PEs)*.

The main structure is the CU, whose PEs contain their own registers and ALUs, but share a single program counter, instruction register, and control unit. This creates the SIMD, as each processing element within the CU executes the same instruction from the PC in parallel, but on its own data from its own registers. A CU may also contain other structures such as a cache and some shared memory, allowing the PEs to communicate with one another. Compute devices typically package several independent CUs together. SIMD exists only within single CUs.

NOTE *Khronos standards are designed to be generalizable, not just to many different types of GPU but also to any other SIMD-implementing technologies. For example, they could also be implemented on an FPGA or on an SIMD CPU in some cases. This is why the generic name "compute device" is used in place of "GPU."*

The die shot in Figure 15-5 shows what GPU silicon actually looks like. It shows that the die is arranged much more regularly than the layout of a CPU, with square CUs split evenly throughout and a general cache in the middle.

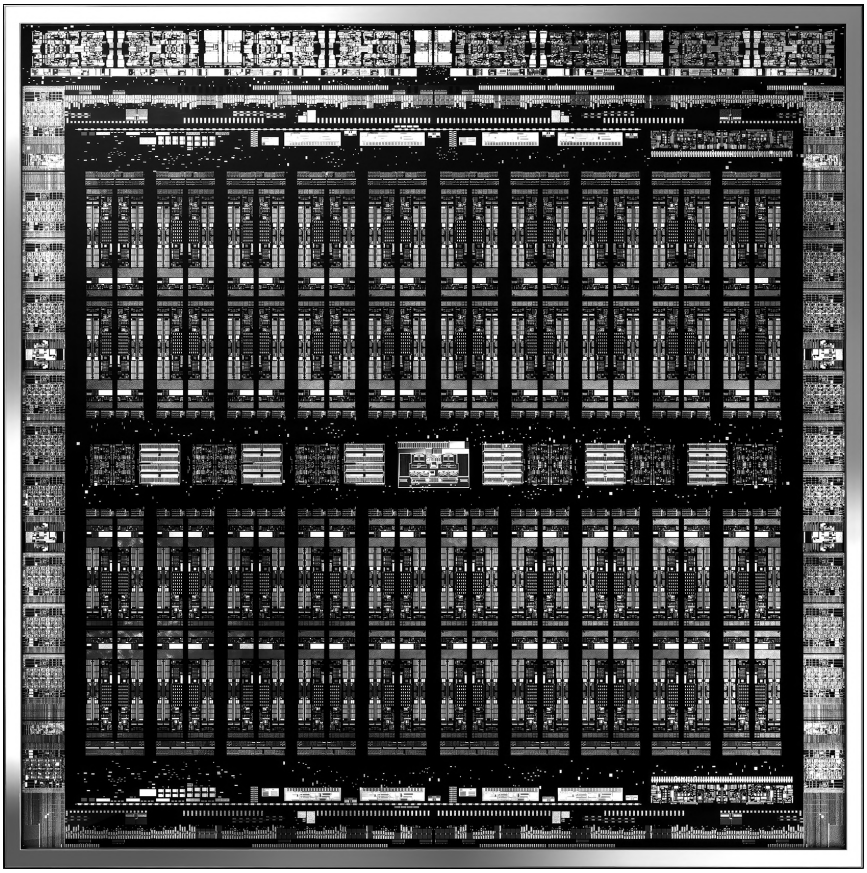


Figure 15-5: A die shot taken from an Nvidia Pascal GPU chip

Nvidia GPU Assembly Programming

In CPUs, SIMD is expressed by single instructions that perform a fixed number (for example, four or eight) of identical operations in parallel. Programs are written as a series of such instructions, with one instruction referenced from the program counter executing at a time. In GPUs, however, SIMD is usually expressed differently: we want to enable large and arbitrary numbers of copies of the instruction to run in parallel, rather than a fixed number.

Khronos defines software-level concepts to express GPU SIMD programs. A *kernel* is a usually small function written by the user programmer, with the intent of each (assembled) line of the code running as a single instruction on multiple data. A *work-item* is one instance of the kernel—that is, the sequence of instructions as applied to a single piece of data by running on one processing element. A *work-group* is the collection of work-item instances running over the multiple data items. Unlike CPU SIMD, kernel code is written by describing its effects on a single work-item. When you run a kernel, you choose and specify how many work-items you want to launch in parallel.

Graphics shaders are traditionally small, simple programs that perform a fixed sequence of operations on each pixel. They're thus well suited to SIMD, with work-items for each pixel stepping through the same instructions in the same order. However, other kinds of compute kernels may require branching. This presents a problem, somewhat in the same spirit as pipeline hazards, where different work-items need to take different branches. Taking different branches destroys the SIMD because the work-items are no longer running the same instructions. There are two methods to deal with kernel branching: masking and subgroups.

Like 1980s CPU designers, modern GPU designers each maintain their own, mutually incompatible ISAs that define their platforms. Nvidia is the most popular GPU designer at the time of writing, so we'll learn to program their ISA as an example of programming GPUs in general. As with other systems we've programmed in this book, we'll simplify the truth a little in order to make learning easier. We'll here assume that all general-purpose Nvidia GPUs implement a single ISA called PTX (Parallel Thread Execution), and we'll learn to program in PTX assembly. You can assemble and run PTX programs on any general-purpose Nvidia GPU.

Data Movement and Arithmetic

The following is a simple PTX kernel program. Like all kernels, many copies forming a work-group are intended to run in SIMD parallel, so this code describes only the actions of a single work-item:

```
mov.u32          %r1, %tid.x;       // r1 := my threadID
cvt.rn.f64.s32   %fd1, %r1;         // convert threadID to float
mul.wide.s32     %rd4, %r1, 8;      // id times 8 = address offset
add.s64          %rd5, %rd3, %rd4;  // global address to store result
st.global.f64    [%rd5], %fd1;      // store threadID in result address
ret;
```

PTX assembly is written with semicolons at line ends, and two slashes for comments. Register names are conventionally written starting with a percent symbol. We'll use four groups of registers: `r` denotes 32-bit integer registers; `rd` denotes 64-bit (double) int registers; `fd` denotes 64-bit (double) floating-point registers; and our fourth group, `tid`, denotes internal registers used to store information about the parallelism.

As usual, most instructions use three operands, with the first being the destination and the others being inputs. As we have several types of register available, most instruction names use Amiga-like suffixes, separated by periods, to indicate which version is being used. For example, `add.s64` means addition for 64-bit signed integers, while `mult.wide.f64` means wide (full) multiplication of 64-bit floats. Load (`ld`) and store (`st`) have suffixes to indicate whether to use global or local memory. The `cvt` instruction means convert, and with various suffixes it converts numbers between signed and unsigned integers and floats of the different bit sizes. As usual, `ret` is return.

The above program, as well as the rest of the PTX programs shown here, assumes at the start that registers rd1 through rd3 contain the global memory addresses of three arrays of doubles, with rd1 and rd2 being inputs, which we'll nickname `x` and `w`, and rd3 being the output, which we'll nickname `out`. (The reason for these conventions will become clear later, when we get to neurons.)

The program's function is very simple. It completely ignores the two inputs. It then obtains its threadID, which is a unique integer assigned to each work-item in the work-group. For example, if we were to launch a work-group of 5,000 copies of the program, each one would be given a unique threadID in the range 0 to 4,999 in its `tid.x` register during the launch. The work-item then writes a copy of its threadID into the corresponding element of the output array. For example, the 573rd work-item, with threadID 573, will write the floating-point number 573.0 into the 573rd element of `out`. If we launch a work-group of 5,000 copies in SIMD, they'll each write a single such number into `out` simultaneously, so that the `out` array then contains the list of numbers from 0 to 4,999 when they complete together.

Although PTX uses 64-bit words (which can be restricted to 32-bit, as seen in the example), it still uses byte addressing. This means that adding 1 to an address moves forward through memory by 8 bits. To move along by a 64-bit word, we have to add 8 to an address. The program thus works by obtaining its threadID, multiplying it by 8, adding the result to the address of `out`, and storing a floating-point version of the threadID at that address. The final `out` array thus contains [0, 1, 2, 3, 4, 5, . . .].

Branching

The definition of SIMD is that parallel copies of the kernel execute identical instructions together as the program executes. Branching runs smoothly in SIMD in GPUs if and only if all of the copies take the same branches, but it becomes complex to handle if they need to take different branches.

For example, the following PTX kernel uses branching:

```
      mov.u32        %r1, %tid.x;    // r1 := my thread ID (an int)
      cvt.rn.f64.s32 %fd4, %r1;      // fd4 := convert threadID to doublefloat

      setp.lt.s32    %p1, %r1, 4;    // set predicate1 to "threadID is less than 4"

      // lines starting @%p1 only execute if predicate1 is true
@%p1  mov.f64        %fd2, 0d4008CCCCCCCCCCCD; // load doublefloat 3.1 to fd2
@%p1  add.f64        %fd1, %fd2, %fd4;         // and add it to the id
      // lines starting @!%p1 only execute if predicate1 is false
@!%p1 mov.f64        %fd3, 0d4024000000000000; // load float 10.0 to fd3
@!%p1 mul.f64        %fd1, %fd3, %fd4;         // and multiply it by thread ID

      mul.wide.s32   %rd4, %r1, 8;     // id times 8 = address offset
      add.s64        %rd5, %rd3, %rd4; // global address to store result
      st.global.f64  [%rd5], %fd1;     // result address := fd1
      ret;
```

The first two and last four lines are the same as the previous program. But after the first two lines, the program tests if the threadID is less than 4. If so, it adds 3.1 to the threadID. If not, it multiplies the threadID by 10.0. Whichever of these results was obtained is then placed in the threadID-th element of `out` as before. On completion, `out` thus contains:

```
3.1, 4.1, 5.1, 6.1, 40.0, 50.0, 60.0. 70.0, 80.0, 90.0 --snip--
```

The complexity here is that we only want a work-item to execute some of the lines if a condition is true or false. In PTX, we first test for the condition (less than, `lt`) and set a *predicate* register to true or false to store the result. Predicate registers are internal registers, usually written as `p1`, `p2`, and so on, which can be set and tested similarly to the status flags we've seen in the Analytical Engine and other systems. Unlike those flags, there are many predicate registers that can each store predicates for long periods without overwriting the previous comparison result. Once we've set a predicate, we can indicate that some lines should be executed only if the predicate is true, or if it is false. These indicators are known as *predicate guards*, and in PTX assembly are written as, for example, `@%p1` at the start of a line. Different GPUs, including different Nvidia models, may handle predicate guards in two ways: masking or subgroups.

Masking is a simple, pure SIMD method suitable for small branches, such as `if..else` statements without jumps. The kernel is executed in SIMD at all times, meaning that all copies share the same program counter and execute the same line of code at the same time. If a line is guarded, the PE tests the predicate, and if the line should not execute, then the PE replaces it with an NOP (no operator), as in a CPU pipeline stall. This enables multiple work-items to remain synchronized, with those that need to execute the instruction doing so while the others wait around for them via these NOPs. This wastes some time, with PEs executing NOPs from one branch and real instructions from the other, but it enables all work-items to stay synchronized as SIMD at all times.

Subgroups (a Khronos term, aka "local groups," "warps," "waves," or "wavefronts" by some manufacturers) are a more heavyweight solution that goes beyond pure SIMD to accommodate conditional jumps. All the work-items in a work-group start out running in pure SIMD until a predicate guard is encountered. When this occurs, the work-group is split into two subgroups, with work-items in one subgroup taking the branch and those in the other not taking it. The subgroups are then treated as two independent SIMD programs and are executed independently, either on two different CUs, if available, or in series on a single CU if only one is available.

Every branch in the program creates an additional subgroup split, so, for example, a program with four branches in a series can lead to $2^4 = 16$ subgroups. At this point, the number of physically available CUs determines the efficiency of execution, rather than the PEs within a CU. This is clearly not sustainable for larger programs with many possible branching series. However, subgroups can be merged back together ("resynchronized") if the programmer can find a way to do so. Typically this can be done when

branching has occurred due to different work-items taking different numbers of loop repetitions. In this case, the programmer can ask the work-items whose loops have completed first to wait until the others have also completed; this is known as a synchronization *barrier* and is represented by a special barrier instruction in assembly and machine code.

The challenges of branching make SIMD quite a restrictive style of programming. It's well suited to graphics shaders, which typically have no or minimal branching, but it's tricky for programs that require parallel threads to take many different branches. Neural networks and physical simulations are two major classes of code that have similar minimal-branching structure to graphics; thus, they've greatly benefited from GPU acceleration. If you need different threads to be doing completely different things from one another, however, then SIMD isn't appropriate. You need MIMD, as seen later in the chapter.

Large Work-Groups

Sometimes we need to run more copies of a kernel than there are available PEs in the CU. For example, a pixel shader needs to run for every one of about eight million pixels for a 4K display, while only thousands or tens of thousands of PEs may be available.

In these cases, a similar subgroup method can be used as in branching: you split the work-group into a number of smaller subgroups such that each subgroup runs physically together in SIMD on the PEs, and multiple subgroups can either run in series on a single CU or simultaneously across several available independent CUs. Unlike in branching, these subgroups are chosen to exactly match the total number of PEs. These maximal-sized subgroups are known as *blocks* by some manufacturers, with the set of subgroups known as a *grid*.

Work-items within every subgroup will be allocated the same set of threadIDs, corresponding to the position of the PE in its CU. It's common to need to convert between these and the global "jobID." For example, if you have eight million pixels to compute and 10,000 PEs, the four-millionth job needs to know that it should write to the four-millionth pixel rather than to its threadID-th pixel, which has a maximum value of 10,0000.

This is a common need, so PTX provides some extra machinery to assist with programming it, as in the following example:

```
mov.u32         %r4, %ctaid.x;      //r4:=which subgroup is this?
mov.u32         %r2, %ntid.x;       //r2:=subgroup size
mov.u32         %r3, %tid.x;        //as well as the usual local thread ID
mad.lo.s32      %r1, %r2, %r4, %r3; //compute the global job ID as
                                    // jobID = r1 := r2 x r4 + r3
cvt.rn.f64.s32  %fd1, %r1;          //fd1 := convert global jobID to float

mul.wide.s32    %rd4, %r1, 8;       //id times 8 = address offset
add.s64         %rd5, %rd3, %rd4;   //global address to store result
```

```
st.global.f64    [%rd5], %fd1;       //store threadID in result address
ret;
```

Here, two additional internal registers, `ntid.x` and `ctaid.x`, are loaded automatically during kernel launches, with the subgroup size and a new ID saying which subgroup is being run. By multiplying and adding these using the dedicated `mad` instruction, we recover the global job ID and proceed as usual. (The rest of the program is the same as the first one, storing a float version of this jobID at the jobID-th location in `out`. The difference is that this now works for much larger `out` arrays—with millions of elements.)

A GPU Neuron

Now let's look at a larger example kernel, which computes a neuron for a convolutional deep neural network (CNN). This is roughly how GPUs are used in machine learning:

```
      mov.u32        %r1, %tid.x;       //r1 := thread ID
      mov.u32        %r2, 0;            //r2 = i = input counter := 0
      mov.f64        %fd1, 0d0000000000000000;  //cumsum:=doublefloat(0)
      mul.wide.s32   %rd4, %r1, 8;
  //id x 8 = addr offset from threadID
MYLOOP:
      mul.wide.s32   %rd5, %r2, 8;      //i times 8
                                        //=adr offset from conv iteration
      add.s64        %rd8, %rd1, %rd4;  //rd8:=adr of id-th element of
                                        // x=&x+jobIDoffset
      add.s64        %rd8, %rd8, %rd5;  // + convoffset
      ld.global.f64  %fd3, [%rd8];      //fd3:=x_(job+i)

      add.s64        %rd9, %rd2, %rd4;  //rd9:=adr of id-th element of
                                        //  w=&w+convoffset
      ld.global.f64 %fd2, [%rd9];       //fd2:=w_i

      mul.f64       %fd4, %fd3, %fd2;   //fd4:=x_(job+i) * w_i
      add.f64       %fd1, %fd1, %fd4;   //cumsum += fd4

      add.u32       %r2, %r2, 1;        //i++
      setp.ne.s32   %p1, %r2, 10;       //test if i==10
@%p1  bra   MYLOOP;                 //stop if so, else loop, using pred guard
     //reLU
      setp.lt.f64   %p0, %fd1, 0d3DA5FD7FE1796495; //pred0:=(cumsum<double 0)
@%p0  mov.f64       %fd1, 0d3DA5FD7FE1796495;      //predicate guard:
                                                   // if p0, cumsum:=0
      add.s64       %rd5, %rd3, %rd4;    //global address to store result
      st.global.f64 [%rd5], %fd1;        //store cumsum in result address
      ret;
```

Here, 0d3DA5FD7FE1796495 is floating-point zero. As in all our examples, we assume at the start that registers rd1 through rd3 contain the global memory addresses of three arrays of doubles, with rd1 and rd2 being inputs which we nickname *x* and *w*; rd3 is the output, which we will nickname *out*. The nicknames *x* and *w* are chosen because the neuron computes:

$$out[id] = reLU\left(\sum_{i=0}^{10} w[i]x[id+i]\right)$$

Here, $reLU(a) = a$ if $a > 0$ and 0 otherwise (*reLU* standing for rectified linear unit). *x* is a 1D signal such as a sound wave, and *w* are weights that are shared by and convolved across the work-group of neurons.

The program is based on a loop that iterates over the terms of the sum in the above equation. During each iteration *i*, it brings w_i and x_i into registers and multiplies them. Each of these $w_i x_i$ terms is then added into a cumulative sum (`cumsum`). Predicate `p1` is used to determine the end of the loop. The *reLU* function is especially easy to implement and fast to run, which is why it's used. We use another predicate, `p0`, to check if `cumsum` > 0. If it is, the *reLU* output in fd1 is set to `cumsum`, and otherwise to zero.

The recent "deep learning revolution" in machine learning owes much more to the ability of GPU SIMD to run models like this at massive scales than it does to any new algorithms.

SASS Dialects

As a manufacturer releases new models, they may modify their ISA, usually by extending it with additional instructions, but often also by breaking backward compatibility with older versions (unlike the x86 tradition of retaining backward compatibility at any cost). For example, Nvidia's ISAs are named after famous scientists, such as Tesla (2006), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016), Volta (2017), Turing (2018), Ampere (2020), Lovelace (2022), and Hopper (2022). They share a core set of similar instructions, but with some variations between them.

Each of these ISAs has its own assembly language dialect, known as a SASS, whose instructions correspond directly to machine code. These assembly languages are each compatible only with their particular architecture, so they change every couple of years. They aren't officially documented, and don't present a stable platform for user programmers to learn. Nvidia developed PTX as a single stable assembly representation, usable by human programmers, that gets translated during assembly to the appropriate SASS dialect.

The following code shows Turing SASS together with corresponding Turing executable machine code, as assembled from the neuron PTX example shown earlier, together with wrapper code to interface it to input and output parameters:

```
0000 MOV R1, c[0][28] ;              00000a0000017a02  003fde0000000f00
0010 MOV R2, 160 ;                   0000016000027802  003fde0000000f00
```

```
0020 LDC.64 R2, c[0][R2] ;            0000000002027b82  00321e0000000a00
0030 MOV R12, R2 ;                    00000002000c7202  003fde0000000f00
0040 MOV R13, R3 ;                    00000003000d7202  003fde0000000f00
0050 MOV R12, R12 ;                   0000000c000c7202  003fde0000000f00
0060 MOV R13, R13 ;                   0000000d000d7202  003fde0000000f00
0070 MOV R2, 168 ;                    0000016800027802  003fde0000000f00
0080 LDC.64 R2, c[0][R2] ;            0000000002027b82  00321e0000000a00
0090 MOV R10, R2 ;                    00000002000a7202  003fde0000000f00
00a0 MOV R11, R3 ;                    00000003000b7202  003fde0000000f00
00b0 MOV R10, R10 ;                   0000000a000a7202  003fde0000000f00
00c0 MOV R11, R11 ;                   0000000b000b7202  003fde0000000f00
00d0 MOV R2, 170 ;                    0000017000027802  003fde0000000f00
00e0 LDC.64 R2, c[0][R2] ;            0000000002027b82  00321e0000000a00
00f0 MOV R8, R2 ;                     0000000200087202  003fde0000000f00
0100 MOV R9, R3 ;                     0000000300097202  003fde0000000f00
0110 MOV R8, R8 ;                     0000000800087202  003fde0000000f00
0120 MOV R9, R9 ;                     0000000900097202  003fde0000000f00
0130 MOV R12, R12 ;                   0000000c000c7202  003fde0000000f00
0140 MOV R13, R13 ;                   0000000d000d7202  003fde0000000f00
0150 MOV R10, R10 ;                   0000000a000a7202  003fde0000000f00
0160 MOV R11, R11 ;                   0000000b000b7202  003fde0000000f00
0170 MOV R8, R8 ;                     0000000800087202  003fde0000000f00
0180 MOV R9, R9 ;                     0000000900097202  003fde0000000f00
0190 S2R R4, SR_TID.X ;               0000000000047919  00321e0000002100
01a0 MOV R4, R4 ;                     0000000400047202  003fde0000000f00
01b0 MOV R0, RZ ;                     000000ff00007202  003fde0000000f00
01c0 CS2R R2, SRZ ;                   0000000000027805  003fde000001ff00
01d0 IMAD.WIDE R4, R4, 8, RZ ;        0000000804047825  003fde00078e02ff
01e0 MOV R14, R4 ;                    00000004000e7202  003fde0000000f00
01f0 MOV R15, R5 ;                    00000005000f7202  003fde0000000f00
0200 MOV R14, R14 ;                   0000000e000e7202  003fde0000000f00
0210 MOV R15, R15 ;                   0000000f000f7202  003fde0000000f00
0220 MOV R12, R12 ;                   0000000c000c7202  003fde0000000f00
0230 MOV R13, R13 ;                   0000000d000d7202  003fde0000000f00
0240 MOV R10, R10 ;                   0000000a000a7202  003fde0000000f00
0250 MOV R11, R11 ;                   0000000b000b7202  003fde0000000f00
0260 MOV R8, R8 ;                     0000000800087202  003fde0000000f00
0270 MOV R9, R9 ;                     0000000900097202  003fde0000000f00
0280 MOV R0, R0 ;                     0000000000007202  003fde0000000f00
0290 MOV R2, R2 ;                     0000000200027202  003fde0000000f00
02a0 MOV R3, R3 ;                     0000000300037202  003fde0000000f00
02b0 IMAD.WIDE R4, R0, 8, RZ ;        0000000800047825  003fde00078e02ff
02c0 MOV R6, R4 ;                     0000000400067202  003fde0000000f00
02d0 MOV R7, R5 ;                     0000000500077202  003fde0000000f00
02e0 IADD3 R4, P0,R12, R14, RZ;       0000000e0c047210  003fde0007f1e0ff
02f0 IADD3.X R5,R13,R15,RZ,P0,!PT;    0000000f0d057210  003fde00007fe4ff
0300 IADD3 R4, P0, R4, R6, RZ ;       0000000604047210  003fde0007f1e0ff
```

```
0310 IADD3.X R5,R5,R7,RZ,P0,!PT;       0000000705057210  003fde00007fe4ff
0320 MOV R4, R4 ;                      0000000400047202  003fde0000000f00
0330 MOV R5, R5 ;                      0000000500057202  003fde0000000f00
0340 MOV R4, R4 ;                      0000000400047202  003fde0000000f00
0350 MOV R5, R5 ;                      0000000500057202  003fde0000000f00
0360 LDG.E.64.SYS R4, [R4] ;           0000000004047381  00321e00001eeb00
0370 IADD3 R6, P0, R10, R14, RZ;       0000000e0a067210  003fde0007f1e0ff
0380 IADD3.X R7,R11,R15,RZ,P0,!PT;     0000000f0b077210  003fde00007fe4ff
0390 MOV R6, R6 ;                      0000000600067202  003fde0000000f00
03a0 MOV R7, R7 ;                      0000000700077202  003fde0000000f00
03b0 MOV R6, R6 ;                      0000000600067202  003fde0000000f00
03c0 MOV R7, R7 ;                      0000000700077202  003fde0000000f00
03d0 LDG.E.64.SYS R6, [R6] ;           0000000006067381  00321e00001eeb00
03e0 DMUL R4, R4, R6 ;                 0000000604047228  00321e0000000000
03f0 DADD R2, R2, R4 ;                 0000000002027229  00321e0000000004
0400 IADD3 R0, R0, 1, RZ ;             0000000100007810  003fde0007ffe0ff
0410 ISETP.NE.AND P0,PT,R0,a,PT;       0000000a0000780c  003fde0003f05270
0420 MOV R2, R2 ;                      0000000200027202  003fde0000000f00
0430 MOV R3, R3 ;                      0000000300037202  003fde0000000f00
0440 MOV R0, R0 ;                      0000000000007202  003fde0000000f00
0450 @P0 BRA 2b0 ;                     fffffe5000000947  003fde000383ffff
0460 DSETP.LT.AND P0,PT,R2,c[2][0],PT; 008000000200762a  00321e0003f01000
0470 MOV R4, e1796495 ;                e179649500047802  003fde0000000f00
0480 MOV R5, 3da5fd7f ;                3da5fd7f00057802  003fde0000000f00
0490 MOV R0, R4 ;                      0000000400007202  003fde0000000f00
04a0 MOV R4, R5 ;                      0000000500047202  003fde0000000f00
04b0 MOV R5, R2 ;                      0000000200057202  003fde0000000f00
04c0 MOV R2, R3 ;                      0000000300027202  003fde0000000f00
04d0 FSEL R0, R0, R5, P0 ;             0000000500007208  003fde0000000000
04e0 FSEL R2, R4, R2, P0 ;             0000000204027208  003fde0000000000
04f0 MOV R3, R2 ;                      0000000200037202  003fde0000000f00
0500 MOV R2, R0 ;                      0000000000027202  003fde0000000f00
0510 IADD3 R4, P0, R8, R14, RZ ;       0000000e08047210  003fde0007f1e0ff
0520 IADD3.X R5,R9,R15,RZ,P0,!PT;      0000000f09057210  003fde00007fe4ff
0530 MOV R4, R4 ;                      0000000400047202  003fde0000000f00
0540 MOV R5, R5 ;                      0000000500057202  003fde0000000f00
0550 MOV R4, R4 ;                      0000000400047202  003fde0000000f00
0560 MOV R5, R5 ;                      0000000500057202  003fde0000000f00
0570 STG.E.64.SYS [R4], R2 ;           0000000204007386  0033de000010eb00
0580 MOV R2, R2 ;                      0000000200027202  003fde0000000f00
0590 MOV R3, R3 ;                      0000000300037202  003fde0000000f00
05a0 EXIT ;                            000000000000794d  003fde0003800000
05b0 BRA 5b0;                          fffffff000007947  000fc0000383ffff
```

The hex seen here is the actual executable code that's transferred over the bus and run on the GPU for the neural network; it's a direct translation of the SASS assembly. (Compare this with the Baby machine code seen in Chapter 7—it's not really so different!)

SASS dialects aren't officially documented, but we—and the internet—can make some guesses as to likely meanings of some instructions based on what we've seen in the corresponding PTX. `MOV` is a move instruction, with operands being either registers or memory locations such as `c[][]` to obtain inputs to the kernel call. `LDG` loads from global memory, and `STG` stores to global memory and is used to return the output of the kernel call. `TID` is the threadID, which tells us which work-item we're running. `IADD` and `FADD` are integer and floating-point addition. `SHL` and `SHR` are shift left and right. `XMAD` is "integer short multiply and add." `BRA` is branch, `NOP` is null operation. `@P0` is a predicate guard, where `P0`'s value is set in the previous line by the `ISETP` instruction. The usual `JMP`, `CALL`, and `RET` are also provided for control flow.

SASS dialects also have dedicated instructions for graphics operations. For example, there's `SUST`, surface store, to actually write to the graphics surface, as well as instructions to load and query textures and barrier sync (`BAR`).

To get the executable code onto the GPU, and then to specify when and how many copies to launch, the host needs a CPU program. For general computation, you need to write this yourself, using tools provided by the GPU manufacturer. For graphics, driver software such as Vulkan will do this work if you tell it where your kernel (known as a shader in this context) is and what type of shading it does (vertex or pixel).

NOTE *Recent GPUs may have many additional features and optimizations, including many CISC-like specialist instructions, and even their own CPU SIMD–style instructions to split up registers into parts and operate on them together. Recent approaches to branching have begun to abandon SIMD altogether and assign separate program counters to PEs, resulting in the machine looking more like the MIMD systems in the following sections than conventional SIMD GPUs.*

Higher-Level GPU Programming

PTX, and occasionally SASS, code is currently written by hand in some cases, where human creativity and knowledge of the underlying architecture can allow for speed optimizations. However, it's more common to use higher-level languages to compile into GPU assemblers in order to achieve portability between different GPUs and to make programming easier.

CUDA is Nvidia's proprietary C-like language that compiles to PTX and then SASS, but not to anything usable by other manufacturers' GPUs. For example, this CUDA program adds two vectors together element-wise:

```
__global__ void myKernel(double *x, double *w, double *out) {
    int id = threadIdx.x;   //get my ID
    out[id] = x[id] + w[id];
}
```

It can be compiled to PTX with Nvidia's `nvcc` compiler:

```
> nvcc -arch=sm_75 -ptx kernel.cu
```

SPIR-V (pronounced "spear vee," as unlike with RISC-V, this *V* is for "Vulkan") is the Khronos standard for representing GPU kernels in an assembly-like language. As PTX generalizes over many Nvidia architectures, SPIR-V is intended to generalize over *all* manufacturers' architectures. Like PTX, it's designed to be converted into assembly languages for each specific architecture. As different architectures may have different numbers of registers, SPIR-V doesn't describe registers at all. Instead, each instruction's result is given a unique ID number, which can be used similarly to a register ID. When someone writes a converter program for a new architecture, they need to think about how to best make use of the available registers to realize the computations described in this way. Intel has also worked on converting SPIR-V to x86 SIMD, enabling its CPUs to compete against GPUs to execute the same code. The following shows SPIR-V code for a roughly equivalent kernel to the vector addition seen in the CUDA example:

```
EntryPoint Kernel 9
MemoryModel Physical64 OpenCL1.2
Name 4 "LocalInvocationId"
Name 9 "add"
Name 10 "in1"
Name 11 "in2"
Name 12 "out"
Name 13 "entry"
Name 15 "call"
Name 16 "arrayidx"
Name 18 "arrayidx1"
Name 20 "add"
Name 21 "arrayidx2"
Decorate 4(LocalInvocationId) Constant
Decorate 4(LocalInvocationId) Built-In LocalInvocationId
Decorate 10(in1) FuncParamAttr 5
Decorate 11(in2) FuncParamAttr 5
Decorate 12(out) FuncParamAttr 5
Decorate 17 Alignment 4
Decorate 19 Alignment 4
Decorate 22 Alignment 4
1: TypeInt 64 0
2: TypeVector 1(int) 3
3: TypePointer UniformConstant 2(ivec3)
5: TypeVoid
6: TypeInt 32 0
7: TypePointer WorkgroupGlobal 6(int)
8: TypeFunction 5 7(ptr) 7(ptr) 7(ptr)
4(LocalInvocationId): 3(ptr) Variable UniformConstant
9(add): 5 Function NoControl 8
10(in1): 7(ptr) FunctionParameter
11(in2): 7(ptr) FunctionParameter
12(out): 7(ptr) FunctionParameter
```

```
13(entry): Label
14: 2(ivec3) Load 4(LocalInvocationId)
15(call): 1(int) CompositeExtract 14 0
16(arrayidx): 7(ptr) InBoundsAccessChain 10(in1) 15(call)
17: 6(int) Load 16(arrayidx)
18(arrayidx1): 7(ptr) InBoundsAccessChain 11(in2) 15(call)
19: 6(int) Load 18(arrayidx1)
20(add): 6(int) IAdd 19 17
21(arrayidx2): 7(ptr) InBoundsAccessChain 12(out) 15(call)
Store 22 21(arrayidx2) 20
Return
FunctionEnd
```

Third-party open source efforts are underway at the time of writing to compile CUDA to SPIR-V, though they aren't supported by Nvidia. Nvidia does, however, accept SPIR-V as input, providing closed tools to compile it to SASS, via PTX and another intermediate language, NVVM.

OpenCL is another Khronos open standard, defining a language similar to Nvidia's CUDA. Open source compilers are available from OpenCL to SPIR-V. The following is an OpenCL kernel roughly equivalent to the CUDA example:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void vecAdd( __global double *a,
__global double *b,
__global double *c,
const unsigned int n) {
  int id = get_global_id(0);
  if (id < n)
    c[id] = a[id] + b[id];
}
```

GLSL is the Khronos standard graphical shader language, which also compiles to SPIR-V. A sample of GLSL implementing Gouraud shading (from *https://www.learnopengles.com/tag/gouraud-shading/*) is shown here:

```
precision mediump float;       // default precision to medium
uniform vec3 u_LightPos;       // the position of the light in eye space
varying vec3 v_Position;       // interpolated position for this fragment
varying vec4 v_Color;          // color interpolated across the triangle
varying vec3 v_Normal;         // interpolated normal for this fragment
void main() {
    float distance = length(u_LightPos - v_Position);
    vec3 lightVector = normalize(u_LightPos - v_Position);
    float diffuse = max(dot(v_Normal, lightVector), 0.1);
    diffuse = diffuse*(1.0/(1.0+(0.25*distance*distance))); //attenuation
    gl_FragColor = v_Color * diffuse;
}
```

Here, `lightVector` is the vector from a light to a vertex, and `diffuse` is the diffuse component given by the dot product of the light vector and vertex normal. If the normal and light vector point in the same direction, then it will get maximum illumination. The color is multiplied by the diffuse illumination level to give the final display color.

Multiple Instruction, Multiple Data

SIMD is like a lot of people acting on the same instruction. *Multiple instruction, multiple data (MIMD)*, on the other hand, is like a lot of people acting on a lot of different instructions. Think of SIMD as a gym class with a trainer shouting out instructions, and the class all moving together in response. MIMD, then, is more like a gym where everyone has their own personal trainer telling them each to do different exercises at the same time. As with SIMD, there are multiple different flavors of MIMD, which we'll explore here.

MIMD on a Single Processor

The simplest MIMD can occur on a single CPU, in architectures called *very long instruction words (VLIW)*. VLIW architectures are related to the vector architectures in SIMD. Vector architectures have multiple data items packed into a single large register, with a single instruction acting on all of the entities packed into that register. In VLIW, each entity in the register has different operations performed on it. For example, instead of adding 1 to everything, we could add 1 to the first number, divide the second by 7, and multiply the last two together, storing them somewhere else.

This may seem counterintuitive, but there are certain combinations of instructions that tend to reappear. For example, when writing a video codec, there are standard complex mathematical operations that you repeat over and over on different data. You can design a single long instruction word to perform this exact specialist sequence of operations. For example, a single VLIW instruction `ADDABCFPMDEFINCGSFTH` might mean "integer add register A to register B, store result in C; floating-point multiply registers D and E, store in F; increment register G; and bit-shift register H"—all in a single instruction! This might, for example, be a standard but intensive part of a video codec computation.

Shared-Memory MIMD

A step above single-CPU MIMD is *shared-memory MIMD*, in which multiple CPUs share an addressed memory space. They can communicate with one another by loading and storing data within this space. If the CPUs are identical, the style of parallelism is known as *symmetric multi-processing (SMP)*. If the CPUs differ, the style of parallelism is known as asymmetric multi-processing (AMP). When the CPUs are located on the same CPU piece of silicon, they're known as *cores*, and the parallelism is called *multicore*.

AMP shared memory goes back to the 1980s, when separate co-processor chips were sometimes plugged in alongside the main CPU for extra operations such as floating-point computation. For example, the Sega Megadrive used a Z80 as a second processor to look after sound, freeing up its main 68000.

SMP shared-memory computer designs have also existed all through the history of x86, beginning with mainboards hosting two or more physical 8086 chips sharing the bus and memory.

With shared-memory MIMD, we have to think about how cache levels should be shared. Often the L1 and maybe the L2 cache are stored inside a single CPU and are specific to it, while the L3 and maybe the L2 cache are shared between the CPUs. This makes managing the caches quite complex. Imagine two CPUs are accessing the same address of RAM, caching it independently. The first CPU writes to the cache, changing the value for the address. Remember the different cache write algorithms we looked at in Chapter 10: What is the cache going to do when the CPU tells it to update the location? Will it just update the local cache? Will it send the change straight back to main memory, or wait until the cache line is victimized before doing so? If the second CPU tries to read from the same address in main memory, how can we ensure it will get the newly updated version? We have to be careful when there are multiple CPUs, as they all have the ability to write out to shared memory equally, which requires extra communication between the CPUs so that the values can be updated and the data can remain in sync across all CPUs and their shared cache.

Multicore on x86

Multicore silicon is now the most common type of shared-memory MIMD, and is probably found in your desktop, laptop, and phone. The first dual-core x86 chip was the AMD Athlon X2, made from two Hammer K8 cores on the same silicon. This was soon followed by Intel's dual-core Core 2. Both companies quickly followed with 4-, 8-, and 16-core processors—including extra cores from hyperthreading—and by 2020 were able to produce 64 cores on high-end processors. Figure 15-6 shows a die shot of an eight-core Zen2 chiplet.

Chiplets such as this are a recent innovation that split up a large chip into several smaller pieces of silicon that are placed together in the same plastic package. This is done because chips are now so large and complex that the statistical probability of a manufacturing error occurring somewhere has become significant. Traditionally, whole chips had to be discarded if any error was present. By using chiplets, only the single chiplet with an error needs to be discarded. Multiple copies of the chiplet shown here can be combined together—and with additional I/O chiplets—to place many more CPUs into a single package than would otherwise be reliably possible.

In Figure 15-6, each core has its own L1 and L2 caches, with an L3 cache shared between them. Notice how the subcomponents of the cores have an almost organic quality in their layouts, like growing mold. This is because—unlike the older CPUs we've seen in die shots—they were laid out not by

human designers but by automated routing algorithms, which prioritize efficiency over beauty or human comprehension.

The cores run independently, with the onus on the software to perform MIMD using them. Some new instructions have been added to the x86 ISA to make this programming easier, however, such as Intel's Transactional Synchronization Extensions (TSX).

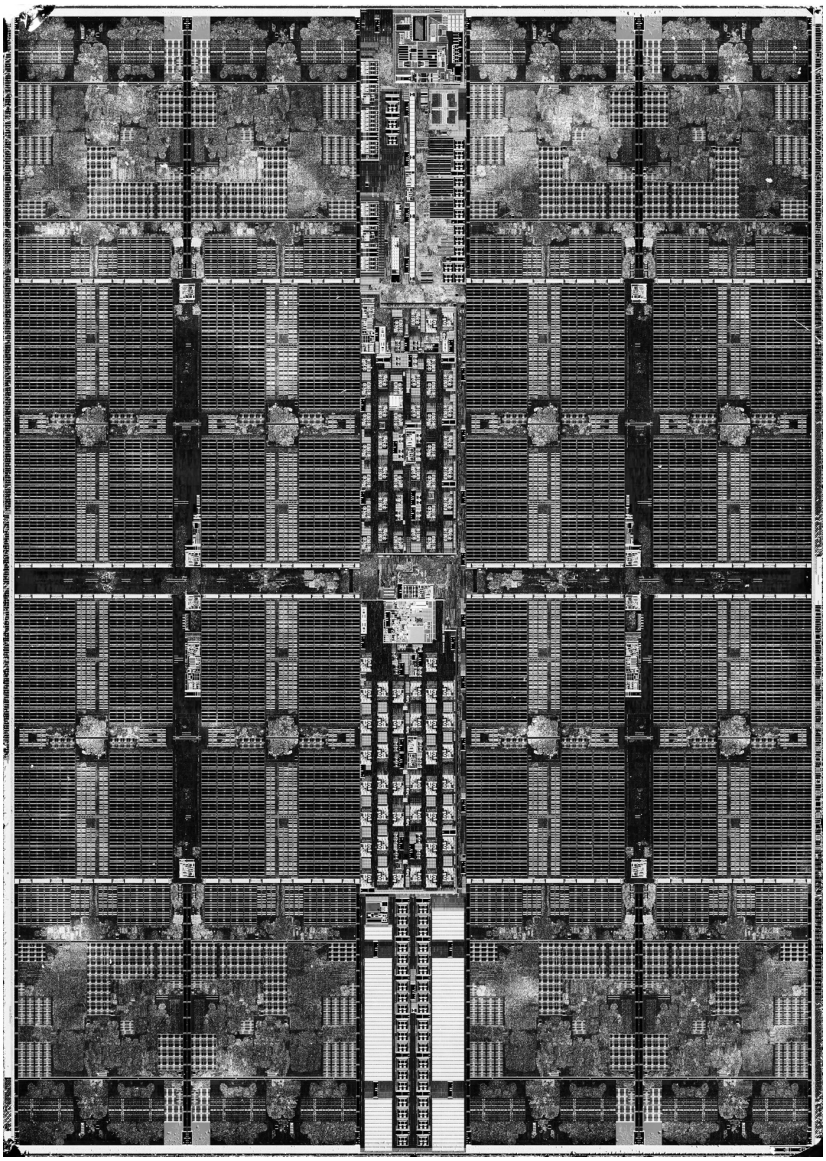


Figure 15-6: A die shot of an AMD Zen2 chiplet, showing eight cores (four rectangles spanning the top, four spanning the bottom) and an L3 cache (eight rectangles in the vertical center)

LOOP VS. MAP PROGRAMMING

Sequential and parallel thinking lead to two different ways of programming multiple copies of work: loops and maps. For example, in the following code we have an array containing four elements that need to be processed. Thinking sequentially, you'd create a loop and do something with each element in sequence:

```
data = [1,2,3,4]
for i in data:
    doSomething(i)
```

The problem with using a loop for this type of work is that it conflates two ideas. First, it expresses that we would like each element of data to be processed. But second, it also specifies the order in which to process them—in this case, starting with the leftmost element and working rightwards one element at a time. The first idea is usually what we actually want to express, and if parallelism is available we don't care about the ordering; rather, we want to allow the machine to order the work in whatever way gets it done most efficiently.

Some programming languages now provide libraries that parallelize regular code across multicores for sections of programs; here's an example from Python:

```
from multiprocessing import Pool
data = [1,2,3,4]
pool(4).map(doSomething , data)
```

This says that we would like a pool of four computations to take place, in any order, such that each computation is performed on one of the elements from the data. Assigning data elements to tasks is called *mapping*, hence the `map` function here.

If you have four cores in your computer, you can run this Python code, and if the system is set up properly, it will know to run on the four cores in parallel to complete the task.

Non-uniform Memory Access

Non-uniform memory access (NUMA) architectures are shared-memory designs in which the speed of access to memory differs according to which part of memory is accessed and by which CPU.

NUMA requires specialist programmers to understand the architecture and manually design programs to take full advantage of it. This includes considering where data is located in memory and trying to group data and processors together so that loads and stores are done on the fastest available parts of memory.

As an example of NUMA, say we have four physical enclosures (cases), each containing several CPUs and RAM. Initially, this may look like four separate computers, but the memory from all four enclosures is connected and mapped together, sharing a single address space. These aren't independent

computers; they are, arguably, a single multicore computer. Unlike with a regular shared-memory machine, however, it takes longer for a CPU to access memory in another enclosure than it would to access memory in its own enclosure.

NOTE *You can address 16 exiwords of memory with 64-bit addressing, which is 16 exibytes if byte addressing is used. This is large enough to cover the entire shared memory of current supercomputers. If we want efficient shared-memory computing to go above this, however, we may need to move to 128-bit architectures.*

NUMA is used in high-performance computing (HPC), where devices are also known as *supercomputers* or "big iron." These are made from many physical, enclosed computers, known as *nodes*, each containing one or more CPUs together with memory, all connected by cables. Unlike regular networking, these *interconnect* connections and the digital logic controlling them are designed to enable direct access to the address spaces of each machine. One possibility for interconnect is to physically extend a single main bus along cables connecting all the machines such that every CPU, RAM, and I/O module share the same bus. Another option is to map all external addresses for a machine to a single I/O module in that machine, which caches all loads and stores to these addresses and arranges for them to take place by communicating with similar I/O modules on the remote machines. Such communications can also include remote DMA (RDMA) to enable large CPU-free bulk transfers between RAM and secondary storage across nodes. Most NUMA architectures include an additional layer of cache, which locally caches the data from remote machines. This is known as *cache-coherent* or cc-NUMA.

The world's most powerful publicly known supercomputer in 2022 was *AMD Frontier*, at the US Department of Energy's Oak Ridge National Laboratory, shown in Figure 15-7.



Figure 15-7: The AMD Frontier supercomputer

Frontier consists of 74 liquid-cooled HPE Cray EX cabinets, each containing eight chassis of eight blades. Each blade has two AMD CPUs and

eight GPUs, giving around 9,400 CPUs and 37,000 GPUs in total. It's capable of performing one quintillion floating-point operations per second, known as an *exaflop*. It has 700 PB of storage, managed using the Lustre filesystem. The secret sauce is the interconnect system, known as HPE Slingshot, which is used with the HyperTransport protocol and over 90 miles of cabling—including direct point-to-point connections between every pair of nodes—to provide NUMA-style memory, making memory on remote nodes appear and act as if it were local. Slingshot uses a similar amount of space, electronics, and power as the compute nodes.

NUMA supercomputers are used for tasks such as weather and climate prediction, physical simulation, and brain modeling, taking advantage of the topographical nature of these domains and linking that to the topographical hierarchy of the NUMA system. *Topography* means the connectivity over physical space; in the context of climate prediction, we're talking about modeling the 3D space of Earth's atmosphere. Each point interacts heavily with adjacent points, with the amount of interaction decreasing with the distance between points. Each point has its own data properties, such as wind velocity, temperature, humidity, and wind pressure. To predict what's going to happen over the next few days or months, we discretize the space into small chunks and give each chunk to a processor to look after, with neighboring chunks of atmosphere given to neighboring processors in the NUMA hierarchy. The processor computes details and makes predictions, factoring in data from other local chunks.

NUMA is also sometimes implemented within a single physical computer enclosure, in high-end workstations and servers. These systems are more likely to run many small, non-interacting programs than a single large scientific program, so specialist programming is less likely to be needed.

MIMD Distributed Computing

Distributed computing means that we have multiple CPUs that each have their own address space and aren't directly accessible to each other. Often, these address spaces are each contained in separate physical boxes, such as server or ATX cases. CPUs in different address spaces can communicate only with one another using I/O. Depending on your definition of a computer, these systems can look a lot like many separate computers, loosely connected by networking I/O. But in other cases, the work they do can be so tightly coupled that it makes more sense to think of them as a single, multicore machine that just happens to have multiple address spaces linked by slower I/O networks, like an extreme form of NUMA.

Servers are computers designed to remain powered on at all times that are often used for distributed computing as well as for providing online services such as websites and databases. Any computer connected to the internet can be used as a server, including desktop PCs and Raspberry Pis, but specialized computer designs have evolved to better meet servers' high-reliability requirements. These include dual power supplies and auto power-on after an outage to reduce downtime due to power grid failures; efficient

heat-flow designs and use of ECC-RAM (as in Chapter 10) to reduce internal failures; 19-inch unit form factors to enable rack mounting; and various forms of physical security to reduce human interference.

Let's take a look at a few forms of distributed computing.

Cluster Computing

There may be a lot of constant communication between the nodes in a cluster. Beowulf is a particular informal standard for building clusters from commodity computers (often many old, recycled desktops). Cluster computing, especially Beowulf, tends to be quite hacky, amateur, and ad-hoc, but can produce powerful systems from low-end machines.

Grid Computing

Grid computing, also known as the *single program, multiple data (SPMD)* style of programming, is where we give the same program but separate data to multiple identical computers. The computers don't run the program's instructions in sync; rather, they can all branch differently, depending on the data, running copies of the same program all at different places in its execution. Grid computing is well suited for applications in data science, speech recognition, data mining, bioinformatics, and media processing. Here you have terabytes or more of information and want the machines to chug away on separate chunks of it at the same time.

A characteristic of this style is that all the machines are exactly the same, and are kept that way by a dedicated technician, who hosts them in a secure environment. Lots of identically high-spec servers are stacked together in racks to guarantee that each instance of your program will run fast and in exactly the same way as the others.

Grid computers don't use shared memory; rather, they're connected by networks via I/O. Network capacity is largely used for the compute nodes to access data on storage nodes hosting hard drives, rather than to communicate with one another. Typically, work is divided into chunks that are sent to compute nodes, which then work independently of one another on their given chunks—this is in contrast to supercomputers, which are all about dense communication between the processors.

Because of the relatively weak connections between nodes, grids are sometimes built from nodes hosted at geographically separate locations. For example, the CERN super-grid links many smaller grids at many universities around the world, enabling them to spread load between them for analyzing big data from millions of particle physics experiments, as needed to find subtle statistical evidence of the Higgs boson.

Decentralized Computing

As we get even looser in our types of parallelism, we get to *decentralized computing*. This is somewhat like multi-site grid computing, but the connection between devices is weaker still. A grid is a stack of the same machines kept running, healthy, and operating identically by a professional IT technician. By contrast, decentralized computing takes a lot of consumer-grade,

non-identical computers, possibly all owned by different people in different countries, and connects them to each other, typically via the public internet. The machines have no shared memory, aren't treated as trusted, and have even less communication across nodes. There's no professional maintaining the machines, nor is there a large setup cost for buying identical components.

Decentralized computing became popular in the 1990s with the famous Search for Extraterrestrial Intelligence (SETI) project. SETI collected big data from large radio telescopes pointed at candidate parts of the night sky, then analyzed it to look for alien communications signals. You could download the SETI program on your desktop, which ran as a screensaver (see Figure 15-8) when your computer was powered on but not otherwise busy.

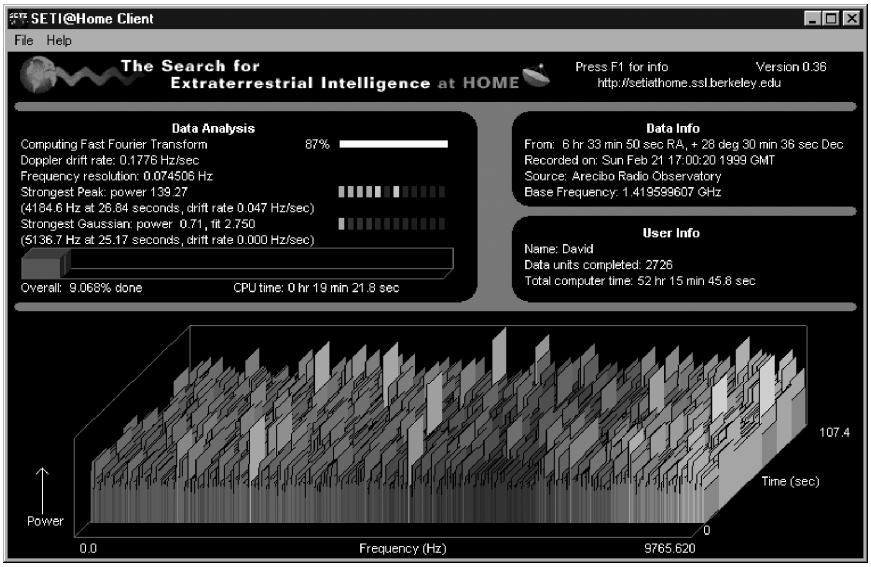


Figure 15-8: The SETI software analyzing radio telescope data for alien communications on a home computer

The program would connect to the main SETI server to register, and be sent one or more chunks of data to analyze. It would return the results to the server, which collected them together with the results from other computers.

HTCondor is modern software that enables arbitrary compute jobs to run decentralized in the background on regular desktop PCs, for example turning unused desktops in an office or classroom into a grid.

Unlike grid computing, the worker machines are no longer under the control of the central manager, so trust and reliability can't be assumed. The manager might send work to workers that don't return a reply or that return fraudulent results. A standard mitigation is to send the same work to three workers and check that they all return the same result—or if two agree, then the third is cheating.

Cloud Computing

The logical evolution of decentralized computing would have been, and might still be, for ordinary computer users around the world to routinely connect together and trade their unused CPU cycles with each other. This way, when you need to run your giant machine learning model, you can run it on one million CPUs all around the world that are otherwise sitting idle apart from displaying screensavers, instead of having to buy your own personal grid. Then, for the other 99.9 percent of the year, you would similarly allow other people to use your own CPU for parts of their large computation when it isn't otherwise being maxed out. Why this still hasn't happened is an interesting social and economic question.

Instead, we've seen—as with other aspects of the internet—a few big companies move in to dominate the market for distributed computing by maintaining their own collections of loosely connected machines, known as *clouds* or *cloud computing*. These remove some of the trust, reliability, and payment issues from open distributed computing, but at the cost of concerns around privacy and loss of control and freedom.

Compute and Storage

A long-standing debate in distributed computing asks whether it's better to store data on the same machines that are doing the computations, or on separate machines.

Separating computation from storage means having two different types of machines in your distributed network: some specialized for storing data and others specialized for computing power. This has the advantage that any available compute node can be used to perform computation on any data. Typically, a software filesystem is used on the storage nodes, which makes them appear and function as if they were a single, very large hard drive. The separation enables the two types of machine to be better specialized for their purposes and to be upgraded independently of one another; it also makes it easy to balance the ratio of storage to computing power. When computation isn't needed, the compute nodes can be switched off to save energy, or made available to other users. When stored data isn't accessed for long periods, it can be relocated down the memory pyramid to tertiary or offline memory, then brought back as needed. The disadvantage of this approach is that it requires lots of network communication to constantly move data around from where it's stored to where it's used, which may become a bottleneck.

Co-location, on the other hand, means having a single type of machine that stores a small part of the data on a local hard drive and also performs computation on it. A big dataset can be split across many such machines, each of which performs the required computations on the data that it hosts locally, with networking used only to transmit the results and to update the data. The advantage here is that network communications are minimized, but the disadvantage is that the computation for a data chunk can be performed only by the single machine that hosts it, making machines easily over- or under-used.

Which approach works best depends on the relative speed and costs of networking, storage, and computing technologies, which change over time, providing much employment for IT consultants swapping between them. Traditional clusters tended to use separation, relying on fast networking such as InfiniBand to move the data around quickly. However, programmers would sometimes switch to co-location, taking local cache copies of data from storage for applications requiring the data to be reread quickly, many times. In the 2000s, co-location became more popular, with the *map-reduce* algorithm used by search engines finding broader applications through the open source software Hadoop and Spark. Map-reduce uses the map replacement for loops discussed earlier, but in a recursive manner, with jobs recursively subdividing their work to pass to other machines, then collating and merging (that is, reducing) their results.

More recently, the move to cloud computing has seen gains in networking speeds in data centers, clearer cost savings from separating storage and computation, and the need to dynamically reallocate users and work to different physical machines, which has all made separation more attractive again. Some systems try to combine elements of both styles, allowing data to be transported over networks to available compute nodes, but preferring the data to be computed on its original node if possible. Any future move from clouds to decentralized computing, which has slower networking than cloud data centers, would likely encourage another swing back to co-location.

Instructionless Parallelism

SIMD and MIMD both extend the classical CPU concept of fetching, decoding, and executing a sequential program of instructions. But there are many ways to use digital logic in parallel that don't involve creating CPUs and programs of instructions at all. Let's look at some of them here.

Dataflow Architectures

Unlike computer scientists, engineers never got hung up on Turing machine serialism in the first place. As they deal with the physical world, engineers tend to view electronic information processing systems, both analog and digital, as physical groups of devices connected together and all operating at all times according to the laws of physics, as in a mechanical machine. For them, such systems have always been parallel and designed using circuit diagrams, such as Figure 15-9, rather than as sequential programs of instructions.

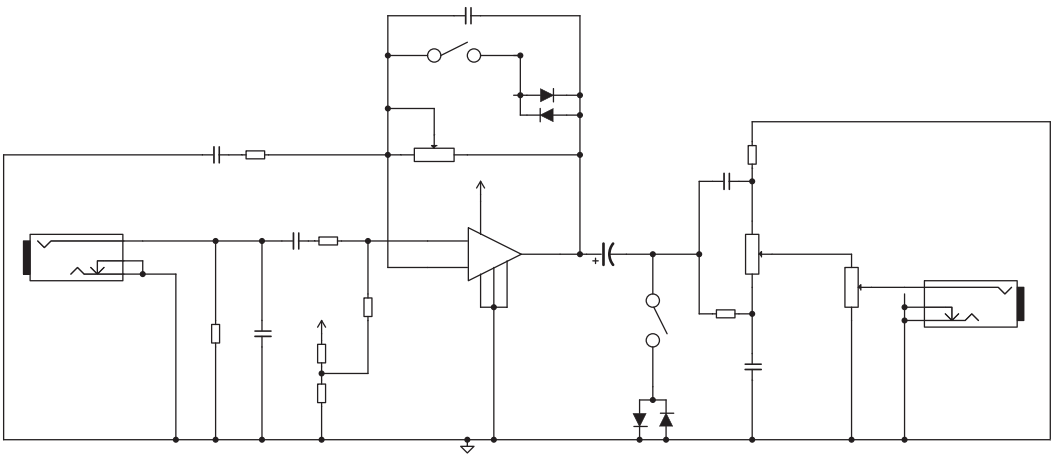


Figure 15-9: A diagram showing parallel information processing in an analog guitar distortion pedal

Rewriting structures like the one in Figure 15-9 as sequential programs would often appear to engineers to be bizarre, inefficient computer science madness. These circuits are composed of hardware components each doing their thing, all at the same time, with data flowing continually around connections between them. Working with digital logic presents a similar view of the world, until we reach the level of Chapter 7, where we choose to use such logic to implement a serial CPU. But digital logic doesn't have to be used just for that purpose. It can also be used in the engineering style of just continuing to design higher- and higher-level parallel machines, running together, with connections between them. Most of the engineers' circuit designs, both analog and digital, can be translated to LogiSim (or Verilog, VHDL, or Chisel) networks of this form. Analog data values can be converted to one of the digital representations we've seen, and analog operations on them converted to digital arithmetic simple machines. As with CPUs, these designs can be burned onto ASIC or FPGA silicon.

This approach can be especially efficient for signal processing computations, in which a pipeline of processing steps is required. For example, a guitar effects unit might require steps of compression, distortion, delay, and reverb. Rather than implement these steps in a sequence, they can all exist together in a pipeline, as is the case when a guitarist chains together several analog hardware pedals implementing one effect each.

Dataflow Compilers

Dataflow languages such as PyTorch, TensorFlow, and Theano, and MATLAB's Simulink, are higher-level languages for specifying parallel information processing. These languages enable the programmer to represent the elements of symbolic mathematical calculations and their dependencies on one another, and then use specialist compilers targeting various types of parallel hardware to order and parallelize them. For example, Figure 15-10 shows a graphical dataflow description of a neural network calculation.

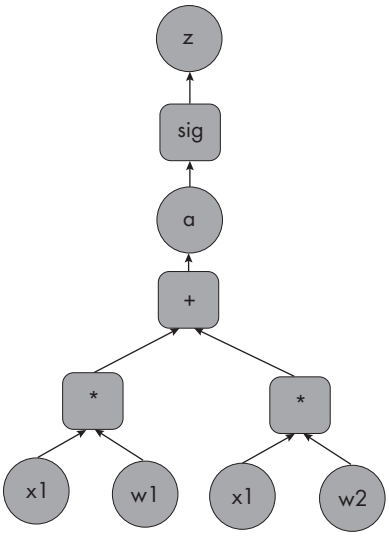


Figure 15-10: The dataflow of a PyTorch neural network calculation

Like hardware description languages, these aren't *programming* languages, but instead are *declarative* languages, more like writing XML or databases than writing traditional imperative programs as sequences of instructions. (SPIR-V can also be considered as a mid-level dataflow language due to its abstraction of registers to identifiers.) Compilers may also exist from dataflow languages to hardware description languages such as Verilog and VHDL, as well as to GPU, CPU SIMD, or serial CPU instructions.

An ongoing research area is how to automatically compile a regular, serial, C-like language into a dataflow language. OOOE is perhaps just the first tip of the iceberg here, optimizing machine code instructions only in small windows of time, but we can imagine a day when entire programs are transformed similarly and automatically into Verilog or perhaps SPIR-V by using advanced parallel algorithms and complexity theory to extract the most parallelized form of the program. Modern compilers can do this for "trivial" cases such as converting loops to maps when iterations of the loops clearly don't affect one another. In general, however, this is difficult work, not least because so much computer science theory is built on serial machines; it may be that big future ideas are needed to rebuild the subject with parallelism as a more fundamental starting point. Functional programming languages may form part of the solution, as they limit the amount of visible state, making it easier to split work into independent and parallelizable pieces.

Hardware Neural Networks

A particular application of dataflow architectures is to enable fast hardware implementations of the backpropagation neural network algorithm. We've known since the 1960s that this algorithm is able to recognize and classify any pattern, given enough data and computing time. We've also known that

it's highly parallelizable, with its neural network being constructed from many "neuron" units that can compute independently and pass messages to their neighbors.

During the 2010s, GPU architectures first enabled these computations to be implemented cheaply in parallel, and were found to enable successful and accurate recognition of complex patterns such as faces in images and words in speech. This created a huge commercial demand for even faster, specialized architectures to implement the backpropagation algorithm even more efficiently than on GPUs.

There are two main approaches to hardware neural networks in current use: FPGAs and NPUs. Let's consider them now.

Backpropagation on FGPAs

Researchers have been building backpropagation neural networks on parallel FPGAs for many decades. FPGA designs may try to physically lay out circuits in terms of component modules for each neuron, or they may just leave the layout to a Chisel or Verilog compiler, which tends to produce random-looking circuits that implement the same logic, sometimes more efficiently. During the 2010s, these systems were built at larger scales for commercial use, especially by "big tech" companies for use in *training* neural networks to make predictions about their big data.

Backpropagation on Neural Processing Units

A recent architecture trend has been the production of similar parallel neural network hardware on ASICs, which run faster than FPGAs. Such chips are known as *neural processor units (NPUs)* or *tensor processor units (TPUs)*.

Some of these are designed as high-power systems for use in *training* neural network models, typically deployed in clusters in racks in computing centers. Others are designed as low-power embedded systems for use in *running* pretrained networks for real-time pattern recognition, and are included in smartphones and IoT devices (for example, Intel Neural Compute Sticks and the Arduino-based Genuino). These units can power applications such as Snapchat's real-time face recognition and filtering. The difference between Moore's law for clock speed and for transistor size has been a major driver of these systems, with phone designers having lots of spare silicon to use up and looking for things to do with it. NPUs were initially "pushed" by manufacturers onto phones, looking for applications, rather than "pulled" by consumer demand.

Summary

We've seen several forms of parallelism in previous chapters, beginning with Babbage's parallel arithmetic instructions and register-level parallelism (ripple-carry adders), then instruction-level parallelism such as pipelining and OOOE. At those levels, the programmer still writes a serial program and doesn't need to know or care that parallelism is making the program run faster.

In contrast, the parallelisms seen in this chapter, SIMD and MIMD, *do* affect the programmer, who needs to understand their details and write programs to best take advantage of them. We looked at architectures in order of the tightness of their parallelism, beginning with systems that are clearly single computers and gradually making the parallel executions more independent until the systems look more like multiple computers connected by networks.

SIMD is where a single instruction is executed multiple times in parallel, on multiple different data items. It can be found in CPUs and GPUs. Typically, user assembly programming for CPU requires thinking in terms of parallel SIMD instructions with fixed, power-of-two parallel copies, while on GPU the ISAs may be structured in terms of instructions for a single thread, allowing more variation in the number of threads launched. The CPU style doesn't easily enable programs with branches, while the GPU style does so via masking or serial split subgroup execution.

MIMD is a looser form of parallelism that can enable different programs to run on different machines. This includes shared-memory systems, in which all processors can load and store in the same address space. These systems can be multicore CPUs located in the same physical box as RAM, or large NUMA supercomputers in which memory in physically further away boxes takes longer to access than nearby memory. Distributed systems are looser still, as each processor or small group of processors has its own address space, and communication between nodes occurs only via network I/O.

The boundary between a single versus multiple computers seems blurry. Most people would consider that a CPU with SIMD instructions is a single computer. It's harder to classify a NUMA supercomputer or a grid system. Decentralized systems such as SETI and Bitcoin combine resources from machines around the world to behave in similar ways to grids. Today, almost every computer has been connected to the internet at some point, where it has communicated with others, perhaps becoming part of a single global computation and computer.

There are still many programmers untrained in parallel algorithms who see them as the exotic stuff of graduate research degrees. The traditional view of parallel programming was that "by the time you've finished writing your fancy parallel thing, Intel will have made a faster processor that makes my serial C code go faster than yours." This doesn't work anymore. Programming now has to be done in parallel because the serial silicon-based architecture has reached its limit. This may require some quite foundational change to computer science as a whole.

Will you as a programmer have to care about parallel programming? There are several possible futures here. In one, you go on writing serial programs as you do now, with clever programmers writing compilers to turn those into parallel systems. Another scenario, happening now, involves a few programmers creating specific libraries to do parallel computing operations, and you calling single functions in your serial program to run each one. A third scenario is that you'll need to write more and more SIMD programs by

yourself, requiring a significant change to your programming style. A fourth is that you'll need to become an MIMD programmer, which is likely a larger style change. Along the way, you might switch your loops to maps, and perhaps from imperative to functional programming. Or perhaps you'll stop programming altogether and, like engineers, just design hardware circuits to perform computations using a declarative language. This is now a big open question, with many programmers placing their career bets by choosing which styles to learn.

Exercises

x86 SIMD

Try running the x86 MMX, SSE, and AVX codes shown in this chapter. You can run them on bare metal using *.iso* files as in Chapter 13. Or, if you have some understanding of operating systems, see the Appendix for how to run them from inside your operating system. This is a faster way to do x86 assembly development.

Nvidia PTX Programming

1. If you have access to an Nvidia GPU—either on your own PC or via a free cloud service with GPU options such as *https://colab.google*—you can compile, edit, and run the chapter's PTX examples. We assumed in the examples that someone or something will be calling the kernels and sending inputs to them. To create that linkage, create a file *mykernel.ptx* with the following code in it:

```
// directives to tell the assembler what versions to use
.version 7.1
.target sm_75
.address_size 64
// this describes how the code will interface with C code on the host
.visible .entry _Z8myKernelPdS_S_(
    .param .u64 _Z8myKernelPdS_S__param_0,
    .param .u64 _Z8myKernelPdS_S__param_1,
    .param .u64 _Z8myKernelPdS_S__param_2
)
{
    // directives to say how many registers we will be using
    .reg .pred  %p<2>;    // predicate reg
    .reg .b32   %r<5>;    // regs of 32-bit ints
    .reg .f64   %fd<5>;   // regs of 64-bit (double) floats
    .reg .b64   %rd<10>;  // regs of 64-bit (double) ints
    // generic part to load argument pointers to rd1-3 and jobID to r1
    ld.param.u64  %rd4, [_Z8myKernelPdS_S__param_0]; //rd1:=pointer arg0
    ld.param.u64  %rd5, [_Z8myKernelPdS_S__param_1];
    ld.param.u64  %rd6, [_Z8myKernelPdS_S__param_2];
    // convert address of pointers, generic to global memory
```

```
    cvta.to.global.u64  %rd1, %rd4;   // rd1 stores global address of x
    cvta.to.global.u64  %rd2, %rd5;   // re2 stores global address of w
    cvta.to.global.u64  %rd3, %rd6;   // rd3 stores global address of out
    //------put your chosen example code below------

}
```

2. Paste in the code from any of the examples below the indicated line to wrap them up. Then assemble to Nvidia executable (cubin) code with:

```
> ptxas -arch=sm_75 --opt-level 0 "mykernel.ptx" -o "mykernel.cubin"
```

   Here, the `-arch` argument is the code for the Nvidia model you're using—for example, `sm_75` is the real name for Turing.

3. If you'd like to inspect the executable as human-readable hex and SASS, this can be done with:

```
> cuobjdump -sass -ptx mykernel.cubin
```

4. You now need some code to run on the host CPU to manage the process of sending this executable to the GPU. You also need to send data inputs for it to run on, as well as commands to launch the desired number of kernels and print out their results. The following code will do all this, and can be used with any kernel that's been wrapped the way we've discussed.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cuda.h"
int main(int argc, char* argv[]) {
    cuInit(0); CUcontext pctx; CUdevice dev;
    cuDeviceGet(&dev, 0);   cuCtxCreate(&pctx, 0, dev);
    CUmodule module;  CUfunction vector_add;
    const char* module_file = "mykernel.cubin";  int err;
    err = cuModuleLoad(&module, module_file); // load cubin executable
    const char* kernel_name = "_Z8myKernelPdS_S_";
    err = cuModuleGetFunction(&vector_add, module, kernel_name);
    int n = 100000;                  // size of vectors
    double *h_x, *h_w, *h_out;       // host in and out vectors
    double *d_x, *d_w, *d_out;       // device in and out vectors
    size_t bytes = n*sizeof(double); // size, in bytes, of each vector
    h_x=(double*)malloc(bytes);      // allocate memory for vectors on host
    h_w=(double*)malloc(bytes);h_out=(double*)malloc(bytes);
    // allocate memory for each vector on GPU
    cudaMalloc(&d_x,bytes);
    cudaMalloc(&d_w, bytes);
```

```
    cudaMalloc(&d_out, bytes);
    int i; for(i = 0; i < n; i++) // init host vecs to arbitrary values
        {h_x[i] = sin(i)*sin(i);  h_w[i] = cos(i)*cos(i);}
    cudaMemcpy(d_x, h_x, bytes, cudaMemcpyHostToDevice); // device<-host
    cudaMemcpy(d_w, h_w, bytes, cudaMemcpyHostToDevice);
    // set arguments and launch the kernels on the GPU
    int blockSize, gridSize;      // threads in block, blocks in grid
    blockSize = 1024;   gridSize = (int)ceil((float)n/blockSize);
    void *args[3] = { &d_x , &d_w, &d_out };
    cuLaunchKernel(vector_add, gridSize,1,1, blockSize,1,1, 0,0,args,0);
    cudaMemcpy(h_out,d_out,bytes,cudaMemcpyDeviceToHost); // host<-result
    for(i=0; i<10; i++) printf("out: %f\n", h_out[i]);    // print result
    cudaFree(d_x);  cudaFree(d_w); cudaFree(d_out);       // free device mem
    free(h_x); free(h_w); free(h_out);  return 0;         // free host mem
}
```

5. Compile and run this with Nvidia's nvcc tool:

```
> nvcc myptxhost.cu -lcuda
> ./a.out
```

   You should see the result printed on the host terminal.

More Challenging

1. If you'd like to try programming in SASS, or even Nvidia machine code, third-party SASS assemblers are available and documented for many of the Nvidia architectures. At *https://github.com/daadaada/turingas* you can find a SASS assembler for Volta, Turing, and Ampere; this site also has SASS assembly code examples and links to similar assemblers for Fermi, Maxwell, and Kepler. Nvidia provides a SASS debugger tool at *https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm*, and a GPU emulator at *https://github.com/gpgpu-sim/gpgpu-sim_distribution*. Nvidia lists the meaning of SASS mnemonics, but not their arguments and semantics, at *https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html*. Some of the third-party SASS assemblers include useful example SASS programs.
2. If you have access to a non-Nvidia GPU, find its make and model and see if there's a public ISA and assembler available for it, similar to PTX or SASS. Assemblers are sometimes created and documented by third-party reverse engineers, even if a GPU manufacturer doesn't make or document one itself. How does it compare to the CPU ISAs you've seen?
3. Simulate a cluster of PCs by running multiple instances of VirtualBox, as used in Chapter 13. Research how to install and run SGE, MPI, or HTCondor across them.

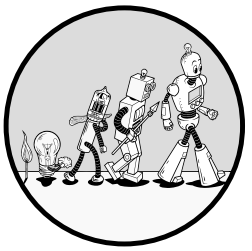
4. If you know neural network theory, add backpropagation to the GPU neuron. Add code to create and run several layers of several neurons each to learn and run some pattern recognition.

Further Reading

- For a reverse engineering of, and third-party open source assembler for, the Nvidia Kepler architecture, see X. Zhang et al., "Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York: Association for Computing Machinery, 2017).
- For a fully open source hardware GPU architecture, see MIAOW, *https://raw.githubusercontent.com/wiki/VerticalResearchGroup/miaow/files/MIAOW_Architecture_Whitepaper.pdf*.
- For more on SPIR-V, see J. Kessenich, "An introduction to SPIR-V," *https://registry.khronos.org/SPIR-V/papers/WhitePaper.pdf*.
- For an example of compiling Python into CPU-less dataflow digital logic, see K. Jurkans and C. Fox, "Python Subset to Digital Logic Dataflow Compiler for Robots and IoT," in *International Symposium on Intelligent and Trustworthy Computing, Communications, and Networking (ITCCN-2023)* (Exeter, UK: IEEE, 2023).

16

FUTURE ARCHITECTURES



Historically, looking at academic research that's close to transitioning to industry tends to accurately predict what will happen over the next decade. At present, research is still mostly done on semiconductor-based technologies, but there are some researchers looking at alternatives. While it's hard to predict much further than a decade ahead, we'll look at some current ideas that might one day go somewhere beyond the present electricity-based computing age. We'll go roughly in order of uncertainty, starting with some close-to-market developments associated with the current "new golden age" of architecture, then traveling through research labs studying optical and DNA architectures, neural architectures, and quantum computing, and finally moving on to speculative ideas based on more distant theories of physics.

The New Golden Age

Architecture is cool again! In the 2010s, trends like the maker, open source, and "mindful design" movements helped drive the resurgence of interest in architecture. Rebelling against the prepackaged black-box interfaces sold to them, artists, innovators, hipsters, and steampunks instead chose to gain greater understanding, control, and satisfaction over the technology in their lives by opening up these boxes and looking at and modifying what's inside. In the professional world, commercial architecture careers over the next decade seem likely to focus on low-cost, low-power embedded and smart systems rather than desktops, laptops, and servers.

The 2010s was also a decade of parallelization and centralized computing, with computation moving off the desktop into the "cloud" of dedicated centralized computing and data centers. It's widely expected that the next step in computer evolution will be the disappearance of desktops and even laptops, replaced by a multitude of small, low-power devices all around the real world that are in constant communication with the cloud, relaying data to the cloud for processing. Smartphones and tablets are early versions of this, but we expect to see even cheaper and smaller devices all over the real world, enabling smart homes, smart farms, and smart cities.

A recent trend identified by Hennessy and Patterson is the demand for custom, domain-specific architectures. In this view, GPUs and NPUs are only the beginning of a new wave of custom silicon designed to accelerate specific, single tasks. It's likely that architects will work on these designs as part of larger teams—for example, working more closely with machine learning engineers and cryptographers to understand and accelerate their algorithms. This would create a cultural shift in computer science, bringing architects back into the mainstream, and requiring everyone else to understand and interact with their work as they did in the 1980s.

Open Source Architectures

For the first time in architectural history, open source thinking has extended into the creation of fully open source hardware and software tooling stacks—RISC-V, BOOM, and Chisel—for professional-quality, state-of-the-art chip design. Along with new affordable FPGAs, these enable anyone to access equipment that was previously only available to a handful of secretive and elite architecture companies. Now almost anyone can be the creator of anything and see and hack the entire stack, from the level of transistors to operating systems. Now is thus the best time to be involved in architecture—even better than the 8-bit days, when hackers could see the ISAs but were still only customers of their chipmakers.

Open source hardware designs have even started to appear for entire consumer PCs, such as the ARM-based Olimex TERES laptop, which users often modify through PCB design software and 3D printing. Open source interest is also being driven by end users, who are feeling increasingly uneasy about the proprietary architecture of individual CPUs that may be backdoored at the digital logic level. For example, Intel has been accused of

hiding and running an entire operating system based on MINIX inside its processors, which can communicate with its Intel home to say potentially anything about what the machine is doing. Open source architectures may become standard and expected—an architectural revolution analogous to the open source software revolution of the 2000s.

While large-scale fabrication is only possible in expensive fab plants, a few companies are large enough to make new masks and fabricate experimental chips on a fairly regular basis. These big companies sometimes now allow researchers and hobbyists to fabricate their own real ASICs for free or low cost by including their designs in an otherwise unused corner of their masks and wafers (for example, *https://developers.google.com/silicon*). There's also been recent progress allowing makers to fabricate their own simpler chips in their garages using open source hardware methods. Sam Zeloof pioneered this approach and in 2021 was able to place and connect 1,200 transistors on a chip—about half the number used in the Intel 4004.

Openness is also becoming an issue in the cloud. There are currently significant concerns around moving from desktop computing—where everyone owns their own computer—to the 2020s cloud, where the computers are owned by a small number of large, powerful companies. This has raised some questions: who will control these computers and the data on them, and how can users be sure that their computations and data aren't being spied on or resold by these companies or other actors?

These concerns might drive new architecture trends. The *open cloud* concept calls for replacing corporate clouds hosted in dedicated computing centers with a shared, loose, decentralized, federated network of ordinary citizens' machines, in their homes. Everyone will have a small, always-on server in their home, a cross between a high-end router, NAS drive, and Intel NUC. These servers will enable non-technical home internet users to easily host their own websites and media streams. They'll also enable fully open source search engines (YaCy), social media (Mastodon), video storage and streaming (PeerTube), video conferences (Matrix), and physical goods marketplaces (OpenBazaar) to replace big tech equivalents by distributing their computations and using cryptographic methods and currencies to ensure trust. The FreedomBox website already has a working software distribution that you can run today on your Raspberry Pi to do some of this. New architectures may be needed to optimize for these use cases.

While hackers and makers can now get their hands on these nice tools, big companies with big resources aren't standing still. They continue to develop smaller and more advanced systems to try to stay ahead, as we'll see next.

Atomic-Scale Transistors

We saw in Chapter 4 that Moore's law for clock speed is over, but Moore's law for silicon transistor density is still holding up. The density law can't go on forever either, though, because we'll hit a point where a transistor is the same size as an atom, and then it will be impossible to go any smaller with

semiconductors. Quantum effects will also kick in as we approach this point, leading to inherent uncertainties about where things are and what they represent. Moore's law for density suggests this will occur around 2060.

IBM can currently manipulate single atoms into simple shapes. For example, Figure 16-1 shows an electron microscope image of a copper surface in which each dot is a single atom, placed and read with their technology.

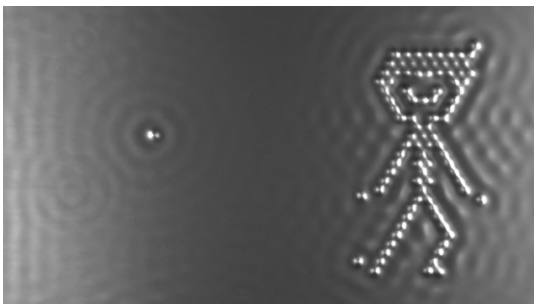


Figure 16-1: IBM manipulates single atoms to create images

The fuzzy, wave-like quality of this image is due to quantum effects. At this scale, it becomes inherently uncertain where the atoms are and how they're moving around. These atoms don't yet function as transistors or computers, but they can, for example, be used for data storage, and IBM would eventually like to develop the technology toward single atom–based computation.

Before we get to this scale, but after conventional semiconductors hit fundamental size limits, nanotechnologies such as carbon nanotubes and graphene might be used to build smaller transistors; this is a current research area. In 2022, researchers at Tsinghua University fabricated a graphene transistor about the size of a single carbon atom, running millions of times faster than silicon.

3D Silicon Architectures

Classical chip layouts were 2D, with a good bit of graph theory and complexity theory needed to optimize the design and minimize the wiring. As we saw in Figure 4-19, current CPUs can be made with a few layers of *overlapping* copper wires, whose 3D structure greatly reduces the wiring. Modern chips still place transistors in a single layer, on the base of the chip, but allow several (typically 2 to 10) layers of wires to be formed on top, insulated from one another by filler materials.

It's possible that today's basic layering technologies will grow incrementally to add more and more layers of wires and transistors, culminating in a move from 2D silicon chips to fully 3D silicon cubes.

However, silicon cubes will create issues around power supplies and heat, requiring something analogous to the brain's blood supply system mixed around the computing elements to get energy in and heat out of the dense 3D structure. We don't currently know how this should be done. The

chip design community has, for living memory, been so focused on 2D layout concerns that it's not clear how it could move to thinking in 3D.

RAM usually has lower usage and heat requirements than processing because most of the time it just sits there doing nothing in a serial computer. Therefore, it's easier to make 3D RAM than it is to make 3D CPUs. There have been recent commercial attempts at 3D RAM, such as Micron's Hybrid Memory Cube.

One source of inspiration for 3D CPU design might come from today's *Minecraft* gaming community. *Minecraft* can act as a Church-powerful computer, using its redstone elements as switches. Fans have already constructed several functioning CPU components inside it, looking similar to Figure 4-19, and even whole CPUs such as "ANDROSII." Unlike previous generations, these players have grown up with *Minecraft*'s inherent three-dimensionality, so instead of laying out their processors on 2D circuit boards or ICs, they've instinctively evolved inherently 3D architectures to optimize their layouts, completely free from manufacturing constraints and the 2D thinking built into the silicon industry.

10,000-Year Memory

What will happen to your data when you die? Will anyone be able to read your files or view your videos thousands of years into the future? Or even 10 years into the future?

The clay tablets from 4,000 years ago that we saw earlier (Figure 1-5) are still perfectly readable. Paper was an advance over clay tablets in terms of speed and capacity, but it doesn't survive as long. As memory technology has advanced and miniaturized, it's gotten faster and increased its capacity, but at the expense of robustness, both to physical decay and to "bit rot" or other technological incompatibilities. All the tertiary and offline storage options we've seen will decay in 100 years. Commercial data centers keep data "alive" by continually copying it to new physical media. Spinning hard disks break and are replaced; tapes and optical discs decay and are replaced. But this relies on continual attention by human maintainers, employed by a company that continues to exist and doesn't go bankrupt or get bought out by new owners who don't want to continue maintaining it.

Research efforts are currently underway to find longer-term storage options as durable as clay tablets, but at modern data sizes. M-disc is a recent optical disc format, backward compatible with Blu-ray, that is claimed to store 100GB for 1,000 years. In 2018, the Arch Mission Foundation deposited a DVD-sized nickel disk onto the moon's surface, containing a full backup of Wikipedia and other documents deemed useful for rebooting humanity in the event of total data loss on Earth. They claim it will last for at least 10,000 years. Glass laser nanostructuring, as developed at the University of Southampton, may store 350TB in a 1-inch cube of very hard glass, with a 14 billion–year lifetime. It's a similar idea to the 3D markings you see in glass trophies, etched deep inside their structure with lasers.

Lasers might also be used to perform computations, as in optical architectures; we'll turn to these now.

Optical Architectures

We've mostly looked at computers that are based on the flow of electrons. Electrons have mass, so they must travel slower than the speed of light. Energy is also required to give momentum to their mass so that they can move around. Light, on the other hand, has no mass, so it moves faster than electrons, at the speed of light (about 300 million meters per second). As this is the physical speed limit of everything in the universe, since the 1960s researchers have asked whether we can compute with light instead of electrons. Like electrons in electricity, light comes in discrete units called *photons*, and the engineering field that studies how to manipulate them is called *photonics*.

Optical Transistors

The speed of electric current is different from the speed of electrons themselves; current usually flows with each electron only moving a small distance, pushing the next electron forward in the circuit. Electrons moving through wire are in a complex environment with many collisions as they bump around, backward and forward, in random walks. The speed of individual electrons drifting along wire is thus very slow, around 1 meter per hour, while the speed of the current in copper wire can be around 90 percent of the speed of light in a vacuum. Thus, a naive expectation that light will compute much faster than electrons seems overly optimistic; switching over our entire hardware technology for just a 10 percent speedup, from 90 percent to 100 percent of light speed, seems not so useful.

However, optical systems have different advantages: there's higher throughput and lower energy consumption than when using electrons because of the lower noise in light propagation. That's why we already use light for routine high-bandwidth, long-distance networking—that is, fiber optics. Optical computing doesn't seem so far-fetched when you remember that most of your internet and phone traffic is already sent around the world via fiber optics.

The difference between mere information transfer and actual computation is that in computation, data elements need to physically interact with one another via some kind of device analogous to a transistor, which in turn would build up logic gates and the rest of the architectural hierarchy. The key problem for optical computing, however, is that photons don't naturally interact with one another. In physics terms, they're bosons rather than fermions, which means that if two of them "collide" they just go straight through each other instead of bouncing off each other. This is great for optical communication, but not for optical computing.

To make an optical transistor, we thus need some form of electro-optical hybrid technology, in which photons can interact with electrons and vice versa to perform computation. Transferring energy between photons and electrons is slow, however, and uses up energy. Such devices currently exist in large photonics labs, made of lasers and precision equipment on optical tables. These systems fill whole rooms and implement only a few hybrid

optical-electronic transistors. Their scale is reminiscent of early electronic computers of the early 20th century. But like those large electronic computers, research also aims to miniaturize them once the basic principles are worked out, probably via photolithography (chip masking) processes similar to the ones used to make conventional electronics; current plans involve using silicon as the electronics substrate, similar to conventional chips.

Optical Correlators

An *optical correlator* (or *4f system*) is a special case of optical computation that has become very practical in the last few years. Rather than being Church powerful, an optical correlator is used for a single purpose: to implement and speed up a single algorithm, the *discrete Fourier transform (DFT)*. The DFT converts streams of spatial and time-series data, such as in sound and video codecs, into frequency-based representations. It uses this equation:

$$X[k] = DFT(x[t]) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{\frac{-i2\pi kn}{N}}$$

For audio signals, the DFT results correspond roughly to the underlying frequencies that generated the signals. For images and video, they correspond roughly to different textures useful in recognition and compression. This is such a basic operation, and used so heavily, that it's worth optimizing it with dedicated hardware, as is currently done by many CISC and DSP instructions.

A key computational property of the DFT is that it speeds up the common operation of convolution (or filtering). For one-dimensional signals, convolution is defined as:

$$(x * y)[t] = \sum_{i=0}^{N-1} x[t] y[t - i]$$

Here, N is the length of the signal y. Implementing this equation directly results in an $O(N^2)$ algorithm, though the *fast Fourier transform (FFT)* is a faster $O(N \log N)$ algorithm based on a mathematically equivalent rearrangement of the equation. The FFT is the fastest known implementation of DFT for a serial computer; it's been described as "the most important numerical algorithm of our lifetime."

Convolution in the source domain is equivalent to multiplication in the Fourier domain. So rather than convolve two raw signals in the raw domain, it can be faster to Fourier transform them both, multiply these transforms together, and use a final DFT to convert back into the raw domain:

$$(x * y)[t] = DFT(DFT(x[t]) \times DFT(y[t]))$$

When a single ray of laser light goes through a single tiny hole, it's *diffracted* to produce a diffraction pattern of light on the other side. It can be shown that if this light signal is passed through a lens, positioned at its focal length f from the image, then at the same distance f on the other side

of the lens, an image is formed that happens to be the DFT of the original image. This was an unexpected and coincidental property of the mathematics of diffraction and lensing, but once discovered it provided an ultra-fast, $O(1)$ physical device to compute DFTs at the speed of light.

Once we have this Fourier image, X, we can implement convolution in $O(1)$ by multiplying pointwise by the DFT of our filter, Y. We precompute Y offline and manufacture a physical filter—just like the colored filters put in front of theater stage lights to change their properties. For most DSP applications, such as video processing, we'll want to apply the same filter y to many images x in a rapid sequence, so we have to compute Y only once. Passing the light image X through this physical filter has the effect of multiplying it by Y, equivalent to convolution $x * y$ in the raw domain. The DFT is self-inverse, so to obtain the final convolution we pass this image through a second lens, of the same focal length, again positioned at distance f from both its input and output. The final result can then be viewed as an image at distance $4f$ from the original input (hence the name *4f system*). The complete system, illustrated in Figure 16-2, computes the entire convolution for fixed Y in $O(1)$ time, at the speed of light.

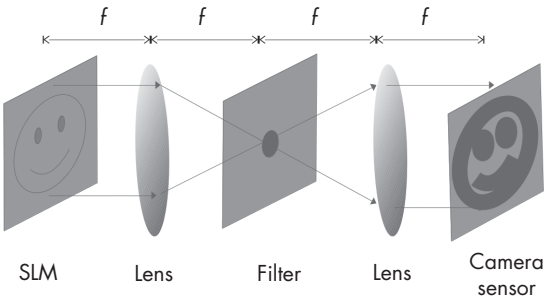


Figure 16-2: A 4f filter structure

This structure has been known since the 1960s but has only recently become practical by piggybacking off well-funded commercial smartphone screen technology. It requires a way to filter laser light through very small but high-resolution images, both to create the initial input image x, and to create changeable filter patterns Y. *Spatial light modulators (SLMs)* are a similar display technology to 4K smartphone displays, originally developed for use in high-end digital overhead projectors. SLMs from these projectors can be taken almost off the shelf and used to create fast, efficient input and filter displays for 4f filters. To complete the setup, an image sensor is also needed to read off the final convolved image. Smartphone digital camera CMOS sensors have been developed, almost symmetrically with display technology, to provide similar resolutions and frame rates required, and can again be used almost off the shelf. Systems built from these components at the time of writing might use 4 megapixels at 15 kHz frame rates.

Optical Neural Networks

Practical optical correlators have become available at the same time that deep learning has revolutionized commercial machine learning. Deep learning has so far consisted of running 1970s neural network algorithms on fast, parallel GPU architectures. In many cases, however, it could be massively accelerated using optical correlators. This is because many problems, especially object recognition in images and video, have a spatially invariant structure, meaning the properties of images don't vary significantly based on which part of the image is being looked at; similar objects are found at all locations around the image. This structure enables *convolutional neural networks (CNNs)* to use the same weights in all nodes within each layer of the network. Mathematically, the effect of this is that each network layer can be viewed as performing a convolution of the layer's inputs with a single weight vector. Computing these convolutions thus becomes the main workhorse operation of these neural networks.

The first practical demonstration of an optical CNN was in 2018, and UK company Optalysys is now commercializing this technology by producing prototypes of consumer optical correlators, as shown in Figure 16-3.

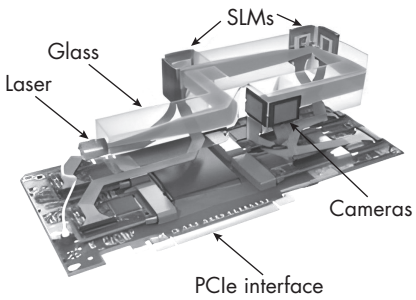


Figure 16-3: An optical correlator PCIe card

This device can now be plugged into a desktop PCIe slot to replace GPUs for deep learning and other applications.

DNA Architectures

Beginning around 2000, labs have investigated *DNA computing* as a way to solve hard computation problems using massive biological parallelism. DNA molecules, as found in living cells, may be viewed (depending on one's conception of representation) as performing computations, and it's been shown that they can encode and efficiently solve computationally NP-hard problems such as the traveling salesperson problem. To understand DNA computing, we'll need a bit of background information.

DNA (deoxyribonucleic acid) is the "source code" for life on Earth. In cellular organisms, every cell contains a complete copy of the whole organism's code (genome) in a set of large double-helix molecules (chromosomes) inside the cell nucleus.

Small parts (genes) of the information encoded in the DNA molecule are copied (transcribed) onto RNA (ribonucleic acid) molecules, which then move out of the nucleus and form construction sites for particular protein molecules to be built. These protein molecules build up the actual body of the organism. This process is known as the *central dogma* of molecular biology: DNA makes RNA; RNA makes proteins.

Each rung of DNA's double-helix ladder is formed from a matching pair of *nucleotides*, small (around 20 atoms) organic molecules of which there are four types: A, T, C, and G. Each has one partner to make pairs: A and T go together; C and G go together. Humans have 23 chromosomes containing a total of about 3 gigapairs of nucleotides. DNA thus uses a base 4 data representation with symbols A, T, C, and G, and the source code for the human genome is about 6 gigabits. This is a similar size to an operating system, and like an operating system, the human genome has been distributed on a single CD-ROM.

DNA technology used to be expensive; for example, it took $100 million to sequence the first human genome in 2001. But it has recently rapidly fallen in price, reaching $1,000 in 2015 and $100 in 2023. This decline in price means the time is ripe to consider DNA as a medium for computation.

Synthetic Biology

Rather than using DNA to store source code for making proteins, as in nature, *synthetic biologists* can use DNA to represent, edit, select, and copy arbitrary data. This enables Church computers to be constructed using DNA data representation and processing.

ATCG strings of DNA can be edited via cutting, splicing, and inserting symbols, as in an ASCII text editor. This is done using custom enzymes that promote the desired reactions. A small set of these enzymes is now well known and can be used routinely to perform these operations.

As for the strings themselves, it's now surprisingly easy to produce your own arbitrary sequences of DNA, which can then be used to store and compute in base 4 as a string of ATCG symbols. It can almost now be done at home using a modified consumer inkjet printer, with its usual cyan, magenta, yellow, and black (CMYK) inks replaced by solutions of ATCG molecules. DNA manufacturing can also be performed on industrial-chemistry scales, making huge numbers of identical or related molecules in a liquid the size of a swimming pool. Consider that just a glass of water contains around 10^{24} water molecules, more than all the bits of data in the world.

Information can be read back from physical DNA using electrophoresis, the same technique used for DNA fingerprinting in crime scene investigations. The *polymerase chain reaction (PCR)* also provides a method to select and copy one particular strand of DNA from a large solution of different strands, the equivalent of extracting a substring from a string.

DNA Computing

Computationally, PCR provides a fast search algorithm. If we can make a liquid containing billions of strands, each encoding a different candidate answer to a computational problem, then we can use PCR to quickly pick out and read off the correct answer.

PCR is a chain reaction, meaning it continues to run and to expand its effects exponentially over time. If the mixture contains just a single DNA strand containing the search string, then that strand will be copied, then each of the copies will also be copied, and so on, until almost the whole liquid ends up full of billions of copies of the answer. This means that a sample of the liquid analyzed by electrophoresis will almost certainly show the desired result.

In 1994, Leonard Adelman successfully used DNA computations to solve a seven-city traveling salesperson problem. The traveling salesperson is a classic NP-hard problem that asks for the shortest route someone can take to visit each of N cities and return home, given a matrix of distances between them. Adelman represented the identity of each city with a short DNA string, then represented routes as strings concatenating these identifiers.

As in standard traveling salesperson formulations, the shortest route question was reformulated as an $O(n)$ series of Boolean questions of the form "Does there exist a route with length less than n?" This question, along with the distance metrics between the cities, was encoded as a primer, which binds only to DNA strands representing routes with the desired property (having a length less than n). For each n, a chemical solution was prepared consisting of many copies of strands of every possible route, in a human-scale vat. The primer was mixed in, then PCR applied to amplify any successful result. Electrophoresis was used to read off the results. This was able to find shortest routes for $N = 7$ cities.

This doesn't mean that $P = NP$ for DNA computers; in time, this is $O(n)$, but it still requires exponential resources in the number of molecules. It's just that with DNA there are a lot of molecules available. DNA is thus able to solve much larger instances of NP-hard problems than other technologies, but like all technologies, there will exist problem sizes that are still too large due to the nature of NP-hardness.

Current research is trying to move DNA computing architectures out of vats in biology labs and into miniaturized biochemical chips that will operate more like normal silicon computers. DNA computing seems unlikely to replace electronics for day-to-day computing tasks, such as running desktop applications, but it might become useful as co-processing in scientific computing for solving large, hard computational problems.

Neural Architectures

Neuroscience has been an important influence on architecture since at least John von Neumann's *Draft Report on the EDVAC*, which used many neural ideas as direct inspiration. Hardware neural networks have been researched

for many decades, but the 2010s saw them take off spectacularly with the GPUs used for deep learning. In the 2020s, NPUs began to appear on mobile phones and in the cloud for machine learning. Computational neuroscience research continues and may inspire radically different computer architectures, beyond current neural networks used in deep learning. As with all computer architectures, we'll here consider the brain's architecture on multiple levels of hierarchy, from its equivalents of transistors, through neurons (brain cells), up to its equivalent of computer design.

Transistors vs. Ion Channels

Recall that a transistor is a digital switch, about 10 nm in diameter in modern chips. It has an input and output, and if you activate the switch, current flows between them. We've seen that transistors work by balancing several chemical and physical forces, and the switch tips this balance to allow the current to flow. Transistors (and chips in general) are made from semiconductors based around silicon, which makes four chemical bonds with neighboring atoms. Really understanding transistors needs chemistry and quantum mechanics.

The brain analog of the transistor isn't the neuron but the *ion channel*, which is a subcomponent of a neuron, as shown in Figure 16-4.

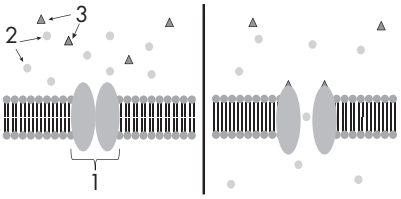


Figure 16-4: An ion channel, in closed (left) and open (right) states. Ligands (3) bind to the channel (1) to open it, allowing ions (2) to flow through it.

Ion channels are single-molecule digital switches, also about 10 nm in diameter, made from proteins and built into the membranes of neurons. Depending on their switching state, they either allow or don't allow certain chemicals to flow between the inside and the outside of the neuron. Their switching states are determined by the balance of electrical and chemical forces, which can be tipped when another chemical binds to the ion channel, or when a voltage is applied to it.

Ion channels (and brains in general) are based around carbon, which makes four chemical bonds with neighboring atoms. As with transistors, really understanding ion channels needs chemistry and quantum mechanics.

Logic Gates vs. Neurons

Neurons (Figure 16-5) are usually considered as the basic unit of computation in brains.

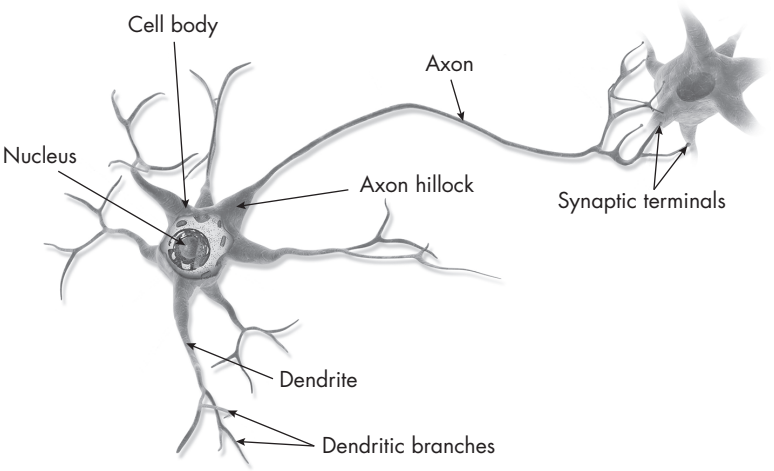


Figure 16-5: A neuron

Neurons are about 1 μm in diameter. Computationally, they're built from many ion channels. They also have many other cellular structures needed to support their existence and power requirements. They function as boxes that take a number of digital inputs and give one digital output, somewhat like the multi-input AND gate seen in Figure 6-2. A multi-input AND gate's function can be written mathematically using a Boolean algebra equation:

$$output = b\left(\sum_{i=0}^{N} input_i\right)$$

Here, b is the Boolean "sum squashing function":

$$b(x) = (x \geq N)$$

Logic gates such as a multi-input AND are clocked, so their inputs and outputs are considered valid only for short periods of time, until the next computation begins.

In simple computational models, as typically used in current machine learning neural networks—and coded as a GPU kernel as in Chapter 15—a single neuron's function is assumed to be given by the equation:

$$output = f\left(\sum_{i=0}^{N} w_i \times input_i\right)$$

Here the ws are adjustable weight values modified during learning, and f is the same squashing function:

$$f(x) = (x \geq N)$$

This notation assumes that one of the inputs is set permanently to 1, rather than containing any actual data. This special input is known as a *bias* or *affinating* input; it's needed to make most neural network models work.

Neurons typically "fire" for short periods of time, so their inputs and outputs are considered valid only for short periods of time until the next computation begins. Unlike logic gates, there is typically a lot of noise in neurons, which can be modeled by adding random numbers to their inputs. Some models consider this noise to be an important probabilistic aspect of their computation.

This is a very simple model of neuron function, and close relations of it work well for current machine learning applications, such as the $f = reLU$ GPU neuron we built in Chapter 15. However, real biological neurons come in hundreds of different shapes and sizes that may have much more complex behaviors, and these have been argued to include more complex computations such as summations, multiplications, divisions, exponentiation, logarithms, temporal memory, and filtering. This school of thought emphasizes the complexity of neurons as whole living and computing cells in themselves, and reminds us of the complex computations performed by other single cells such as bacteria and sponge cells.

Copper Wires vs. Chemical Signals

Let's compare the wiring in the brain with the wiring in a chip. In a chip, we use photolithography to first lay down layers of transistors on a 2D plane. In modern chips, we then lay down a few overlapping layers of copper wire on top of the transistors to make connections between them, as we saw in Figure 4-19. Communication over these wires is very fast and accurate, as it's purely electrical. Messages are digital, meaning the wire is either high or low voltage, which can be viewed as representing 1 or 0.

Neurons are usually long, extended cells, including a long *axon* component that functions as a wire to carry information around the physical brain. Human axons range from 1 μm to 2 m long—the longest are the axons in the neurons connecting your toe to your brain. Communication is slow and noisy, as messages travel along axons via a complex biochemical process involving ion channels opening and closing to move chemicals in and out of the cell. When the end of the axon connects to another neuron (at a joint called a *synapse*), there's a second biochemical process in which chemicals released from the first cell travel into the second one. Messages are digital: the axon is either firing or not firing, which can be viewed as representing 1 or 0. Architecturally, a whole neuron is thus analogous to a logic gate with a single long output wire.

Simple Machines vs. Cortical Columns

The next architectural level is the simple machines level. In human-designed computers, simple machines are made from several logic gates that together perform some single useful function. There are many different standard simple machines, such as adders, decoders, and registers, each specialized

for a particular task. Typical simple machines can be laid out with TTL chips, as we saw in Figure 5-13.

This is the least understood level of brain architecture, and therefore the most exciting topic for scientific research. Some researchers argue that the human cortex consists entirely of repeated *cortical column* microcircuits, each composed of a few hundred or thousand instances of a handful of different types of biological neuron, occupying a cylinder around 20 μm in diameter and 2 mm in depth.

The neurons forming the cortical column microcircuit are arranged across six distinct layers of cortex and always connected in the same specific way, as shown in Figure 16-6. We know the connectivity between the populations of the different types of neurons here, but not the connectivity between individual neurons or the weights of the connections. There's at least some superficial similarity between the structure in Figure 16-6 and the RAM seen in Figure 6-22.

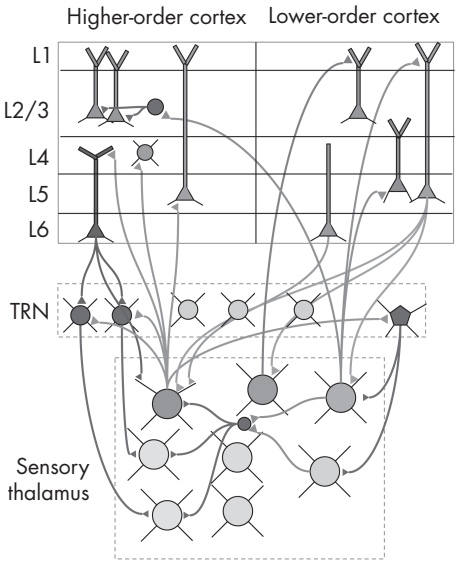


Figure 16-6: The cortical microcircuit architecture

Some computer scientists have speculated that these microcircuits might function as building blocks of probabilistic or other calculations. The precise wiring of the module circuit remains unclear and requires advances in brain imaging technology before we can run a "debugger" on it to learn what it's actually doing. Unlike with digital logic microcircuits, there appears to be only this one cortical microcircuit, which is used all over the cortex. Reverse engineering the cortical microcircuit is one of the biggest science challenges of the 21st century. It needs computer architects and their experience to help suggest computational functions, alongside biological neuroscientists to collect data and link to their biological knowledge, and

physicists to design new experimental equipment able to see this data. Nobel Prizes seem likely for those who crack its code.

Chips vs. Cortex

At the highest level of structure, the cortex is surprisingly similar to chips. This is because they're both laid out in 2D planes, and composed of tens of fairly independent modules with connections between them. For chips, we're used to seeing 2D layouts such as in Figure 11-5. For brains, it's less obvious because the 2D sheet of the cortex is crumpled like a discarded piece of paper into three dimensions. It can easily be uncrumpled, though, and spread out across a 2D surface to show its true structure (Figure 16-7). It's this sheet that contains the six-layer microcircuits discussed earlier.

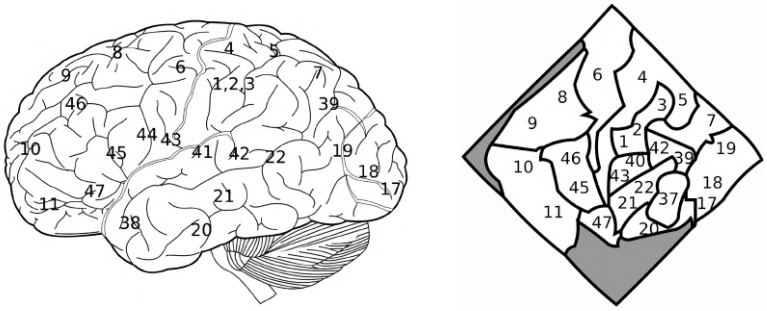


Figure 16-7: The cortex appears three-dimensional (left), but uncrumpled it becomes a 2D sheet, like a paper page or silicon chip (right). The modules, known as Brodmann areas, are labeled with numbers.

The modules of cortex are known as *areas*, and most have been associated with particular functions and activities, such as vision, hearing, touch, and planning. Within each module, connectivity always follows the cortical microcircuit architecture and (arguably) a columnar structure, (arguably) with strong connectivity within each column and weaker connectivity between columns. Most modules have large bundles of axons that send output information to other modules. Projections are always from and to the same layers within these modules, as part of the microcircuit. We know which modules send outputs to which others, but not the detailed connectivity of which neurons connect to which within them.

This is all very like chip architectures, which also often have tens of modular components, each having strong connections within them, and more limited bundles of connections flowing between them. Modern chips have several layers of 3D-printed copper wires to connect components together, sharing the brain's basic plan of a 2D layout with 3D connections linking different areas. A big difference, though, is that all the component areas of the brain share the same internal architecture of layers and columns, while the component areas of chips usually contain completely different designs.

Parallel vs. Serial Computation

Consider how modules are connected together in a CPU or a cortex. CPUs are inherently serial machines, designed to execute programs of instructions in sequence. As such, a CPU has a clearly defined "top" of its design hierarchy, the control unit, that acts as the executive telling all the other modules what to do and when to do it; we saw this in Figure 7-13.

A cortex also has a hierarchy, with frontal areas thought to be involved in executive control and posterior (rear) areas more involved in running perception and action. Perception and action for each sense (vision, touch, and so on) are known to consist of hierarchies of areas; for example, lower visual areas process edges and corners, and higher ones detect faces and named people. These areas all run in parallel, and they're composed of columns that also all run in parallel. The frontal areas seem to coordinate the overall activity, but the perception and action areas can function by themselves when the frontal areas are damaged.

None of these modules are active unless triggered by the *thalamus*, which in this context looks and acts somewhat like a CPU control unit, and can be seen in the lower part of Figure 16-6.

While the modules relay information directly to one another, they also communicate with regions of the thalamus that appear to mirror their structure and act to turn them on and off and resolve conflicts between them.

When you introspect your own subjective experience of computing solutions to complex high-level perception and action-planning problems, it may appear that your brain is operating like a serial machine, imagining and testing out different hypotheses and actions in a sequence. This observation can be supported by the more objective evidence that other humans take *O(N)* time for such tasks when timed in a lab. Internally, however, we also think of the brain as a massively parallel system, with all its neurons potentially in use simultaneously. This is similar to thinking of the CPU first as a serial processor, then thinking of it as a massively parallel digital logic circuit in which all its billions of transistors are potentially in use simultaneously. Outside the brain and CPU, both have external modules, whether connected by a spinal cord or a bus.

The *hippocampus* is a special part of the cortex: it sits right at the top of the hierarchy, and its microcircuit is a little different from the rest. Instead of having cortical layers that process data to send up to higher regions, the hippocampus has different layers, called DG, CA1, and CA3, that include feedback connections, connecting what would usually be outputs back into themselves. Rather than send computations up to more abstract layers of processing, they're sent through *time* to the same, functionally highest, area of the cortex. Computational architectures have been developed based on the hippocampus, on the assumption that it's used as a form of spatiotemporal memory. These architectures enable robots to navigate and map the space and objects around them.

Architects have been intrigued and inspired by the brain throughout the electronic era. Current interest in deep learning has brought some of these links into mainstream architectures, such as the neural processing

units now found in many phones. These architectures are loosely based on models of neurons and on hierarchical cortical areas. But we've seen here that real brains include much additional complexity—ion channels, cortical microcircuits, and emergent serial computation from parallel structures—that may provide inspiration for further developments. It's common for philosophers to debate whether any silicon-based simulation of brain structures could fully replicate human intelligence or consciousness. Those arguing against typically invoke properties of physics that don't usually appear in silicon, such as quantum effects. However, computer scientists have begun to explore computing with some of these effects too, as we'll see in the next sections.

Quantum Architectures

Quantum computing is based on the physics of quantum mechanics, which is famously strange and counterintuitive. In quantum mechanics, objects no longer have precise locations or velocities; rather, they exist in wave-like states that range over many possible locations and velocities. These states define the probabilities of actually seeing the object at one of these locations or velocities when you look at it. Quantum concepts are genuinely mind-blowing, and will radically change your whole view of reality, causation, and time.

A full presentation of quantum mechanics or quantum computing is beyond the scope of this book. Here I can only give a flavor of the concepts and a glimpse at what the basic equations look like. It's worth pointing out, however, that modern quantum computing can be studied with little or no reference to the usual presentation of quantum mechanics given in physics, making the field somewhat easier to approach. In particular, computer science is a mostly discrete subject, dealing with 0s, 1s, and sums, rather than the continuous real numbers and integrals typical of quantum mechanics. The discretized mathematics used in quantum computing requires only high school linear and matrix algebra, complex numbers, and probability.

A Cartoon Version of Quantum Mechanics

The following is *not a correct presentation of quantum mechanics* and is intended only as a cartoon to introduce some key concepts.

Suppose that objects in the world don't just exist in a single state at a time. For example, a cat inside a box may at the same time be both standing up, being alive, and also lying down, being dead. This famous example is known as *the superposed cat*. Suppose that the cat is locked inside the box along with a piece of radioactive material. Radioactive material decays completely at random: its behavior can't be predicted in any way. A radiation decay detector is placed next to it, and connects to a mechanism that releases poison gas into the box, killing the cat if radiation is detected and leaving it alive if not.

You leave this experimental apparatus alone for, say, 10 minutes. You might know something about the strength of the radiation, so you can say that after 10 minutes there's some probability, say 20 percent, that a decay has taken place and the cat is dead, and some other probability, say 80 percent, that it hasn't taken place and the cat is alive. We might represent the current "state" of the cat by a distribution such as:

$$Cat = \{alive : 0.8, dead : 0.2\}$$

Classically—that is, without considering quantum mechanics—you would normally think of this distribution as being a property of your own *knowledge* rather than a property of the *world*. You would assume that the cat is actually in only one state or the other, either alive or dead. It's just that your brain doesn't know which, so *it* (your brain) contains a model carrying the two states and the probabilities.

In quantum mechanics this is absolutely and demonstrably *not* the case. The two versions of the cat aren't only in your head but are also both actually out there in the world in some sense. Roughly, we imagine two versions of reality—one where the cat is alive and one in which it's dead—existing together until the moment you open the box. When that moment comes, reality "decides" which state will be the actual one, randomly but according to the probabilities, and the other state goes away forever. We say that your act of observing the cat changes its state, from existing as two versions with probabilities to existing as a single version.

Now that we have the basic idea, let's take a look at the math version. You aren't expected to understand all of the math symbols, technical terms, or commands used in the rest of this section. If you happen to be familiar with linear algebra and complex numbers, then you can follow the details, but otherwise it's okay just to glance over them to get a flavor of the field.

The Math Version of Quantum Mechanics

The correct presentation of quantum mechanics consists of four rules; superposition, observation, action, and combination.

Rule of Superposition

Objects exist in a *superposition* of states, each with a complex number amplitude whose squared moduli sum to 1. For example:

$$|\phi\rangle = \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2i}{\sqrt{5}} \end{bmatrix}$$

Here, $i = \sqrt{-1}$ and the rows of the vector represent the amplitudes of the cat being dead (binary state 0) and alive (binary state 1), respectively.

Rule of Observation

When you observe states in a basis, they collapse to one of the basis states of the observation basis, at random, according to the moduli of their *squared* amplitudes. For example:

$$P(dead) = |\frac{1}{\sqrt{5}}|^2 = \frac{1}{5}, \quad P(alive) = |\frac{2i}{\sqrt{5}}|^2 = \frac{4}{5}$$

These results are always real numbers between 0 and 1, representing the probabilities of observing each possibility.

Rule of Action

Any physical action, including computation, performed on the system—apart from observation—is modeled by a unitary matrix. The matrix operates on the state by ordinary matrix multiplication. For example, the action of a NOT gate is modeled by a matrix that swaps the dead and live states' amplitudes:

$$NOT|\phi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2i}{\sqrt{5}} \end{bmatrix} = \begin{bmatrix} \frac{2i}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{bmatrix}$$

Here, the NOT matrix, like all unitary matrices, preserves the property of state vectors that the sum of their probabilities, from the rule of observation, is 1.

Rule of Combination

The state of two objects considered together is the joint state formed by the tensor product of the individuals:

$$|\phi_1\phi_2\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$$

The $|\rangle$ notation used in quantum mechanics and quantum computation is called *ket notation*. For discrete computer scientists, this simply denotes a column vector, which in other subjects is sometimes denoted by underlining, using an arrow, or using bold. The name comes from a pun on the word *bracket*. Inner products are sometimes written $\langle \boldsymbol{a}|\boldsymbol{b}\rangle = \boldsymbol{a}^T\boldsymbol{b}$, which is called a "braket." If we write $\langle \boldsymbol{a}| = \boldsymbol{a}^T$ and $|b\rangle = \boldsymbol{b}$, then we can call these two new symbols the "bra" and the "ket," which together make a "braket."

Quantum Registers of Qubits

Could we exploit the apparent existence of interacting parallel realities as a form of parallel computation? If we could distribute computational work across the parallel realities, as we would more normally distribute across multiple CPUs in a single reality, and then somehow find a way to combine the results into the single reality in which we happen to be ourselves, then we could exploit the vast additional computational resources in those other realities. We could build a single CPU and have it compute many things at once across the parallel realities, instead of needing to build many CPUs.

This idea, first proposed by Richard Feynman in 1988, is the beginning of quantum computing.

Consider the superposed cat. We could use the cat's aliveness and deadness as a 1-bit data representation via the encoding *dead* = 0 and *alive* = 1. Call this a *qubit*, for *quantum bit*. We could then build an *N*-bit register by placing *N* of these cats-in-boxes in a row, to store words, as in a classical register based on flip-flops. Until we open the boxes in the register, there are multiple realities in which the cats inside them are alive and dead. When we open them, we see just one version of reality and that becomes the reality we experience ourselves.

Like a classical register, an *N*-qubit quantum register has 2^N possible states. Each of these states of the whole register can exist at the same time in a "parallel world." This is a much larger set of states than just the *N* cats.

You can play with this using a quantum computer simulator such as QCF (Quantum Computing Functions). In QCF, you can begin by creating non-superposed register states, such as:

```
>> phi_1 = bin2vec('011')
[0 0 0 1 0 0 0 0]
```

The resulting output shows a state vector for a 3-bit register that's entirely in the 011 state (representing the decimal number 3). There's zero amplitude of being in the zeroth state 000; zero amplitude of being in the first state, 001; zero amplitude of being in the second state, 010; full amplitude, 1, of being in the third state, 011; and zero amplitude of the other states, up to the seventh, 111.

QCF also has a command to create similar, non-superposed states directly from the decimal numbers being represented; for example, to create a 3-bit register entirely in the state representing decimal 5, use this command:

```
>> phi_2 = dec2vec(5, 3)
[0 0 0 0 0 1 0 0]
```

So far, these are only the same single states that a classical 3-bit register could exist in. Next, we can simulate a register in a superposition of both of these states at the same time, such as:

```
>> psi = [1/sqrt(2)*phi_1 + 1/sqrt(2)*phi_2]
[0 0 0 0.7071 0 0.7071 0 0 ]
```

To simulate a measurement (observation) of this register, we can do this:

```
>> psi = measure(psi)
```

This will randomly produce one of the following two outputs, with probabilities given by the squared amplitudes:

```
[0 0 0 0 0 1 0 0]
[0 0 0 1 0 0 0 0]
```

The individual qubits' states aren't independent; they're *entangled*. In our QCF example, the first two observed bits must read either 01 or 10; they can't read 11 or 00. Therefore, if you look at the first bit initially and see a 0, it means you'll see a 1 if you later look at the second bit, and vice versa. This remains the case even if the qubits are physically transported millions of miles apart before either observation.

The register in our example may be modeled as existing in eight (that is, 2 binary digits ^ 3 bits) states at the same time, across a set of eight "parallel worlds." The number of worlds grows exponentially with register size; for example, a 64-bit quantum register has $2^{64} \approx 2 \times 10^{19}$ states, the same number as the number of addresses in the entire address space of a 64-bit machine, existing all at the same time in a single register.

Physicists usually don't like to talk in terms of "parallel worlds." Instead, they prefer to "shut up and calculate" to predict the outcome of a particular scenario: it's all just math once they've been given a system to analyze. To *create* new quantum programs, however, it's useful for computer scientists to think of each state of the register existing in a parallel world. Thinking in this way helps you to visualize what you're creating and suggests ideas for what to create next.

Computation Across Worlds

The amplitudes, but not the contents, of the states can affect one another in certain ways that are very limited by the rules of quantum mechanics, enabling interaction between the parallel worlds during computation. The billion-dollar question in quantum computing is always: how do we read back the results? We observe only *one* of the parallel worlds, and the one we get is random, so we need to find mechanisms that ensure either that the result we want to see exists in all of the worlds, or that the world we observe happens to be the one with a single copy of the result in it.

For example, we might try to parallelize the traveling salesperson problem by superposing a register so that each world has part of the register encoding a different possible route. Within each world, we then calculate the length of that route and store it in another part of the register. Then we answer a question such as "Does this route have a length less than 5?" and store the result as a single bit in a third part of the register. But our task is to answer a question about the whole set of possible routes, such as "Does any route have length less than 5?" This is a function of the information stored in all of the worlds.

Finding ways to get that information all into one place that we're guaranteed, or even just likely, to see when we make an observation forms the hard part of quantum algorithm design, and as far as we know, these ways all introduce large computational complexity overheads. As a result (again, as far as we know), quantum computers aren't able to make *P = NP*, but they *are* able to speed up *NP* problems to lower complexities within *NP*. Most quantum algorithms, such as *Grover's algorithm*, work by gradually updating state amplitudes so that all worlds that we don't want to see cancel each other out,

and only the world that we do want to see remains with a large probability of appearing in the actual world. This is a similar approach to DNA computing's PCR, which also performs computing over time to amplify the desired solution at the expense of the others. Some researchers believe that quantum computers will provide a general speedup of $O(\sqrt{N})$ via this approach, but theory is still needed to confirm this.

There are a few particular problems, such as breaking public key cryptography, that are known to have larger speedups due to their structures being especially close matches to the quantum laws. Finding and classifying these special cases is a current research topic.

Practical Quantum Architectures

Small-scale quantum computers, having just a few qubits, have been successfully constructed and demonstrated to prove that the concept works. The main barrier to larger practical quantum computers is *decoherence*. This is the problem that *any* interaction between a superposed system and anything in the rest of the world tends to spread out the superposition into that thing and then into the rest of the world. The amount of superposition behaves roughly like a fixed resource, so once it leaks out of your computer it's gone and it can't be used in your computation anymore. Quantum engineers are working hard to design ways to isolate quantum systems from all outside influence. This is a somewhat similar problem to nuclear fusion, in which we set off a nuclear explosion and then try to use magnets to keep it controlled and contained from its surroundings.

Adiabatic quantum computing is sometimes reported in the media—notably by the company D-Wave Systems, which has successfully sold devices to Google and the US government—as successfully performing quantum computing with 1,000 bits or more. However, adiabatic quantum computing isn't quantum computing in the sense we've discussed. It's a different physical process based on a very different mathematical model that assumes time is continuous rather than discrete, so that an infinite number of observations can be made in any given time interval. Completely opposite to quantum computing, it relies on observations (or decoherence, in some views) taking place continually, rather than trying to shield the system from them; the observations form the essential part of the actual computation. Many quantum computing researchers are highly skeptical of these claims, noting that there's a long history of cranks in normal computer science claiming to have made $P = NP$ via models that similarly assume infinite amounts of computation performed in finite time intervals.

Rose's law has been proposed as a quantum version of Moore's law, hypothesizing that the number of qubits in working quantum computers is currently doubling every two years.

Future Physics Architectures

Beyond what's currently called quantum computing, we might more generally turn to modern physics and ask what else it's discovered that might also be made into computing machinery.

Our best current theory of physics, the *Standard Model*, is based on *quantum field theory (QFT)*, which combines quantum mechanics with special (but not general) relativity to model reality as comprising a set of fields that each cover space and interact with one another. Each field corresponds roughly to one type of particle, and as in basic quantum mechanics, its amplitudes are those of finding a particle there if we look for it. Unlike basic quantum mechanics, the fields are also able to represent probabilities of finding multiple particles at locations, and it's possible for these particles to interact and transform into one another in various ways.

The Standard Model specifies particular fields and interactions to make a quantum field theory with 17 types of particle. (More accurately: the fields are a gauge quantum field containing the internal symmetries of the unitary product group $SU(3) \times SU(2) \times U(1)$, and the 17 particle types emerge as patterns across several of these fields.) The Standard Model has been tested experimentally since the 1960s and hasn't changed since then. CERN confirmed the final Higgs field in 2012. A few anomalies are now known that suggest a better model might one day be found.

Particle accelerators such as CERN have perfected the ability to not only observe but also control individual particles of the fields. Beams of different types of particles can be reliably produced, collided with each other or with test objects, and the individual particles flying out of the collision observed.

Particle physics has thus given rise to particle engineering, in which this technology is reused not to do science but to build practical engineered systems for other purposes. Governments have funded particle physics for many decades, not for inherent interest in what the world is made of, but because of weaponization potential. The beams firing around CERN can kill anything in their path. American 1980s BEAR experiments put an accelerator in space, able to produce and fire beams over huge distances, trying to destroy satellites—and eventually ground targets—with laser-like precision. Accelerators and detectors are also being repurposed for treating brain cancer. By firing proton beams through the brain and detecting changes in their speeds, we can infer tumor structures with higher accuracy and less damage than other methods. Once these are known, beam strength can be turned up to destroy the tumors, again more accurately than with other methods.

Now that particle engineering has begun to develop, it's natural to ask if, like mechanical, electrical, and electronic engineering before it, it can be used to construct new computer hardware. It might one day be possible to use particles other than electrons and photons from the Standard Model to store and compute with data—for example, creating a Higgs boson–based computer. Such computers might be constructed by accelerating particles, then using their interactions to form computations, perhaps as microscopic billiard ball logic gates, such as seen in Chapter 5.

QFT isn't a complete theory of physics, because it omits gravity, which is instead modeled by Einstein's *general relativity (GR)*. GR is incompatible with QFT because, unlike QFT, it allows space and time to change shape, bending around mass. We rely on relativistic engineering every day—for example, for time correction among GPS satellites, to correct for warping of telescope images, and to correct paths for missions to Mars and elsewhere. As predicted by Einstein, gravitational waves were observed in 2016, and are now becoming a new tool for astronomy. These effects are small and subtle. In contrast, while engineering systems to actively manipulate and exploit bending space-time is possible in theory, it requires astronomical scales of energy and mass. It may take centuries or millennia, or be impossible, to obtain these. Gödel's "closed timelike curves" can occur in GR if space-time loops around on itself to form a "wormhole" shortcut path between perhaps engineerable points in time and space, including backward time travel.

Observers in GR may see events occur in different temporal orders depending on where they are and how they move. The notion of a sequential *program* becomes problematic if observers can't agree on the order in which instructions are executed, with later stages of execution appearing to cause earlier ones. Time runs at different speeds for different GR observers, so if we live on a large mass, we could make a computer run faster by sending it far away from this mass. However, accelerating and decelerating it for this journey have the opposite effect of slowing its time, which would need to be balanced against any gains.

Hypercomputation theorists have claimed theoretical machines with formal powers stronger than Church computers. They could use GR to predict their own future behavior by looking at their own past in a closed timelike curve, and thus solve the halting problem. This would require a radical update of our concept of computation.

QFT and GR famously don't fit together, so we have no working "Grand Unified Theory" (GUT) to explain the structure of reality. Current attempts include "string theory/M-theory," "loop quantum gravity," and "twistor theory," but none actually work yet. Some of these theories postulate the existence of extra dimensions. Some theories try to model a "graviton" as an additional particle, to treat gravity similarly to the other forces in the Standard Model. This might be of interest for computation, because any gravitons must have zero mass and travel at the speed of light, like photons, but must also be able to interact with each other, unlike photons. This would avoid the non-interaction problem of speed-of-light photonic computers.

Meanwhile, discoveries about galaxy and galaxy supercluster structure and motion are challenging both QFT and relativity. Observations appear to require either the invention of new "dark matter" and "dark energy" particles, such as "axions," or the replacement of relativity with a new theory. If we find that the world is made from superstrings, twistors, gravitons, or axions, then we can also look for ways to use their properties to represent data and perform computation.

Summary

A new golden age of architecture is upon us. There's never been a better time to get involved in architecture, both as a user and as an architect. Open source hardware and software now enables you to design and build serious CPUs at home, and to contribute them to the community.

Taking the long view of computing history, as in Chapter 1, suggests that modern ICs are just one of many possible computing technologies that come and go. The end of Moore's law for clock speed has already forced us to move to parallel architectures, but Moore's law for transistor density must also end as we reach scales of single atoms and quantum effects. This may force us to switch to entirely new technologies.

Optical computing is limited by photons' non-interactivity with one another, though at least in the special case of convolution filters it's possible to make use of interactions within their waves, which are a coincidentally good fit to current deep learning computations.

DNA computing seems unlikely to appear on consumer desktops, but may have a niche for solving large one-off NP-hard problems. Your university or public transportation timetable might one day be optimized by a swimming pool full of DNA.

The human brain continues to inspire new architecture ideas. Going beyond current deep learning architectures, it could lead to ideas for microcircuit-based simple machines and the emergence of serial behavior from massively parallel systems.

Quantum computing is now a well-understood theory, but with research still progressing around its difficult implementation and only a theoretical understanding of what speedups it can provide. Quantum computing is based on quantum mechanics, which has been superseded by QFT and perhaps by attempts at GUTs. Some of these theories are still glints in physicists' eyes, but as with every other technology, from rocks to gears to silicon chips, they may also one day form the basis for future computer architectures.

Exercises

Crank Speedups

You could solve any computation problem in 1 second of wall clock time using an Analytical Engine if you assume that you can turn its crank at arbitrary higher and higher speeds. Why would that not work? What might this tell us about adiabatic quantum computing claims?

Challenging

1. Download QCF from *https://github.com/charles-fox/qcf* and work through the examples shown in the "Quantum Architectures" section on page 414.
2. QCF comes with a longer tutorial that builds up to running Grover's algorithm; work through this.

3. Which technology do you think will yield practical new computers first: quantum, optical, DNA, neural, or other? Write a blog post articulating why.

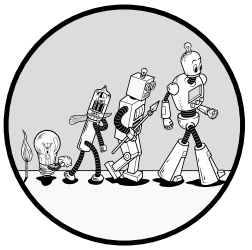
Further Reading

- For DIY fabrication, see Stephen Cass, "The Garage Fab," *IEEE Spectrum* 55, no. 1 (2018): 17–18.
- For graphene transistors, see F. Wu et al., "Vertical MoS2 Transistors with Sub-1-nm Gate Lengths," *Nature* 603 (2022): 259–264.
- An example of 3D integrated circuits is Vasilis Pavlidis, Ioannis Savidis, and Eby Friedman, *Three-Dimensional Integrated Circuit Design*, 2nd ed. (Burlington: Morgan Kaufmann, 2017).
- For details of 10,000-year storage, see J. Zhang et al., "5D Data Storage by Ultrafast Laser Nanostructuring in Glass," paper presented at CLEO: Science and Innovations, San Jose, June 2013.
- For a general introduction to optical computing, see Jürgen Jahns and Sing H. Lee, eds., *Optical Computing Hardware* (Boston: Academic Press, 1994).
- For details of deep learning with optical correlators, see J. Chang et al., "Hybrid Optical-Electronic Convolutional Neural Networks with Optimized Diffractive Optics for Image Classification," *Scientific Reports* 8, no. 12324 (2018).
- For a popular science introduction to DNA computing, see Martyn Amos, *Genesis Machines: The New Science of Biocomputation* (London: Atlantic Books, 2006).
- For details of the traveling salesperson problem with DNA computing, see J. Lee et al., "Solving Traveling Salesman Problems with DNA Molecules Encoding Numerical Values," *BioSystems* 781, no. 3 (2004): 39–47.
- For details of DNA inkjet printing, see T. Goldmann and J. Gonzalez, "DNA-Printing: Utilization of a Standard Inkjet Printer for the Transfer of Nucleic Acids to Solid Supports," *Journal of Biochemical and Biophysical Methods* 42, no. 3 (2000): 105–110.
- The definitive text on quantum computing is Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information* (Cambridge: Cambridge University Press, 2000).
- For the origin of quantum computing, including links to heat, energy, and information issues in computing, see Richard Feynman, *The Feynman Lectures on Computation* (London: Westview Press, 1996).
- For an overview of biological neural architectures, see Larry Swanson, *Brain Architecture: Understanding the Basic Plan* (Oxford: Oxford University Press, 2011).

- For the definitive guide to the many complex computations by single neurons, beyond simple models, see Christof Koch, *Biophysics of Computation* (Oxford: Oxford University Press, 1999).
- For examples of advanced computations performed by single-cell organisms, see R. Lahoz-Beltra, J. Navarro, P. Marijuan, "Bacterial Computing: A Form of Natural Computing and Its Applications," *Frontiers in Microbiology* 5, no. 101 (2014).
- See *https://ai.googleblog.com/2021/06/a-browsable-petascale-reconstruction-of.html* for an interactive 3D view of human cortical microcircuit connectivity.
- For a popular science introduction to future physics, see Brian Greene, *The Elegant Universe: Superstrings, Hidden Dimensions, and the Quest for the Ultimate Theory* (New York: Vintage, 2000).

APPENDIX

OPERATING SYSTEM SUPPORT



We've avoided discussing operating systems in this book in order to see "bare metal" architecture more clearly. Operating systems are a distinct area of study with their own books. It's common to study architecture first, then operating systems. However, demand from operating systems has led to several features being added at the architectural level, and these *do* belong in an architecture book.

This appendix is designed for you to come back to later, during your study of operating systems, as it covers the areas in which the two fields overlap. We'll review some basic features of operating systems, then look at how recent architectures have developed in order to support them at the hardware level.

Concurrency

The most basic function of an operating system is to create the illusion of multiple user programs running simultaneously on a single CPU. The operating system program that does this is usually called the *kernel*. The user programs being run by the kernel are called *processes*.

The kernel runs each process in turn for a short period of time, before switching to the next one; this is called a *cycle*, and this form of execution is called *concurrency*. This means that processes appear to be running in parallel, but are actually being time-sliced, with the slices run in series. Concurrency is roughly the opposite of parallel computing. Parallelism usually takes many CPUs and uses them to execute a single program at the same time. Concurrency takes a single CPU and uses it to execute multiple processes at the same time.

The kernel typically uses architectural timers, IRQ lines, and IRQ callbacks to control switching between processes and kernel code itself. At startup, the kernel sets up a hardware timer that creates a regular IRQ to the CPU. The kernel also has a subroutine, which we'll call a *callback*, that's set up to be called when this IRQ appears. The kernel is given a set of processes to run. It loads all of them into memory, at different locations. It then jumps to the first process's main subroutine, passing control to it to run as normal.

The first process will run for a while, then the timer that was previously set will activate an IRQ. The IRQ hardware detects this, makes a copy of the program counter somewhere (such as in a dedicated internal register), and then sets the program counter to the address of the callback.

The callback is usually programmed to first save a copy of each of the registers and the previously copied program counter in an area of RAM reserved for use by the kernel (that is, not used by any of the processes). It then decides (schedules) which process to run next. The simplest way to do this is for the processes to take turns in a fixed order. The saved register and program counter states for the new process are loaded into the registers and program counter. The updated program counter thus transfers control to the new process until the timer triggers the next IRQ and calls the callback again.

Kernel Mode and User Mode

The kernel will work well as long as the processes can be trusted to play nicely with one another—that is, as long as they access only separate individual areas of memory. It won't work well if processes are malicious. The obvious security problem is that any process could read and write to memory intended for use by the other processes, and by the kernel itself. This could include stealing data, overwriting data, or overwriting code, including overwriting kernel code to take full control of the machine.

Modern CPUs prevent this at the architectural level by providing two (or more) *CPU modes*, called *kernel mode* and *user mode*. In kernel mode, all of the CPU's features are available for the kernel to use. This includes full access to

RAM. In user mode, restrictions are enforced that prevent access to instructions and memory locations outside the region of memory allocated to the user process.

Virtual Memory

A modern operating system doesn't allow user processes to access each other's data, or the kernel's own data. Each user process is presented by the operating system with a *virtual memory* space, which appears to the process as if it were memory in a bare metal machine, isolated from the other processes. For example, all processes might think they're using memory locations 0x00000000 to 0xffffffff. The physical addresses of memory are thus unavailable to the user program, and processes are separated from one another and can't read and write each other's memory. The load and store instructions in user programs work entirely using virtual memory addresses.

Virtual memory can also be made substantially larger than physical RAM by making use of *swap space* with secondary and primary memory. Here, both primary and secondary memories are divided into standard-sized chunks called *pages*. Caching is used to move whole pages between primary and secondary memory according to how recently they were used.

Unlike the hardware CPU and RAM caches we saw previously, this is a slower process that's usually managed at least partly in software by the operating system. A hardware *memory management unit (MMU)* may be added to the architectural level to perform translations between physical and virtual addresses as configured by the operating system.

Different CPU and operating system combinations will use virtual memory in different ways. For example, a key architecture design decision is whether to use physical or virtual addresses in the different CPU-RAM caches.

A *translation lookaside buffer (TLB)* cache is a dedicated cache designed at an architectural level for the operating system to use to implement its virtual memory. It can exist as a third specialist L1 cache along with the instruction and data L1 caches seen in Figure 10-12. When a user program mentions a virtual address, the TLB cache looks up and converts it to a physical address, invisible to the user. If the virtual address is missing from the TLB cache, the TLB then calls back to the operating system code using an IRQ, asking it what to do. The operating system will either find the required virtual-physical mapping and add it to the TLB cache, or it will give an *access violation* error—often known as a *segmentation fault*—if it's not available or allowed. If you've ever run into a segmentation fault in your C code before, it arises here, when you try to access memory that isn't allocated to you.

Device Drivers

A modern operating system also doesn't allow user processes to access I/O addresses directly. Instead, they must call operating system subroutines called *device drivers* to politely request I/O functionality, via the operating

system's API. As with other processes' memory, user mode prevents processes from loading or storing outside their designated address space, and it will raise an exception—such as a segmentation fault—if this is attempted.

I/O modules and device drivers are different concepts. I/O modules are hardware connected to the bus. A device driver is a higher-level concept, a piece of software that takes sole responsibility for all communications with the I/O module or with one (of the many) devices connected using it; it also provides higher-level interfaces (such as C or C++ libraries) that wrap the memory-mapped instructions. In the 8-bit era, these were simple programs located in ROM or loaded into RAM that were accessible to user programs. Today, they're usually implemented as kernel modules that are accessible only to the operating system, and user programs will request their use via the operating system.

ARCHITECTURAL OPERATING SYSTEM SECURITY

Studying the architectural level opens up many interesting opportunities for operating system security. The operating system generally tries to restrict user program access to most parts of the computer, but if you have access to the architectural level, you may be able to circumvent this. What could you do, for example, if you could physically control the IRQ line used by the operating system timer callback, by opening up your computer, attaching a wire to the IRQ pin, and applying voltages to it at times of your choosing?

An ongoing security question is whether device drivers should run in kernel or user mode. Often they're made part of the operating system and given full access to the machine, but this may be dangerous, as it enables any of the writers of the drivers to access your entire machine. This was considered okay in the days when there were just a few reputable printer manufacturers asking to install their own drivers from a CD in the printer box, but it's more worrying now that there are many more international and untrusted hardware manufacturers in operation, not to mention the unbranded websites claiming to host drivers for their products.

Loaders

On an 8-bit machine with no operating system, running an executable file simply requires copying its contents to some location in memory, then setting the program counter of the CPU to point to its first line. This is done by a simple program known as a *loader* stored in ROM. On a modern machine with an operating system, loaders are more complicated: the executable will be running alongside other processes in an area of virtual rather than real memory. A loader thus has to do some work to set this up and alter the executable to use virtual addresses rather than the physical ones the program thinks it's using. On Linux, the loader is invoked with a command such as `./myexecutable`, where the `./` is technically required for security reasons but in practice functions as the loader command.

Let's try writing, loading, and running a "Hello, world!" program from inside an operating system. (We previously did this on the BIOS.) In particular, the following program is able to run inside a window system such as the X Window System and arrange for the text to be displayed in a terminal rather than directly lighting up ASCII patterns of screen pixels. It does this by calling a kernel function—rather than a BIOS function—to request the text display. The operating system's loader assumes there's an externally visible (`global`) label called `_start`, to which it jumps after loading in the code:

```
          global    _start

_start:   mov       rax, 1                ; system call for write
          mov       rdi, 1                ; file handle 1 is stdout
          mov       rsi, message          ; address of string to output
          mov       rdx, 13               ; number of bytes
          syscall                         ; invoke OS to do the write
          mov       rax, 60               ; system call for exit
          xor       rdi, rdi              ; exit code 0
          syscall                         ; invoke OS to exit

message:  db        "Hello, Kernel!", 10    ; note the newline at the end
```

This code runs on 64-bit Linux only. To assemble and run, use this command:

```
> nasm -felf64 hellok.asm && ld -o hellok hellok.o && ./hellok
```

This should write `Hello, Kernel!` to the console using only system calls.

Linkers

When an operating system–hosted executable calls to subroutines in other libraries, virtual memory addresses need to be further relocated. This is to ensure the executable machine code for each library is loaded into memory at a suitable location, meaning one that doesn't conflict with the others. Tweaking these addresses also ensures the libraries can find one another. If one program or library calls a function in another, the address of the target subroutine needs to be changed in its executable machine code to the correct location where the target has actually been loaded. Making these tweaks is called *linking* and is performed by a *linker* program, usually called invisibly by the loader.

As an example of linking, here's another way to write to the terminal, this time by calling the standard C library's `printf` subroutine:

```
global main
extern printf

msg: db "Hello libC!", 0   ; 0 = ASCII endofstring
fmtstr: db "%s", 10, 0     ; ASCII newline and endofstring
```

```
fmtint: db '%10d', 10, 0   ; ASCII newline and endofstring

main:
    mov rdi,fmtstr
    mov rsi,msg     ; pointer to msg
    mov rax,0       ; num of extra stack args used (none)
    call printf     ; call C function

    mov rdi,fmtint
    mov rsi,124     ; 124 is an int to print out
    mov rax,0       ; num of extra stack args used (none)
    call printf     ; call C function
    ret
```

With this style, you can call any C libraries from your assembly programs, as long as you respect their calling conventions. Because it's part of a C compiler stack, the `gcc` compiler looks for an externally visible (`global`) subroutine named `main`, as in C. It will create its own lower-level `_start` subroutine and set it to call `main`; it will also set up any structures needed by the C libraries.

Note that because `printf` can take a variable number of arguments, we have to tell it how many extra arguments are used and should be expected on the stack; we set this number in RAX. This is standard in most x86 calling conventions for variable arguments.

To assemble, link, and run on 64-bit Linux, use this command:

```
> nasm -felf64 helloc.asm  ;  gcc -no-pie -o helloc  helloc.o ; ./helloc
```

You can see what extra code the linker has added by *disassembling*—that is, converting the machine code back to human-readable assembly. You can do this with a tool like `objdump`:

```
> objdump -d helloc
```

Some operating systems make use of x86 segments—or, at least, their assembler directives—to enforce a read-only `.text` section in the code. They typically allow writes in the `.data` section.

Extra Boot Sequence Stages

Most systems can't boot an operating system directly at power on. Operating systems are responsible for loading and configuring device drivers, which aren't available when the operating system still needs to be loaded. Instead, they're gradually brought into being during later stages of the boot process.

We met BIOS and UEFI previously in Chapter 13. Usually only two programs ever get run on your BIOS: an operating system loader and an operating system loader selector program, such as GRUB2 (Grand Unified Bootloader version 2). PCBIOS runs the first such program from a specific

hard disk location called the master boot record. UEFI now has a higher-level view of the filesystem than this, and it includes a specific path on the hard disk to look for and run the first of these programs. GRUB2 provides a text-based user interface, displaying a list of operating systems available on a hard disk and allowing the user to input their selection using their cursor and other keys. GRUB2 checks what kind of BIOS is available, then calls the available subroutines from that BIOS to write the characters on the screen and read the keyboard. When the user makes a selection, it loads that operating system loader and passes control to it.

The operating system loader is thus the first program that's part of the operating system. It will initially rely on the BIOS libraries to access the computer, especially the hard disk that contains the code for the rest of the operating system. An operating system may have its own drivers, hopefully better than those of the BIOS, and it will progressively load and switch over to them. For example, BIOS graphics are by design low resolution so that they work on any monitor, but once the operating system loads it can consider the precise make and model of the monitor and load a new custom driver that can make use of all its features.

Modern boot processes have been controversial for security reasons. The boot process occurs before the operating system kicks in, meaning it has access to the entirety of the computer. UEFI keeps running in the background once the operating system has started, allowing the operating system to call its subroutines. But this means that any malicious code built into UEFI firmware could potentially retain access to the whole machine during regular operating system operation.

UEFI was designed by a committee whose members included proprietary operating system vendors who successfully lobbied for the introduction of a "secure boot" part of its standard. This allows the boot process to be locked down so that buyers of preinstalled machines can't install GRUB2 and other operating systems. It's possible to fix this bug in the standard if you're able to reset the secure boot system itself. This is usually done by soldering two wires to the UEFI chip and applying a voltage to factory-reset it.

Since around 2008, rumors have circulated that Intel motherboards have included an entire additional operating system, based on MINIX3, running somewhere in the boot process between UEFI and the main operating system, as the "Intel Management Engine." If true, these rumors would suggest a major security loophole, as this operating system would have full access to the entire machine, including internet communications and automatic update systems, which would enable Intel or others to push code at any time over the network to run with full read and write access to your computer. These rumors would also suggest that MINIX is now the most widely run operating system in the world—this would be somewhat ironic, as MINIX was created as an educational operating system, with Linux considered to be the more "real world" evolution of it.

Hypervisor Mode, Virtualization, and Containers

Kernel mode is sometimes known as *supervisor mode*, the supervisor being the kernel that controls the switching of the processes being run. *Hypervisor mode* is a related but higher-level concept in which—rather than switching between multiple processes within an operating system—the CPU switches between multiple operating systems running concurrently. This concept has become especially important in current cloud computing, in which the many machines in a computer center are shared in this way to provide each user with the experience of being on a scalable group of machines as their root user.

Similar operating system sharing can also be achieved using software only: there are programs that emulate or simulate virtual machines. However, these incur performance hits, while hypervisors don't. With a hypervisor, each operating system really is running directly on the hardware. Dedicated hypervisor architecture is used to manage the swapping of state in and out of the hardware, in a similar style to how a software supervisor swaps processes in and out of execution. Some virtual machine programs, such as the VirtualBox program used in Chapter 13, can make use of the hypervisor to run their virtual machines on hypervised processors.

Containerization is an alternative to virtualization. Rather than creating a set of completely isolated virtual machines, it works together with additional software to create the *appearance* of many such machines, while having them all actually share a single operating system and other components such as software libraries. (This is arguably what operating systems were intended to do in the first place. But unlike operating systems, containers enable different users to experience different installations and versions of the system, libraries, and installed software.) This is a lighter-weight solution than virtual machines, and it can enable thousands of containers to run together, for different users, on a single computer. Containerization is especially useful for cloud computing, in which thousands of users want to run isolated programs and providers want to minimize costs by having them share a single physical machine.

Real-Time Operating Systems

Most embedded systems run just a single small, simple program, so they have no need for an operating system. However, as the needs of some embedded systems grow in complexity, it's becoming easier and more common to program them as multiple processes. At this stage it can make sense to start running a small operating system on the embedded system, to manage these multiple processes.

Embedded environments typically have special requirements for an operating system, most commonly the need for what's called *hard real time*. Regular operating systems may switch between processes in a way that, from the programs' point of view, seems random; their device drivers will often use buffering and interrupts to read and write data also at apparently random times. Such behaviors would be catastrophic for, say, a precision

industrial robot controller, working in microseconds and micrometers, as they would interfere with its required level of precision motion in the real world. A hard real-time operating system (RTOS)—such as SMX, QNX, FreeRTOS, or Zephyr—is an operating system specifically designed from the ground up to absolutely guarantee the timing of such tasks. This requires different approaches to scheduling and I/O. Typically, an embedded microcontroller is a much lower-power machine than a desktop, so operating system design requirements must also include low computational overheads.

To be used in safety-critical environments, an RTOS, like the microcontroller it runs on, will typically go through an expensive and rigorous safety-assurance process based on either extensive testing or, in the most hardcore cases, formal specification and verification, using mathematics and logic to prove it will always work under various assumptions.

RTOSes are distinguished from *soft* real-time operating systems, such as variants of Linux modified for tasks like computer audio production. In these systems, real time is desirable but not strictly necessary—it won't, say, explode a nuclear power station if it can't be absolutely guaranteed every time—and so occasional slips are tolerated.

Speculative Execution Vulnerabilities

In our study of architecture, we've seen that your computer takes your program and converts it to thousands of different instructions, messes with the order these instructions are run in, tries to execute parts of multiple instructions at once, passes incomplete results between instructions, and secretly updates its microcode to execute in new ways.

Each of these behaviors, and the interactions between them, creates enormous complexity in chip design and function. The resulting chip designs are thus some of the most complex systems known to humanity, with no individual human able to fully understand everything taking place in a CPU. It's natural to ask whether we can thus be confident that our CPU designs are safe and secure when there are so many parts that could go wrong.

The answer to this question was recently found to be "no"—this is why we now have *speculative execution* vulnerabilities, architecture bugs that can enable a process to read the data belonging to another process, such as passwords and bank details. In most cases, this includes the ability for hypervised systems belonging to different users on physical cloud machines to spy on one another. This has been considered a catastrophic security threat to many manufacturers; some consider it the most serious hardware problem of all time. Software patches for the vulnerabilities cause a 5 to 30 percent slowdown in performance, while architects are currently working to redesign hardware to avoid them in their next-generation processors.

Speculative execution vulnerabilities were first discovered in 2018 as bugs called Spectre and Meltdown, and new variants continue to be found at the time of writing. To give a basic understanding of this large class of bugs, we'll examine the Meltdown variant here.

Meltdown is caused by a complex unintended interaction between multiple modern architectural features: speculative execution, virtual memory, CPU kernel mode switching, cache timing effects, and a race condition in indirect addressing. Suppose the target process is running alongside our own process under an operating system. The operating system defines separate areas of memory for the two processes and restricts each process's access to only its own memory space. The memory spaces look like this:

Address	Data
1	
2	
3=BASE	
4=TEST1	FOO
5=TEST2	FOO
6=TEST3	FOO

Address	Target's Data
7	
8=TARGET	PASSWORD
9	
10	
11	
6	

Here, we assume we have access to the source code of the target program that tells us where the user password will be stored in its memory, so the contents of address `TARGET`, written as `*TARGET`, is `PASSWORD`, which we'll assume is known in advance to be an integer from 1 to 3. We want to read this password from our own process. Our own process's address space contains a series of addresses marked `TEST1`, `TEST2`, and `TEST3`. We can store any dummy data at these locations, marked as `FOO`. We'll be reading this data as part of our attack, but we don't actually care what its values are. Let's call the address just before these `BASE`, because it will act as a base address from which we can use offsets to refer to each of the `TEST` addresses.

To attack, we first execute an indirect offset addressing instruction together with a conditional:

```
if (0) LOAD BASE+(*TARGET) else LOAD 1
```

Although the semantics of `if (0)` mean that the condition will never be true—meaning the `LOAD BASE+(*TARGET)` won't be completely performed in the program—eager execution (as in Chapter 8) initially begins to run both branches at the same time. When it does this, `BASE+(*TARGET)` will be evaluated, giving an address that must be one of 4, 5, or 6. The content of the data at this address (one of the three `FOO` items) will then be loaded into cache. (The `FOO` from address 1 is also loaded to cache from the other side of the branch.) While this is happening, the condition is tested and found to be false. At this point the `LOAD BASE+(*TARGET)` instruction is aborted, but its value has already been loaded into cache even though it won't be used any further.

Note that if the condition were true instead of false, the `LOAD BASE+(*TARGET)` would then attempt to complete and at that point, and only at that point, a security exception would occur as the `TARGET` address is tested for security and found to lie in another process's address space. But because the condition is actually false, this test is never performed.

Once the value is loaded into the cache, we run a cache timing attack:

```
for (i=1:3) time(LOAD BASE+i)
```

All three of the instructions in the loop succeed, loading the three FOO values into registers from their three memory locations. But if we time each of these three LOADs, we'll find that one of them is faster than the others because it was cached during the speculative execution. If PASSWORD=i, then LOAD BASE+i is fast, because (BASE+i) was cached. Measuring these times and finding the fast one reveals the value of i, which is equal to PASSWORD, as required.

The Meltdown vulnerability existed undetected in almost all major commercial CPUs for 20 years! It may have been known and exploited by secret state actors during this time, but it hasn't yet been exploited by any other malware as far as we know.

The public disclosure sequence of Meltdown in 2017 was a model of how ethical security bug disclosure can and should work. Following public discovery by security researchers, the manufacturers were first informed in secret. Researchers, CPU manufacturers, and operating system programmers then worked together to patch the bug at the operating system software level for all major operating systems. These operating systems were updated in the field by pushing automatic updates to users.

The operating system level software patch is called KAISER. Here, the operating system randomizes process memory locations to prevent a Meltdown attack from knowing which addresses to search for target data. This is still not completely secure but makes the bug much harder to exploit. After user machines had been patched with KAISER, the discoverers of Meltdown published their findings in 2018, first immediately on the pre-print arXiv server, then submitted for formal academic peer review, which completed and published in 2020.

CISC processors are constructed using microcode that enables their hardware to be "rewired" to some extent by CPU firmware updates; this provides a stronger fix for CISC users. Pushing microcode updates is a more difficult and dangerous procedure than patching operating system software, and developing hardware patches also takes longer, in part because of the extensive testing required before allowing a patch to be pushed out. The cost of "bricking" millions of users' processors is higher than damaging their operating system, which could be more easily reinstalled in the event of a bad update. Microcode patch development thus continued after publication of the Meltdown paper and was later pushed out as firmware updates for CISC users.

The new microcode adds logic to clear cache following all speculative executions, removing the vulnerability. However, this has a cost of a significant performance hit, typically producing a 5 to 30 percent slowdown. Pushing such a performance hit onto users—usually automatically, without telling or asking them—led to some lively debate, especially between the operating system programmers whose work on software-level patches was being replaced by the microcode patches.

At the time of writing, CPU architects are working to redesign their basic architectures to fix Meltdown properly at the hardware level. In 2022, some of these fixes for Meltdown were reported by researchers to have introduced a new speculative execution bug, which they named Retbleed. This may become an ongoing game of whack-a-mole, providing employment for architects for many years to come.

Exercises

A 6502 Kernel

Read the assembly code for Joachim Deboy's minimal 6502 kernel at *http://6502.org/source/kernels/minikernel.txt*. Explain where the IRQs, saves, and restores occur. Try to make an x86 or RISC-V version of the same idea.

Speculative Execution Vulnerability Audit

Find out if and how your own computer has been patched for speculative execution vulnerabilities. For Linux, `lscpu` may show some relevant information.

Further Reading

- The definitive textbook on operating systems is Andrew Tanenbaum and Herbert Bos, *Modern Operating Systems*, 4th ed. (Hoboken: Pearson, 2014).
- For a list of all the subroutines Linux provides for you to call from your x86 code, see R.A. Chapman, "Linux System Calls for x86," *https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/*.
- For more on the Meltdown vulnerability, see M. Lipp et al., "Meltdown: Reading Kernel Memory from User Space," *Communications of the ACM* 63, no. 6 (2020): 46–56.

ACKNOWLEDGMENTS

Thanks to my editors Alex Freed and Nathan Heidelberger for making this much more readable and beautiful, and to Bill Pollock at No Starch Press for believing in the project.

Andrew Bower at Broadcom/Xilinx/AMD went far beyond the call of duty in tech review, with detailed analysis of the Baby and many obscure, fascinating improvements.

At Lincoln, Greg Cielniak, Chris Headleand, and Kevin Jacques helped with my teaching, and Chris Waltham helped with embedded systems. Mark Taylor at Google helped with clouds now and Commodores back then. Nick New at Optalysys, Alex Bradbury at LowRisc, and Andrew Richards at Codeplay/Intel/Khronos helped with technical details.

I'd also like to thank my own undergrad teachers at Cambridge, including Maurice Wilkes, Alan Mycroft, Arthur Norman, Peter Robinson, Simon Moore, Larry Paulson (especially for persuading me not to change to maths to avoid the hardware), John Daugman, Bill Clocksin, and Jean Bacon; Andrew Hodges at Oxford and Stephen N.P. Smith from Algometrics for discussions about World War II computing; my parents for buying my first BBC Micro and Archimedes, which must have cost as much as cars back then; schoolteachers Messrs Yates, Neil, Reid, and Baker, for ignoring missed PE lessons, resulting in enhanced access to wonderful Archimedeses, BBC Micros, and musical Atari STs; Anjee Sian and Raja Ram from TIP Records for introducing me to silk-screening and DSP filters; and Alton and David Horsfall for BBC Micro magazines and books.

Danielle Scullion and Adam Wood transcribed drafts of Chapters 8, 10, and 11 from my lectures. The x86 examples and bootloader are based on

Project Metro by my students Harry G. Riley, Kristaps Jurkans, and Olegs Jakovlevs.

Thanks to the Creative Commons community for many of the images in this book and to the open source software and hardware communities for the tools and systems used.

FIGURE CREDITS

All images are from the public domain or courtesy of the author unless otherwise specified.

Copies of the Creative Commons licenses can be found by visiting the following websites or by sending a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA:

CC BY 2.0: *https://creativecommons.org/licenses/by/2.0/deed.en*

CC BY 2.5: *https://creativecommons.org/licenses/by/2.5/deed.en*

CC BY 3.0: *https://creativecommons.org/licenses/by/3.0/*

CC BY 4.0: *https://creativecommons.org/licenses/by/4.0/*

CC BY-ND: *https://creativecommons.org/licenses/by-nd/2.0/deed.en*

CC BY-SA 2.0: *https://creativecommons.org/licenses/by-sa/2.0/deed.en*

CC BY-SA 2.5: *https://creativecommons.org/licenses/by-sa/2.5/deed.en*

CC BY-SA 2.5-it: *https://creativecommons.org/licenses/by-sa/2.5/it/deed.en*

CC BY-SA 3.0: *https://creativecommons.org/licenses/by-sa/3.0/deed.en*

CC BY-SA 4.0: *https://creativecommons.org/licenses/by-sa/4.0/deed.en*

The original images can be found on the book's Git repo at *https://gitlab.com/charles.fox/comparch*.

Introduction

Figure 1. Courtesy of Fastily, used under CC BY-SA 4.0. Cropped from original.

Figure 2. Courtesy of Evan-Amos, used under CC BY-SA 3.0. Labels added.

Figure 3. Courtesy of Manolis Angelakis, used under CC BY 4.0.

Figure 4. Adapted from "XO Motherboard," courtesy OLPC, used under CC BY-SA 3.0.

Figure 5. Courtesy of Raimond Spekking, used under CC BY-SA 4.0. Labels added.

Figure 6. Courtesy of Phiarc, used under CC BY-SA 4.0.

Chapter 1

Figure 1-2. Courtesy of Francesco D'Errico.

Figure 1-3. Courtesy of Joeykentin, used under CC BY-SA 4.0.

Figure 1-4. Courtesy of Dave Fischer, used under CC BY-SA 3.0.

Figure 1-5. Courtesy of the Schøyen Collection, Oslo and London, MS 5112.

Figure 1-6. *left:* Courtesy of Marsyas, used under CC BY-SA 3.0. *right:* Courtesy of @NatureVideoChannel, used under CC BY 3.0.

Figure 1-7. Courtesy of Gts-tg, used under CC BY-SA 4.0.

Figure 1-8. *left:* Courtesy of Università di Bologna, used under CC BY-ND. *right:* Sketch created by author using *https://runwayml.com*.

Figure 1-10. *right:* Courtesy of Nico71 - Nicolas LESPOUR - *https://www.nico71.fr*.

Figure 1-11. Courtesy of Clem Rutter, Rochester, Kent, used under CC BY 3.0.

Figure 1-12. *left:* Courtesy of Steve Slater, used under CC BY 2.0. *center:* Courtesy of Andrzej Barabasz, used under CC BY-SA 3.0. *right:* Courtesy of Jennifer Schug, used under CC BY-SA 4.0.

Figure 1-13. Courtesy of Jitze Couperus, used under CC BY 2.0.

Figure 1-14. Courtesy of Mrjohncummings, used under CC BY-SA 2.0.

Figure 1-15. Courtesy of bad germ, used under CC BY-SA 3.0.

Figure 1-16. Courtesy of Andy Dingley, used under CC BY 3.0.

Figure 1-18. *left:* Courtesy of Adam Schuster, used under CC BY 2.0. Cropped from original.

Figure 1-20. Courtesy of MesserWoland, used under CC BY-SA 3.0. Text modified from original.

Figure 1-21. Courtesy of Joe Haupt, used under CC BY-SA 2.0.

Figure 1-24. Courtesy of Magnus Hagdorn, used under CC BY-SA 2.0.

Figure 1-26. Courtesy of Mister rf, used under CC BY-SA 4.0.

Figure 1-29. Courtesy of Thomas Nguyen, used under CC BY-SA 4.0.

Figure 1-30. Courtesy of Peter Hamer, used under CC BY-SA 2.0.

Figure 1-32. Courtesy of Thomas Schanz, used under CC BY-SA 3.0.

Figure 1-33. Courtesy of Argonne National Laboratory, used under CC BY-SA 2.0.

Chapter 2

Figure 2-2. © Marie-Lan Nguyen, Wikimedia Commons, CC BY 2.5.

Figure 2-3. © Marie-Lan Nguyen, Wikimedia Commons, CC BY 2.5. Cropped from original.

"Decimal Computers" box. *Babbage's Analytical Engine gears:* © 1982, IEEE, used with permission. *UNIVAC console:* Courtesy of Ik T, used under CC BY 2.0.

Chapter 3

Figure 3-1. Courtesy of Antonio Georgiev, used under CC BY-SA 4.0. Cropped from original.

Figure 3-4. Courtesy of Silvio Peroni, used under CC BY 4.0.

Figure 3-5. Courtesy of Pubby, Stack Exchange, used under CC BY-SA 3.0.

"ALU Mechanisms" box. *Babbage's Analytical Engine gears:* © 1982, IEEE, used with permission.

Figure 3-6. © 1982, IEEE, used with permission.

Chapter 4

Figure 4-2. Courtesy of Svjo, used under CC BY-SA 3.0. Font and arrows altered from original.

Figure 4-5. Courtesy of Cmglee, used under CC BY-SA 3.0. Cropped from original.

Figure 4-10. Courtesy of Svjo, used under CC BY-SA 3.0. Font and arrows altered from original.

Figure 4-17. Courtesy of gratuity, used under CC BY 3.0.

Figure 4-18. Courtesy of Cyril BUTTAY, used under CC BY-SA 3.0. Font and arrows altered from original.

Figure 4-20. Courtesy of Intel Free Press, used under CC BY-SA 2.0. Cropped from original.

Chapter 5

Figure 5-10. Courtesy of Dgarte, used under CC BY-SA 3.0.

Figure 5-20. *left:* Courtesy of Paebbels, Stack Exchange, used under CC BY-SA 3.0. *right:* Courtesy of Eladio Delgado Mingorance, used under CC BY-SA 4.0.

Chapter 9

Figure 9-1. Courtesy of Bill Bertram, used under CC BY 2.0. Labels added.

Figure 9-2. *left:* Courtesy of Bill Bertram, used under CC BY 2.5. *right:* Courtesy of Eric Gaba, Wikimedia Commons user Sting, used under CC BY-SA 3.0.

Chapter 10

"Historical RAMs" box. *Mercury delay lines:* © Department of Computer Science and Technology, University of Cambridge, reproduced by permission.

Figure 10-3. Courtesy of _brouhaha_, used under CC BY-SA 2.0.

Figure 10-4. Courtesy of Encheart and Glogger, used under CC BY-SA 3.0 with modifications.

Figure 10-5. Courtesy of An-d, used under CC BY-SA 3.0. Cropped from original.

Figure 10-6. *MROM:* Courtesy of Silicon Pr0n, used under CC BY 4.0. *PROM:* Courtesy of Raimond Spekking, used under CC BY-SA 4.0. *EPROM:* Courtesy of Javier Pérez Montes, used under CC BY-SA 4.0. *EEPROM:* Courtesy of Raimond Spekking, used under CC BY-SA 4.0. *SD card:* Courtesy of Uwe Hermann, used under CC BY-SA 4.0.

Figure 10-14. Courtesy of Jemimus at Flickr and Robert at Wikimedia Commons, used under CC BY 2.0.

Figure 10-15. Courtesy of Norman Bruderhofer, *www.cylinder.de*, used under CC BY-SA 3.0.

Figure 10-16. *left:* Courtesy of Norman Bruderhofer, used under CC BY-SA 2.5. *right:* courtesy of DJpedia, used under CC BY-SA 2.0.

Figure 10-19. Courtesy of Evan-Amos, used under CC BY-SA 3.0.

Figure 10-20. Courtesy of Cmglee, used under CC BY-SA 3.0. Modified from original.

Figure 10-21. Courtesy of ChrisDag, used under CC BY 2.0.

Chapter 11

Figure 11-1. Compiled with images from the public domain and the following: *Amiga1000:* Courtesy of Kaiiv, used under CC BY-SA 3.0. *A500:* Courtesy of Bill Bertram, used under CC BY-SA 2.5. *Sinclair ZX80:* Courtesy of Daniel Ryde, used under CC BY-SA 3.0. *ZX Spectrum:* Courtesy of Bill Bertram, used under CC BY-SA 2.5. *QL:* Courtesy of Ewx, used under CC BY-SA 2.5. *CPC464:* Courtesy of Bill Bertram, used under CC BY-SA 2.5. *PC200:* Courtesy of Marcin Wichary, used under CC BY 2.0. *5150:* Courtesy of Rama & Musée Bolo, used under CC BY-SA 2.0. *PC-XT:* Courtesy of Ruben de Rijcke, used under CC BY-SA 2.0 with modifications. *PC-AT:* Courtesy of MBlairMartin, used under CC

BY-SA 4.0 with modifications. *ST:* Courtesy of Bill Bertram, used under CC BY 2.5. *7800:* Courtesy of Evan-Amos, used under CC BY-SA 3.0. *Lynx:* Courtesy of Evan-Amos, used under CC BY-SA 3.0. *IIe:* Courtesy of Pratyeka, used under CC BY-SA 4.0. *Macintosh:* Courtesy of *http://www.allaboutapple.com*, used under CC BY-SA 2.5-it. *Mac II:* Courtesy of Alexander Schaelss, used under CC BY-SA 3.0 with modifications. *Electron:* Courtesy of Bilby, used under CC BY 3.0. *Master:* Courtesy of Dejdżer/Digga, used under CC BY-SA 2.0. *A3000:* Courtesy of Binary-sequence, used under CC BY-SA 3.0. *TRS-80 3:* Courtesy of Bilby, used under CC BY 3.0. *TRS-80 4:* Courtesy of Blake Patterson, used under CC BY 2.0 with modifications. *1400 LT:* Courtesy of DigitalIceAge, used under CC BY 4.0.

Figure 11-2. Courtesy of Giorgio Moscardi, used with permission. From the open source hardware Raemixx500 project, *https://github.com/Sukko Pera/Raemixx500*.

Figure 11-4. Courtesy of Pauli Rautakorpi, used under CC BY 3.0.

Figure 11-5. Courtesy of Donald F. Hanson.

"The Monster 6502" box. Photo courtesy of Eric Schlaepfer and Windell Oskay, *https://monster6502.com*.

Figure 11-11. Courtesy of Pauli Rautakorpi, used under CC BY 3.0.

Figure 11-12. Courtesy of *https://www.amigawiki.org/doku.php?id=de:models:a500*, used under CC BY-SA 3.0. Labels added.

Figure 11-14. Courtesy of Job at the English Wikipedia, used under CC BY-SA 3.0.

Figure 11-16. Courtesy of Opersing2688, used under CC BY-SA 3.0.

Chapter 12

Figure 12-1. Courtesy of oomlout, used under CC BY-SA 2.0.

Figure 12-3. Courtesy of Tim Mathias, used under CC BY-SA 4.0 with modifications.

Figure 12-4. *left:* Courtesy of SparkFun, used under CC BY 2.0. *right:* Courtesy of Les Pounder from Blackpool, UK, used under CC BY-SA 2.0, cropped from original.

Figure 12-5. Courtesy of oomlout, used under CC BY-SA 2.0.

Figure 12-6. Courtesy of Eelco, used under CC BY-SA 4.0.

Figure 12-7. Courtesy of Mediaquark, used under CC BY-SA 4.0. Font altered from original.

Figure 12-8. Courtesy of BregesT65421354, used under CC BY-SA 4.0.

Figure 12-9. Courtesy of Artium, Stack Exchange, used under CC BY-SA 3.0.

Figure 12-10. Courtesy of Matarasephotos, used under CC BY 3.0.

Figure 12-12. Courtesy of Hugh jack at English Wikibooks, used under CC BY-SA 3.0 with modifications.

Chapter 13

Figure 13-3. Courtesy of Gribeco (original) and Moxfyre (derivative), used under CC BY-SA 3.0 with modifications.

Figure 13-4. *left:* Rainer Knäpper, Free Art License (*http://artlibre.org/licence/lal/en/*), *https://commons.wikimedia.org/wiki/File:Scsi_intern_hd68.jpg*, cropped from original. *right:* Courtesy of Tomato86, used under CC BY-SA 4.0.

Figure 13-5. Courtesy of w:user:snickerdo, used under CC BY-SA 3.0.

Figure 13-6. Courtesy of _surovic_, Sketchfab, used under CC BY 4.0.

Figure 13-7. Courtesy of Davi.trip, used under CC BY-SA 4.0 with modifications.

Figure 13-8. *right:* © 2017 libretro, used under MIT License.

Figure 13-9. Courtesy of endolith at Flickr, used under CC BY-SA 2.0.

Figure 13-10. Courtesy of *https://fabiobaltieri.com*, used under CC BY-SA 3.0.

"The PC Boot Process" box. *PC BIOS ROM:* © Raimond Spekking / CC BY-SA 4.0 (via Wikimedia Commons). *BIOS display I/O capabilities:* Courtesy HacKurx, used under CC BY-SA 3.0.

Chapter 14

Figure 14-1. Both images © Raimond Spekking / CC BY-SA 4.0 (via Wikimedia Commons).

Figure 14-2. *left:* Courtesy of ScienceStockPhotos, used under CC BY 4.0. *right:* Courtesy of Natural Philo, used under CC BY-SA 3.0.

Chapter 15

Figure 15-2. Library of Congress, Prints & Photographs Division, LC-DIG-npcc-12637.

Figure 15-3. Courtesy of Immae, used under CC BY-SA 3.0 with modifications.

Figure 15-5. Courtesy of NVIDIA, used with permission.

Figure 15-7. Courtesy of OLCF at ORNL on Flickr, used under CC BY 2.0.

Figure 15-8. Courtesy of alfonso.saborido on Flickr, used under CC BY 2.0.

Figure 15-9. Based on DistortOD, *https://github.com/AlexanderBrevig/DistortOD*.

Chapter 16

Figure 16-1. Courtesy of IBM Research, used under CC BY-SA 2.0.

Figure 16-3. Courtesy of Optalysys.

Figure 16-4. Courtesy of Isaac Webb, used under CC BY-SA 3.0. Font altered from original.

Figure 16-5. Courtesy of BruceBlaus, used under CC BY 3.0 with modifications.

Figure 16-6. Derived from Figure 3 in M. Bennett, "An Attempt at a Unified Theory of the Neocortical Microcircuit in Sensory Cortex," *Frontiers in Neural Circuits* 14, no. 40 (2020), *https://doi.org/10.3389/fncir.2020.00040*.

Figure 16-7. *left:* Courtesy of Selket, used under CC BY-SA 3.0 with modifications.

INDEX

Numbers

328, 289–290
386, 304, 306
486, 304, 306, 307, 317
4004, 34, 109–110, 305
6502, 33, 35, 205, 245, 248–260, 266, 278, 280, 289, 342, 436
68000, 266–269
7400, 120–121, 128
8080, 305
8086, 190, 304, 318, 379
8087, 190, 317

A

abacus, 7, 43
  addition, 8
  complex algorithms, 9
accumulator architecture
  abacus, 8
  Manchester Baby, 165
  mechanical calculator, 13
  vs. more registers, 182
Acorn Computers, 342
adders, 140
  carry-save, 143
    in LogiSim, 154
  full adder, 140
  half adder, 140
  ripple-carry, 142
addressing modes, 184
  68000, 267
  x86, 311
address space, 201, 217, 267
Adelman, Leonard, 407
adiabatic quantum computing, 419
Advanced Vector Extensions (AVX), 362
algorithm, 8
Altera, 131
ALU. *See* arithmetic logic unit
AMD, 131, 304–305
amd64, 307
American Standard Code for Information Interchange (ASCII), 64–66, 317
Amiga, Commodore, 269–271, 279
analog computing
  differential analyzers, 20
  transistors, 104
  water computing, 4
Analytical Engine, 17, 77
  arithmetic, 80
  arithmetic logic unit, 83
  as basis of ENIAC, 28
  branching, 81
  data representation, 57
  registers, 83
AND gates, 115–116
  multi-input, 136
  poor man's gate, 111
Antikythera mechanism, 10–11, 14
application-specific integrated circuit (ASIC), 108
Arabic numerals, 48
architecture, instruction set (ISA), xxxiii, 183, 302
Arduino, 288
arithmetic logic unit (ALU), 83
  6502, 254, 257
  68000, 269
  Analytical Engine, 80, 83
  Manchester Baby, 158, 166, 173
  RISC-V, 347
  x86, 312
ARM, 343
array operator, 136
arrays, 62
ASCII, 64–66, 317

assemblers, 160
Manchester Baby, 159
RISC-V, 345
vasm, 279
assembly languages, 160
assembly programming, xxii
ATmega328, 289–290
in Arduino, 288
without Arduino, 293
ATX, xxviii
audio
cards, 327
chiptunes, 265
data, 68
signal processing, 294
automated flute, 12
AVX (Advanced Vector Extensions), 362

B

Babbage, Charles, xxiii, 16–17, 77
backspace fails, 64
Banu Musa brothers, 12
bare metal, xxii
barrel organ, 15
barrel pins, 303
base, multiplication of, 50
BBC Micro, 35, 342
Berkeley Educational cores, 349
big data analytics, 22, 39
big endians, 218
billiard-ball computer, 119
binary, 51
BIOS, 333
booting, 431
Commodore 64, 260
Commodore Amiga, 270
PC, 329
in retro computers, 246
x86, 316
bit, 51, 217
Bletchley Park, 25
and parallelism, 357
blitting, 271
Bombes, 26, 28
Boole, George, 126
Boolean algebra, 115
Boolean logic, 122, 127
BOOM, 349
booting, 329, 331, 333, 430
branching
6502, 258
68000, 268
Analytical Engine, 18, 81, 87
hazards, 192
Manchester Baby, 158, 175
Nvidia, 368
prediction, 195
RISC-V, 346
SIMD, 370
statistics of, 195
x86, 314
breadboard, 107, 122, 128, 288
Bronze Age, 7
bubbling, 194
buffer, 207
double, 208
ring, 207
bugs, 30
burning
bootable USB, 333
firmware, xxxi
optical discs, 239
bus, 322
architecture, 203
Ethernet, 324
hierarchy, 321
PC, 319
PCIe, 322
SATA, 323
SCSI, 323
serial, 321
byte addressing, 218
bytes, 55, 217

C

C64 (Commodore 64), 248, 260, 279, 343
cache, 226
advanced architectures, 232
block, 228
direct-mapped, 229
in LogiSim, 242
fully associative, 230
lines, 228
read policies, 229
set associative, 231

tag, 228
write-back, 231
write policies, 231
write-through, 232
calculation, 76
calculators vs. computers, 4
calling convention, 189, 316, 344
cameras, 351
capacitors, 222
in DRAM, 222
in touchscreens, 351
carry operation, 8
carry-save adder, 143
cathode ray tube (CRT)
display device, 275
memory, 221
central processing unit (CPU), xxvi, 155
Analytical Engine, 77
building from TTL chips, 129
locating, xxvi
CERN, 37, 420
chip, 108
chiplets, 379
chips, 34
chiptunes, 265
Chisel, 150, 152, 349, 353
Church computer, 5, 26, 28, 40–41
Church's thesis, 5. *See also* Church computer
CISC, 183, 192, 199
design philosophy, 302
microprogramming, 303
similarities with RISC, 352
clock, 107, 147, 250
Commodore Amiga, 270
piezoelectric, 107
speed, 107
Moore's law, 110
x86, 307
closed loop, 184
cloud computing, 40, 386, 398
cloud storage, 386
cluster, 39
CMOS (complementary metal-oxide semiconductor), 109, 118, 120
co-location, 386
Colossus, 28
combinatorial logic, 136
Commodore, 35, 246, 265
Commodore 64 (C64), 248, 260, 279, 343
Commodore Amiga, 269–271, 279
compilers, 36, 136
dataflow, 388
GPU, 375
hints, 195
optimizing, xxii
with SIMD, 364
Complex Interface Adapter (CIA), 202
compound notation, 52
compression, 69, 403
computable numbers, 61–62
computation theory, 77
computer
architecture, xxxiii
definition of, 4
design, xxxiii
Commodore 64, 260
Commodore Amiga, 269
desktop PC, 318
smart, 349
inventor of, 41
organization, xxxiii
concurrency, 426
conditional jump, 82
connectors
audio, xxxi
MIDI, 277
ribbon cables, 321
serial port, 276
containerization, 432
controller area network (CAN), 287
control unit (CU)
6502, 255
68000, 268
Analytical Engine, 84
Manchester Baby, 167
RISC-V, 346
x86, 314
convolution, 403
cortical columns, 410
counter, 148, 284
example use, 168
CPU. *See* central processing unit
cryptanalysis, 24
cryptography, 24

cryptology, 24
    electronic, 27
    quantum, 419
CUDA, 375
current, electrical, 101
Cyrix, 306

D

data
    compression, 403
    hazard, 192
    measuring, 70
    structures, 69
data center, 241
    dataflow, 387
        languages, 388
datasheets, xxxiv
da Vinci, Leonardo, 13
decoding, 75, 86, 138, 168, 171, 255
decoherence, 419
de Coriolis, Gaspard-Gustave, 21
deep learning, xxiv, 372
demultiplexer, 139
depletion zone, 99
design philosophy
    CISC, 302
    Internet of Things, 40
    mindful computing, 298
    open source hardware, 398
    RISC, 341
    Ubicomp, 298
desktop computers, 41
    inside, xxvi
device drivers, 427
devices, 201
diesel age, 22
die shot, 120, 250
    68000, 266
    x86, 379
Difference Engine, 16
differential analyzers
    electromechanical, 23
    mechanical, 20
differential equations, 21, 24
digital logic
    laying out, 128
    for operand forwarding, 198
digital signal processor (DSP), 294
Digital Versatile Disc (DVD), 240
diode
    heat, 95
    p-n junction, 96
    vacuum tube, 96
directionality, 94
direct mapping, 229
direct memory access (DMA), 211
    Amiga, 270
    in graphics, 325
    remote, 382
dirty bit, 228, 231
discrete Fourier transform
        (DFT), 403
disc, optical, 239
disk, 236
    floppy, 238
    hard, 238
distributed computing, 383
DMA in blitting, 271
DNA computing, 405
doping, 98
    physical region layout, 120
double buffers, 208
D-type flip-flop, 148, 164
dynamic RAM (DRAM), 222

E

eager execution, 194
Eckert, J. Presper, 28
EEPROM, 225, 226
electrical current, 101
electrically erasable programmable
        ROM (EEPROM), 226
electromechanical computing, 22
electrons, 97
embedded systems, 281
    Arduino, 287–292
    CPU-based, 283–295
    design principles, 282–283
    embedded I/O, 284–287
    microcontrollers, 283–284
    no CPU, 295–298
    vs. smart devices, 340
    washing machine, xxxi
emulators, xxiv
    6502, 278
    Amiga, 279

Analytical Engine, 77, 88
Arduino, 299
Commodore 64, 279
Manchester Baby, 179
encoder, 138
endianness, 218
energy usage, xxiv
billiard-ball model, 119
in CPU, 38
ENIAC, 28
as Church computer, 41
programmers, 29
switches, 103
Enigma, 24–26
entropy, 94
erasable programmable ROM (EPROM), 226
error correction code RAM (ECC-RAM), 224
Ethernet, 324
executable file, 160
execution, 79, 86, 172
exponent, 50

F

fabrication, 108, 399
plant (fab), 111
Fairchild Semiconductor, 305
feedback, 145
fetching, 75, 85–86, 170
field-effect transistors (FETs), 105, 106, 108–109
field programmable gate array (FPGA), 131, 297, 398
neural networks, 390
firmware, xxxi
fixed-point numbers, 60
in DSP, 294
flash memory, 226, 240
Fleming, John, 27
flip-flop
clocked, 147
D-type, 148
SR, 147
floating point
numbers, 61
registers, 189
in RISC-V, 347–348
unit (FPU), 189–190, 199
in x86, 317
floppy disk, 238
flow control, 175–176, 314
Flowers, Tommy, 28
form factor, xxviii
forward bias, 100
Fourier transform, 69
Fourmilab Analytical Engine emulator, 78–79, 88
FPGA. *See* field programmable gate array
front side bus (FSB), 204
full adder, 140
fully associative cache, 230
fuses in programmable logic arrays, 130

G

garbage, 119
general relativity (GR), 421
GLSL, 326, 377
God, existence of, 126
golden age
1980s, 35
Islamic, 12, 48
new, xxiii, 38, 349, 398
GPU (graphics processing unit), 39, 324, 364
locating, xxviii
graphics cards, 324
grid computing, 384
Grover's algorithm, 418
guitar
amplifier
model of sequential logic, 145
tubes, 27
effects, 294

H

half adder, 140
halting, 156
hard disk, 238
hardware description languages, 150–151
Harvard architecture, 20, 29–30, 41
hash function, 228
hazards, 192
branching, 192
correction of, 193

hazards *(continued)*
data, 192
structural, 193
heat, 38, 119
"Hello, world"
in BIOS, 336
in operating system, 429
hexadecimal, 53–55
hex editors, 54
high-performance computing (HPC), 382
hippocampus, 413
history, speculative, 43
history of computing, 3
1950s, 31
1960s, 32
1970s, 34
1980s, 35
1990s, 36
2000s, 37
2010s, 38
2020s, 40
Bronze Age, 7
diesel age, 22
electrical age, 27
Iron Age, 9
Islamic golden age, 12
Renaissance and Enlightenment, 13
Steam Age, 14
Stone Age, 6
holes, 98
Hollerith machine, 22, 26
Hubbard, Rob, 265
human computers, 357
hypercomputation, 421
hyperthreading, 198
hypervisor mode, 432

I

IBM, 22, 26, 32, 35, 42, 304
5150, 318
atomic research, 400
IEEE 754, 61
images, 67
input/output (I/O), 201
embedded, 284
for memory access, 233
modules, 201, 206, 323
PC, 321
retro devices, 275
instructionless parallelism, 387
instruction set, 79, 183
6502, 255
68000, 267
amd64, 307
Analytical Engine, 79
DSP, 294–295
Manchester Baby, 30, 156
RISC-V extensions, 347
SIMD, 359
size of, 183
instruction set architecture (ISA), xxxiii, 183, 302
integers, 59, 84
integrated circuits (ICs), 34, 117, 250
vs. cortex, 412
Intel, 131, 304–305
Inter-Integrated Circuit bus, 286–287
internal registers, 165
Internet of Things (IoT), 40, 253
interrupt architecture, 210
interrupt request (IRQ), 210, 250
ion channels, 408
Ishango bone, 6–7

J

Jacquard loom, 15
jumps, 81–82
6502, 258
68000, 268
Analytical Engine, 81
conditional, 82
Manchester Baby, 175
Nvidia, 368
RISC-V, 346
x86, 314
Jupiter RISC-V simulator, 352

K

KERNAL (C64), 263
kernel, 426, 436
keyboard, 276, 328
Khronos, 365

L

labels
6502, 258
x86, 314

ladder logic, 296
laptop, xxviii
Lebombo bone, 6–7
LEGO, 10, 16
light-emitting diode (LED), 101
Linear Tape Open (LTO), 235–236
Linux architecture information commands, xxxiv
little endian, 218
loaders, 428–429
load instructions, 77
  6502, 256
  Amiga, 267
  Analytical Engine, 80
  Arduino, 291
  Manchester Baby, 157, 172
logic
  as arithmetic, 123
  Boolean, 122
  entailment, 124
  proof, 124
logic gates, 114–115
  AND, 115, 119
  on chips, 120
  clocked, 147
  making from transistors, 117
  multi-input, 136
  NAND, 115
  networks of, 116
  vs. neurons, 409
  NOT, 115
  OR, 115
  on TTL chips, 120
  universal, 117
  XOR, 115
logic networks, simplifying, 127
LogiSim
  adder, 154
  building gates, 133
  cache, 242
  floating-point unit, 199
  as an HDL, 150
  Manchester Baby implementation, 176
  multiplier, 154
  pipelining, 199
  SRAM, 242
loops, 19
Lovelace, Ada, 88
LP-DRAM, 350

M

machine code
  Manchester Baby, 162
  Nvidia, 372
mainboard, xxvi–xxvii, 325
  Arduino, 291
  Commodore 64, 260
  GPU, 325
  laptop, xxviii
  RISC-V, 349
Manchester Baby, 30, 156
  arithmetic, 166
  assemblers, 159
  as Church computer, 41
  complete implementation, 176
  control unit, 167
  decode, 171
  execute, 172
    flow control, 175
    load, 172
    store, 172
  fetch, 170
  instructions
    arithmetic, 158
    constants, 157
    halt, 156
    jumps, 158
    load, 157
    store, 157
  instruction set, 156
  internal structures, 163
  in LogiSim, 174
    attaching SRAM, 242
  machine code, 162
  manual, 180
  programmer interface, 156
  registers, 164
    instruction, 166
    internal, 165
    user, 165
  RTL description, 178
  sample program, 161
map-reduce, 387
mask
  files, 151
  PCB, xxviii

mask *(continued)*
ROM, 225
set, 120
silicon, 108, 129
silkscreen, xxvii
masking (GPU), 369
Mauchly, John, 28
Meltdown, xxiii, 433
memory, 145, 215
10,000-year, 401
address, 217
caches, 226
disk, 236
floppy, 238
hard, 238
embedded, 284
as feedback, 145
flash, 226
hierarchy, 215
modules, 219
offline, 233
optical disc, 239
primary, 217
punch cards, 234
punch tape, 234
RAM, 219
dynamic, 222
error-correcting, 224
historical, 220
LP-DRAM, 350
static, 221, 242
secondary, 233
system, 217
tertiary, 240
memory address register (MAR), 205
memory buffer register (MBR), 205
mercury delay line, 220
microchips, 34
microcontrollers, 283
Arduino, 288
ATmega328P, 288
microprogramming, 303
MIDI (Musical Instrument Device Interface), 277
MIMD, 378
cloud, 386
cluster, 384
decentralized, 384
distributed, 383
grid, 384
NUMA, 381
shared-memory, 378
mindful computing, 298
MMX, 360
model checking, 124
MONIAC, 4
monitors, 324
MOnSter 6502, 33, 253
Moore, Gordon, 34, 109, 305
Moore's law, xxiv, 34, 109, 227, 307, 325, 400
end of, 38
vs. parallelism, 355
in x86, 307
MOS, 265
MOSFETs (metal-oxide-semiconductor FETs), 109
mouse, 328
multicore, 38, 379
multi-input gates, 136
multimedia, 67
multiple instruction, multiple data. *See* MIMD
multiplexer, 139, 168
building, 154
music
chiptunes, 265
MIDI, 277
processing unit, 74
sample-based, 249

N

NaN ("not a number"), 61, 189
NAND gate, 115–116
NASM (Netwide Assembler), 311–312, 314, 317–318, 331–332
natural numbers, 56
negator, 143
netlist files, 151
Netwide Assembler (NASM), 311–312, 314, 317–318, 331–332
neural networks, xxiv
on FPGA, 390
neural processor units (NPUs), 390
neurons, 371, 389, 405, 407
vs. logic gates, 409

neuroscience, 407
non-uniform memory access (NUMA), 381–383
NOP (null operation), 193
NOR gate, 116
Northbridge, 320
“not a number” (NaN), 61, 189
NOT gate, 115
n-p-n transistors, 105
null operation (NOP), 193
NUMA (non-uniform memory access), 381–383
Nvidia, 366

O

odometers, 10
offline memory, 233
OLED (organic LED), 324
online storage, 233
on-the-go (OTG) USB protocol, 324
OOOE (out-of-order execution), 196
opcodes, 160, 256
OpenCL, 377
OpenGL, 325
open loop, 184
open source
  cloud, 399
  hardware, xxiv, 398
  software, 37
operand, 160
operand forwarding, 196
operating systems, real-time, 432–433
optical computing, 26, 402
optical correlators, 403–404
optical disc, 239
optic flow, 328
OR gate, 115–116
out-of-order execution (OOOE), 196

P

parallel
  architectures, xxiv, 355
  programming, xxiv
  thinking, 356, 413
parallelism, instructionless, 387
particle accelerator, 109, 420
particle physics, 420
Pascal’s calculator, 13
  advanced arithmetic, 76
  in digital logic, 148
Patterson, David, 341, 349
PCBs, xxvii–xxviii, 293
Pentium, 306
peripherals, 202
personal computer (PC), xxvi, 36
PETSCII, 262
photolithography
  for camera sensors, 351
  for chips, 129
  for monitors, 324
  for touchscreens, 351
piano roll, 74–75
PIC microcontrollers, 293
piezoelectric properties, 107
pipelining, 190–196, 199
  68000, 267
  vs. hyperthreading, 198
  in LogiSim, 199
  RISC, 341
  x86, 307
p-n junctions, 96
p-n-p transistor, 103
polling, 209
pre-charging, 223
primary memory, 217
principle of locality, 227
printed circuit boards (PCBs), xxvii–xxviii, 293
processor, xxvi
  multicore, 38
program, definition of, 79
programmable logic array (PLA), 130
programmable logic controllers (PLCs), 295
programmable ROM (PROM), 225
programming
  6502, 263
  68000, 267
  assembly, xxii
  Analytical Engine, 88
  Arduino, 291, 299
  to avoid hazards, 193
  Commodore 64, 263
  Commodore Amiga, 271
  in the 8-bit era, xxii, 163

programming *(continued)*
high performance, xxii
Manchester Baby, 179
modern, xxii
Nvidia, 392
Nvidia SASS, 394
parallel, 356
retro, 246
RISC-V, 345
systems, xxii
x86, 308
SIMD, 392
64-bit mode, 333
PROM, 225
pseudo-instructions, 344
PTX, 366
punch cards, 79, 234

Q

quantum computing, 414
adiabatic, 419
decoherence, 419
in practice, 419
qubit, 417
registers, 416
simulation, 417
quantum field theory (QFT), 420
quantum mechanics
cartoon version, 414
math version, 415
quartz, 107
qubit, 417

R

random-access memory (RAM), 77, 150, 219
Analytical Engine, 79
locating, xxvi
Williams tube, 221
rationals, 60
read after write, 192
read-only memory. *See* ROM
real-time operating systems (RTOS), 432
registers
Analytical Engine, 83
floating-point, 189
instruction, 166
internal, 165, 310, 254
Manchester Baby, 164
quantum, 416
RISC style, 341, 344
SIMD splitting, 359
user, 165
6502, 253
68000, 267
choosing number of, 181
x86 status, 309–310
Rejewski, Marian, 25
relay switch, 22
remote DMA, 382
retro architectures, 245
reverse bias, 95–96, 100
reversible computation, 119
ring buffers, 207
ripple-carry adder, 142
RISC, 183, 192, 199, 341
similarities with CISC, 352
RISC-V, 343
arithmetic, 347
community, 349
control flow, 346
floating point, 348
implementations, 348
programming, 345
registers, 341, 344
SIMD, 364
tooling, 349
Rocket, 349
ROM (read-only memory), 225
EEPROM, 226
EPROM, 226
flash, 226
mask, 225
programmable, 225
Roman numerals, 47
Rose's law, 419
Rowhammer, 224
RTOS (real-time operating systems), 432

S

SASS, 372, 394
SATA, 323
screwdrivers, xxxiv
SCSI, 323
secondary memory, 233
sectors, 237

security, xxiii, 428
cracking, 54
embedded, 297
SCADA, 297
secure boot, circumventing, xxix
vulnerabilities
Meltdown, xxiii, 433
Rowhammer, 224
Spectre, xxiii, 433
segmentation, 317
semiconductors, 97
sequencer, 149
sequential logic, 144, 146
serial port, 276
embedded, 286
Serial Storage Architecture (SSA), 323
set associative, 231
SETI, 385
sexagesimal, 52
shaders, 326, 367
Shannon, Claude, 114, 127
shared-memory MIMD, 378
shifter, 137
Shockley, William, 304
SID (Sound Interface Device), 265
silicon, 97
chips, 108
crystal, 98
cubes, 400
ingots, 108
3D architectures, 400
wafers, 108
silk-screening, xxvii
SIMD, 358
on GPU, 364
on RISC-V, 364
on x86, 359
simple machines, 135
adders, 140
vs. cortical columns, 410
decoder, 138
demultiplexer, 139
encoder, 138
multiplexer, 139
shifter, 137
simulator
6502, 278
abacus, 42
Amiga, 279
Analytical Engine, 77, 88
Arduino, 299
Commodore 64, 279
Manchester Baby, 179
quantum computing, 417
RISC-V, 352
single in-line memory modules (SIMMs), 224
Small-Scale Experimental Machine, 30
smart, 340
architectures, 339
cities, xxiii, 40, 398
computer design, 349
computing, xxiii
factories, 40
homes, xxxi, 340, 398
transport, xxiii
smartphones, xxix, xxxv, 340
soldering, xxviii
solenoid, 22
solid-state drives (SSDs), 240
sound card, 327
Southbridge, 320
Space Invaders, 4, 248
Spectre, xxiii, 433
SPIR-V, 376
SRAM (static RAM), 221–222
SR flip-flop, 146
SSE (streaming SIMD extensions), 361–362
stack pointer register, 187–188, 315
stalling, 194
static RAM (SRAM), 221–222
status
flag, 82–83
register
6502, 255
68000, 267
x86, 310
Stone Age, 6
store (write) addresses, 77
6502, 256
Amiga, 267
Analytical Engine, 19, 80
Manchester Baby, 157, 172
stored program, 30

streaming SIMD extensions (SSE), 361–362
structural hazard, 193
subgroups, GPU, 369
subroutines, 186
  6502, 259
  RISC-V, 346
  x86, 315
subtractor, 144
supercomputers, xxiii, 382
supervisor mode, 432
supervisory control and data acquisition (SCADA), 295, 297
swap space, 427
switch, 93, 101
  electrical tube, 102
  telephone exchange, 114
  transistor, 103
  water, 101
  water pressure effect, 105
symmetric multi-processing (SMP), 378
system memory, 217

T

tag, 228
tally sticks, 46–48
tape, 234
taping out, 250
tensor processor units (TPUs), 390
tertiary memory, 240
text representation
  ASCII, 64
  history of, 63–64
  PETSCII, 262
  scrolling, 264
  Unicode, 66
timer, 284
touchscreens, 351
tracks, 237
Tramiel, Jack, 260
transistors, 32, 103
  atomic scale, 399
  field-effect, 105, 106, 108–109
  history of, 32
  vs. ion channels, 408
  making logic gates from, 117
  n-p-n, 105
  optical, 402
  p-n-p, 103
translation lookaside buffer (TLB), 427
truth table, 115, 133
TTL chips, 120, 128
tunnels, 169
Turing, Alan, 26
  on computable numbers, 62
  Manchester Baby program, 161
two's complement, 154

U

Ubicomp, 298
UEFI, 330, 430–431
unconditional jump, 82
Unicode, 66–67
UNIVAC
  data representation, 58
  history of, 31
universal gates, 117, 127
universal serial bus (USB), 323
user registers, 165, 181
UTF-8, 66
UTF-16, 67
UTF-32, 67

V

vacuum tubes, 27, 30
valve
  electrical, 27
  water, 94–95
vasm cross-enabler, 279
Verilog, 150–151
very long instruction words (VLIW), 378
VHDL, 150–151
Via, 306
virtual machine
  in ENIAC, 29
  x86, 332
virtual memory, 427
Visual 6502 project, 251
visual display unit (VDU), 325
volatile memory, 219

W

warrant, voiding, xxxiv
washing machines, xxxi

watchdog, 284
water computer, 4
Watson, Thomas, 26, 32, 40
Wilkes, Maurice, 24, 302–303
Williams tube, 221
wires, copper, 120
    vs. neurotransmission, 410
    on PCBs, xxviii
Wokwi emulator, 299
word, 217
write after read, 192
write after write, 192
write-back, 231
write-through, 232

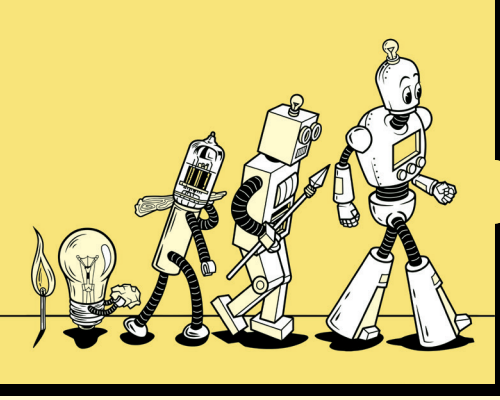
X

x86, 304, 308
    16-bit era, 305
    32-bit era, 306
    64-bit era, 307
    AVX extensions, 362
    back-compatibility, 318
    history of, 304
    MMX extensions, 360
    multicore, 379
    SIMD, 359
    SSE extensions, 361
x87, 317
Xilinx, 131
XOR gate, 115–116

Z

Z3, 26
zero-paging, 257
Zuse, Konrad, 26

The fonts used in *Computer Architecture* are New Baskerville, Futura, The Sans Mono Condensed, and Dogma. The book was typeset with LaTeX 2_ε package `nostarch` by Boris Veytsman *(2008/06/06 v1.3 Typesetting books for No Starch Press).*



A JOURNEY THROUGH COMPUTER HISTORY

Computer Architecture is an in-depth exploration of the principles and designs that have shaped computer hardware through the ages, from counting devices like the abacus, to Babbage's Difference Engine, to modern GPUs and the frontiers of quantum computing.

This engaging blend of history, theory, hands-on exercises, and real-world examples is sure to make for an insightful romp through a fast-changing world. You won't just read about computer architecture, you'll also gain the understanding to touch, build, and program it. You'll explore the basic structures of a CPU by learning to program a Victorian Analytical Engine. You'll extend electronic machines to 8-bit and 16-bit retro gaming computers, learning to program a Commodore 64 and an Amiga. You'll delve into x86 and RISC-V architectures, cloud and supercomputers, and ideas for future technologies.

You'll also learn:

- How to represent data with different coding schemes and build digital logic gates
- The basics of machine and assembly language programming
- How pipelining, out-of-order execution, and parallelism work, in context
- The power and promise of neural networks, DNA, photonics, and quantum computing

Whether you're a student, a professional, or simply a tech enthusiast, after reading this book, you'll grasp the milestones of computer architecture and be able to engage directly with the technology that defines today's world. Prepare to be inspired, challenged, and above all, see and experience the digital world, hands-on.

ABOUT THE AUTHOR

Charles Fox is an award-winning senior lecturer at the University of Lincoln, UK. With degrees from Cambridge, Oxford, and the University of Edinburgh, Fox has published over 100 research papers, which often apply embedded, smart, parallel, and neural architectures to AI and robotics. He's a Fellow of the Higher Education Academy (FHEA) and the author of *Data Science for Transport* (Springer, 2018).

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™

nostarch.com

$59.99 US ($78.99 CDN)

ISBN 978-1-71850-286-4

55999

9 781718 502864