Second Edition

ESSENTIALS OF COMPUTER ARCHITECTURE

DOUGLAS COMER

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

ESSENTIALS OF COMPUTER ARCHITECTURE

Second Edition

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

ESSENTIALS OF
COMPUTER ARCHITECTURE
Second Edition

DOUGLAS COMER

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

SPARC is a registered trademark of SPARC International, Inc. in the United States and other countries.

ARM is a registered trademark of ARM Limited in the United States and other countries.

MIPS is a registered trademark of MIPS Technologies, Inc. in the United States and other countries.

Itanium and Xeon are trademarks of, and Intel and Pentium are registered trademarks of Intel Corporation.

All other trademarks referred to herein are the property of their respective owners.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20161102

International Standard Book Number-13: 978-1-138-62659-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Comer, Douglas, author.
Title: Essentials of computer architecture / Douglas Comer.
Description: Second edition. | Boca Raton : Taylor & Francis, a CRC title,
part of the Taylor & Francis imprint, a member of the Taylor & Francis
Group, the academic division of T&F Informa, plc, [2017] | Includes index.
Identifiers: LCCN 2016041657 | ISBN 9781138626591 (hardback : alk. paper)
Subjects: LCSH: Computer architecture.
Classification: LCC QA76.9.A73 C625 2017 | DDC 004.2/2--dc23
LC record available at https://lccn.loc.gov/2016041657

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Chris, who makes all the
bits of life meaningful

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

PART II Processors 65

Chapter 4 The Variety Of Processors And Computational Engines 69

Chapter 5 Processor Types And Instruction Sets 85

Chapter 15 Buses And Bus Architectures 289

Chapter 16 Programmed And Interrupt-Driven I/O 313

PART V Advanced Topics 355

Chapter 18 Parallelism 359

Chapter 19 Data Pipelining 381

Chapter 20 Power And Energy 395

Chapter 21 Assessing Performance 411

Chapter 22 Architecture Examples And Hierarchy 423

Chapter 23 Hardware Modularity 437

Appendix 1 Lab Exercises For A Computer Architecture Course 445

Appendix 2 Rules For Boolean Algebra Simplification 471

Preface

Hardware engineering has shifted the use of discrete electronic components to the use of programmable devices. Consequently, programming has become much more important. Programmers who understand how hardware operates and a few basic hardware principles can construct software systems that are more efficient and less prone to errors. Consequently, a basic knowledge of computer architecture allows programmers to appreciate how software maps onto hardware and to make better software design choices. A knowledge of the underlying hardware is also a valuable aid in debugging because it helps programmers pinpoint the source of problems quickly.

The text is suitable for a one-semester undergraduate course. In many Computer Science programs, a course on computer architecture or computer organization is the only place in the curriculum where students are exposed to fundamental concepts that explain the structure of the computers they program. Unfortunately, most texts on computer architecture are written by hardware engineers and are aimed at students who are learning how to design hardware. This text takes a different approach: instead of focusing on hardware design and engineering details, it focuses on programmers by explaining the essential aspects of hardware that a programmer needs to know. Thus, topics are explained from a programmer's point of view, and the text emphasizes consequences for programmers.

The text is divided into five parts. Part I covers the basics of digital logic, gates, data paths, and data representation. Most students enjoy the brief look at the underlying hardware (especially because the text and labs avoid minute hardware details). Parts II, III, and IV cover the three primary aspects of architecture: processors, memories, and I/O systems. In each case, the chapters give students enough background to understand how the mechanisms operate and the consequences for programmers without going into many details. Finally, Part V covers the advanced topics of parallelism, pipelining, power and energy, and performance.

Appendix 1 describes an important aspect of the course: a hands-on lab where students can learn by doing. Although most lab problems focus on programming, students should spend the first few weeks in lab wiring a few gates on a breadboard. The equipment is inexpensive (we spent less than fifteen dollars per student on permanent equipment; students purchase their own set of chips for under twenty dollars).

Appendix 2 provides a quick introduction to x86 assembly language and the x64 extensions. Many professors teach x86 and have requested that it be included. The material is in an appendix, which means that professors who choose to focus on a RISC assembly language (e.g., the ARM architecture) can use it for comparison.

The second edition contains two new chapters as well as changes and updates throughout. Chapter 3 on data paths shows the components of a computer and describes how data flows among the components as instructions are executed. The simplified example bridges the gap between digital logic in Chapter 2 and the subsequent chapters that describe processors. Chapter 20 on power and energy covers the basics without going into detail. It explains why a dual-core chip in which each core runs at half speed takes less power than a single core chip that runs at full speed.

We have set up a Web site to accompany the book at:

http://www.eca.cs.purdue.edu

The text and lab exercises are used at Purdue; students have been extremely positive about both. We receive notes of thanks for the text and course. For many students, the lab is their first experience with hardware, and they are enthusiastic.

My thanks to the many individuals who contributed to the book. Bernd Wolfinger provided extensive reviews and made several important suggestions about topics and direction. Professors and students spotted typos in the first edition. George Adams provided detailed comments and suggestions for the second edition.

Finally, I thank my wife, Chris, for her patient and careful editing and valuable suggestions that improve and polish the text.

Douglas E. Comer

About The Author

Dr. Douglas E. Comer, PhD, has an extensive background in computer systems, and has worked with both hardware and software. Comer's work on software spans most aspects of systems, including compilers and operating systems. He created a complete operating system, including a process manager, a memory manager, and device drivers for both serial and parallel interfaces. Comer has also implemented network protocol software and network device drivers for conventional computers and network processors. Both his operating system, Xinu, and TCP/IP protocol stack have been used in commercial products.

Comer's experience with hardware includes work with discrete components, building circuits from logic gates, and experience with basic silicon technology. He has written popular textbooks on network processor architectures, and at Bell Laboratories, Comer studied VLSI design and fabricated a VLSI chip.

Comer is a Distinguished Professor of Computer Science at Purdue University, where he develops and teaches courses and engages in research on computer organization, operating systems, networks, and Internets. In addition to writing a series of internationally acclaimed technical books on computer operating systems, networks, TCP/IP, and computer technologies, Comer has created innovative laboratories in which students can build and measure systems such as operating systems and IP routers; all of Comer's courses include hands-on lab work.

While on leave from Purdue, Comer served as the inaugural VP of Research at Cisco Systems. He continues to consult and lecture at universities, industries, and conferences around the world. For twenty years, Comer served as the editor-in-chief of the journal *Software — Practice and Experience*. He is a Fellow of the Association for Computing Machinery (ACM), a Fellow of the Purdue Teaching Academy, and a recipient of numerous awards, including a USENIX Lifetime Achievement Award.

Additional information can be found at:

www.cs.purdue.edu/people/comer

and information about Comer's books can be found at:

www.comerbooks.com

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter Contents

1

Introduction And Overview

1.1 The Importance Of Architecture

Computers are everywhere. Cell phones, video games, household appliances, and vehicles all contain programmable processors. Each of these systems depends on software, which brings us to an important question: why should someone interested in building software study computer architecture? The answer is that understanding the hardware makes it possible to write smaller, faster code that is less prone to errors. A basic knowledge of architecture also helps programmers appreciate the relative cost of operations (e.g., the time required for an I/O operation compared to the time required for an arithmetic operation) and the effects of programming choices. Finally, understanding how hardware works helps programmers debug — someone who is aware of the hardware has more clues to help spot the source of bugs. In short, the more a programmer understands about the underlying hardware, the better he or she will be at creating software.

1.2 Learning The Essentials

As any hardware engineer will tell you, digital hardware used to build computer systems is incredibly complex. In addition to myriad technologies and intricate sets of electronic components that constitute each technology, engineers must master design rules that dictate how the components can be constructed and how they can be interconnected to form systems. Furthermore, the technologies continue to evolve, and newer, smaller, faster components appear continuously.

Fortunately, as this text demonstrates, it is possible to understand architectural components without knowing low-level technical details. The text focuses on essentials, and explains computer architecture in broad, conceptual terms — it describes each of the major components and examines their role in the overall system. Thus, readers do not need a background in electronics or electrical engineering to understand the subject.

1.3 Organization Of The Text

What are the major topics we will cover? The text is organized into five parts.

> *Basics*. The first section covers two topics that are essential to the rest of the book: digital logic and data representation. We will see that in each case, the issue is the same: the use of electronic mechanisms to represent and manipulate digital information.
>
> *Processors*. One of the three key areas of architecture, processing concerns both computation (e.g., arithmetic) and control (e.g., executing a sequence of steps). We will learn about the basic building blocks, and see how the blocks are used in a modern *Central Processing Unit* (*CPU*).
>
> *Memory*. The second key area of architecture, memory systems, focuses on the storage and access of digital information. We will examine both physical and virtual memory systems, and understand one of the most important concepts in computing: caching.
>
> *I/O*. The third key area of architecture, input and output, focuses on the interconnection of computers and devices such as microphones, keyboards, mice, displays, disks, and networks. We will learn about bus technology, see how a processor uses a bus to communicate with a device, and understand the role of device driver software.
>
> *Advanced Topics*. The final section focuses on two important topics that arise in many forms: parallelism and pipelining. We will see how either parallel or pipelined hardware can be used to improve overall performance.

1.4 What We Will Omit

Paring a topic down to essentials means choosing items to omit. In the case of this text, we have chosen breadth rather than depth — when a choice is required, we have chosen to focus on concepts instead of details. Thus, the text covers the major topics in architecture, but omits lesser-known variants and low-level engineering details. For example, our discussion of how a basic *nand* gate operates gives a simplistic description without discussing the exact internal structure or describing precisely how a gate dissipates the electrical current that flows into it. Similarly, our discussion of processors and memory systems avoids the quantitative analysis of performance that an engineer needs. Instead, we take a high-level view aimed at helping the reader understand the overall design and the consequences for programmers rather than preparing the reader to build hardware.

1.5 Terminology: Architecture And Design

Throughout the text, we will use the term *architecture* to refer to the overall organization of a computer system. A computer architecture is analogous to a blueprint — the architecture specifies the interconnection among major components and the overall functionality of each component without giving many details. Before a digital system can be built that implements a given architecture, engineers must translate the overall architecture into a practical *design* that accounts for details that the architectural specification omits. For example, the design must specify how components are grouped onto chips, how chips are grouped onto circuit boards, and how power is distributed to each board. Eventually, a design must be implemented, which entails choosing specific hardware from which the system will be constructed. A design represents one possible way to realize a given architecture, and an implementation represents one possible way to realize a given design. The point is that architectural descriptions are abstractions, and we must remember that many designs can be used to satisfy a given architecture and many implementations can be used to realize a given design.

1.6 Summary

This text covers the essentials of computer architecture: digital logic, processors, memories, I/O, and advanced topics. The text does not require a background in electrical engineering or electronics. Instead, topics are explained by focusing on concepts, avoiding low-level details, and concentrating on items that are important to programmers.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Part I

Basics Of
Digital Logic
And
Data Representation

The Fundamentals
From Which Computers
Are Built

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter Contents

2

Fundamentals Of Digital Logic

2.1 Introduction

This chapter covers the basics of digital logic. The goal is straightforward — provide a background that is sufficient for a reader to understand remaining chapters. Although many low-level details are irrelevant, programmers do need a basic knowledge of hardware to appreciate the consequences for software. Thus, we will not need to delve into electrical details, discuss the underlying physics, or learn the design rules that engineers follow to interconnect devices. Instead, we will learn a few basics that will allow us to understand how complex digital systems work.

2.2 Digital Computing Mechanisms

We use the term *digital computer* to refer to a device that performs a sequence of computational steps on data items that have discrete values. The alternative, called an *analog computer*, operates on values that vary continuously over time. Digital computation has the advantage of being precise. Because digital computers have become both inexpensive and highly reliable, analog computation has been relegated to a few special cases.

The need for reliability arises because a computation can entail billions of individual steps. If a computer misinterprets a value or a single set fails, correct computation will not be possible. Therefore, computers are designed for failure rates of much less than one in a billion.

How can high reliability and high speed be achieved? One of the earliest computational devices, known as an *abacus*, relied on humans to move beads to keep track of sums. By the early twentieth century, mechanical gears and levers were being used to produce cash registers and adding machines. By the 1940s, early electronic computers were being constructed from vacuum tubes. Although they were much faster than mechanical devices, vacuum tubes (which require a filament to become red hot) were unreliable — a filament would burn out after a few hundred hours of use.

The invention of the transistor in 1947 changed computing dramatically. Unlike vacuum tubes, transistors did not require a filament, did not consume much power, did not produce much heat, and did not burn out. Furthermore, transistors could be produced at much lower cost than vacuum tubes. Consequently, modern digital computers are built from electronic circuits that use transistors.

2.3 Electrical Terminology: Voltage And Current

Electronic circuits rely on physical phenomena associated with the presence and flow of electrical charge. Physicists have discovered ways to detect the presence of electrical charge and control the flow; engineers have developed mechanisms that can perform such functions quickly. The mechanisms form the basis for modern digital computers.

Engineers use the terms *voltage* and *current* to refer to quantifiable properties of electricity: the *voltage* between two points (measured in *volts*) represents the potential energy difference, and the *current* (measured in *amperes* or *amps*) represents the flow of electrons along a path (e.g., along a wire). A good analogy can be made with water: voltage corresponds to water pressure, and current corresponds to the amount of water flowing through a pipe at a given time. If a water tank develops a hole and water begins to flow through the hole, water pressure will drop; if current starts flowing through a wire, voltage will drop.

The most important thing to know about electrical voltage is that voltage can only be measured as the difference between two points (i.e., the measurement is relative). Thus, a *voltmeter*, which is used to measure voltage, always has two probes; the meter does not register a voltage until both probes have been connected. To simplify measurement, we assume one of the two points represents zero volts, and express the voltage of the second point relative to zero. Electrical engineers use the term *ground* to refer to the point that is assumed to be at zero volts. In all digital circuits shown in this text, we will assume that electrical power is supplied by two wires: one wire is a ground wire, which is assumed to be at zero volts, and a second wire is at five volts.

Fortunately, we can understand the essentials of digital logic without knowing more about voltage and current. We only need to understand how electrical flow can be controlled and how electricity can be used to represent digital values.

2.4 The Transistor

The mechanism used to control flow of electrical current is a semiconductor device known as a *transistor*. At the lowest level, all digital systems are composed of transistors. In particular, digital circuits use a form of transistor known as a *Metal Oxide Semiconductor Field Effect Transistor* (*MOSFET*), abbreviated *FET*. A MOSFET can be formed on a crystalline silicon foundation by composing layers of P-type and N-type silicon, a silicon oxide insulating layer (a type of glass), and metal for wires that connect the transistor to the rest of the circuit.

The transistors used in digital circuits function as an on/off switch that is operated electronically instead of mechanically. That is, in contrast to a mechanical switch that opens and closes based on the mechanical force applied, a transistor opens and closes based on whether voltage is applied. Each transistor has three *terminals* (i.e., wires) that provide connections to the rest of the circuit. Two terminals, a *source* and *drain*, have a channel between them on which the electrical resistance can be controlled. If the resistance is low, electric current flows from the source to the drain; if the resistance is high, no current flows. The third terminal, known as a *gate*, controls the resistance. In the next sections, we will see how switching transistors can be used to build more complex components that are used to build digital systems.

MOSFET transistors come in two types; both are used in digital logic circuits. Figure 2.1 shows the diagrams engineers use to denote the two types†.

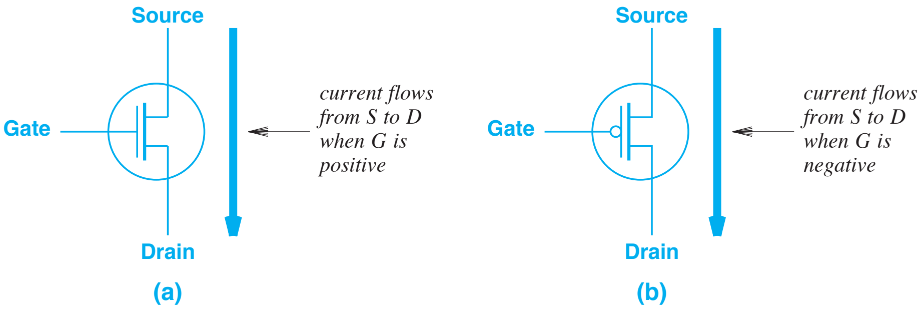

Source
Gate
Drain
current flows from S to D when G is positive
(a)
Source
Gate
Drain
current flows from S to D when G is negative
(b)

Figure 2.1 The two types of transistors used in logic circuits. The type labeled (a) turns on when the gate voltage is positive; transistor labeled (b) turns on when the gate voltage is zero (or negative).

In the diagram, the transistor labeled (a) turns on whenever the voltage on the gate is positive (i.e., exceeds some minimum threshold). When the appropriate voltage appears on the gate, a large current can flow through the other two connections. When the voltage is removed from the gate, the large current stops flowing. The transistor labeled (b), which has a small circle on the gate, works the other way: a large current flows from the source to the drain whenever the voltage on the gate is below the threshold (e.g., close to zero), and stops flowing when the gate voltage is high. The two forms

†Technically, the diagram depicts the *p-channel* and *n-channel* forms of a MOSFET.

are known as *complementary*, and the overall chip technology is known as *CMOS* (*Complementary Metal Oxide Semiconductor*). The chief advantage of CMOS arises because circuits can be devised that use extremely low power.

2.5 Logic Gates

How are digital circuits built? A transistor has two possible states — current is flowing or no current is flowing. Therefore, circuits are designed using a two-valued mathematical system known as Boolean algebra. Most programmers are familiar with the three basic Boolean functions: *and*, *or*, and *not*. Figure 2.2 lists the possible input values and the result of each function.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

A	not A
0	1
1	0

Figure 2.2 Boolean functions and the result for each possible set of inputs. A logical value of zero represents *false*, and a logical value of one represents *true*.

Boolean functions provide a conceptual basis for digital hardware. More important, it is possible to use transistors to construct efficient circuits that implement each of the Boolean functions. For example, consider the Boolean *not*. Typical logic circuits use a positive voltage to represent a Boolean *1* and zero voltage to represent a Boolean *0*. Using zero volts to represent 0 and a positive voltage to represent 1 means a circuit that computes Boolean *not* can be constructed from two transistors. That is, the circuit will take an input on one wire and produce an output on another wire, where the output is always the opposite of the input — when positive voltage is placed on the input, the output will be zero, and when zero voltage is placed on the input, the output will be positive†. Figure 2.3 illustrates a circuit that implements Boolean *not*.

The drawing in the figure is known as a *schematic diagram*. Each line on a schematic corresponds to a wire that connects one component to another. A solid dot indicates an electrical connection, and a small, open circle at the end of a line indicates an external connection. In addition to the two inputs and an output, the circuit has external connections to positive and zero voltages.

Electronic circuits that implement Boolean functions differ from a computer program in a significant way: a circuit operates automatically and continuously. That is, once power is supplied (the + voltage in the figure), the transistors perform their function and continue to perform as long as power remains on — if the input changes, the

†Some digital circuits use 5 volts and some use 3.3 volts; rather than specify a voltage, hardware engineers write V_{dd} to denote a voltage appropriate for a given circuit.

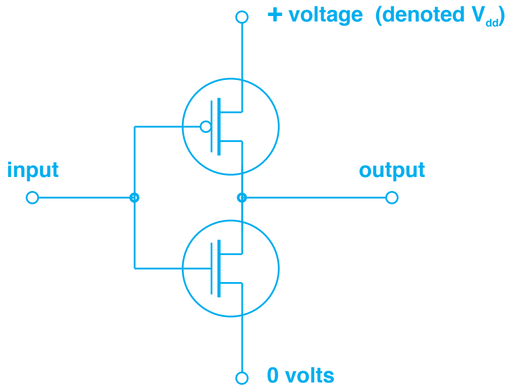


Figure 2.3 A pair of complementary transistors used to implement a Boolean *not*.

output changes. Thus, unlike a function in a program that only produces a result when called, the output of a circuit is always available and can be used at any time.

To understand how the circuit works, think of the transistors as capable of forming an electrical connection between the source and drain terminals. When the input is positive, the top transistor turns off and the bottom transistor turns on, which means the output is connected to zero volts. Conversely, when the input voltage is zero, the top transistor turns on and the bottom transistor turns off, which means the ouput is connected to positive voltage. Thus, the ouput voltage represents the logical opposite of the input voltage.

A detail adds a minor complication for Boolean functions: because of the way electronic circuits work, it takes fewer transistors to provide the inverse of each function. Thus, most digital circuits implement the inverse of *logical or* and *logical and*: *nor* (which stands for *not or*) and *nand* (which stands for *not and*). In addition, some circuits use the *exclusive or* (*xor*) function. Figure 2.4 lists the possible inputs and the results for each function†.

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.4 The *nand*, *nor*, and *xor* functions that logic gates provide.

†A later section explains that we use the term *truth tables* to describe the tables used in the figure.

2.6 Implementation Of A Nand Logic Gate Using Transistors

For the remainder of the text, the details of transistors and their interconnection are unimportant. All we need to understand is that transistors can be used to create each of the Boolean functions described above, and that the functions are used to create digital circuits that form computers. Before leaving the topic of transistors, we will examine an example: a circuit that uses four transistors to implement a *nand* function. Figure 2.5 contains the circuit diagram. As described above, we use the term *logic gate* to describe the resulting circuit. In practice, a logic gate contains additional components, such as *diodes* and *resistors*, that are used to protect the transistors from electrostatic discharge and excessive electrical current; because they do not affect the logical operation of the gate, the extra components have been omitted from the diagram.

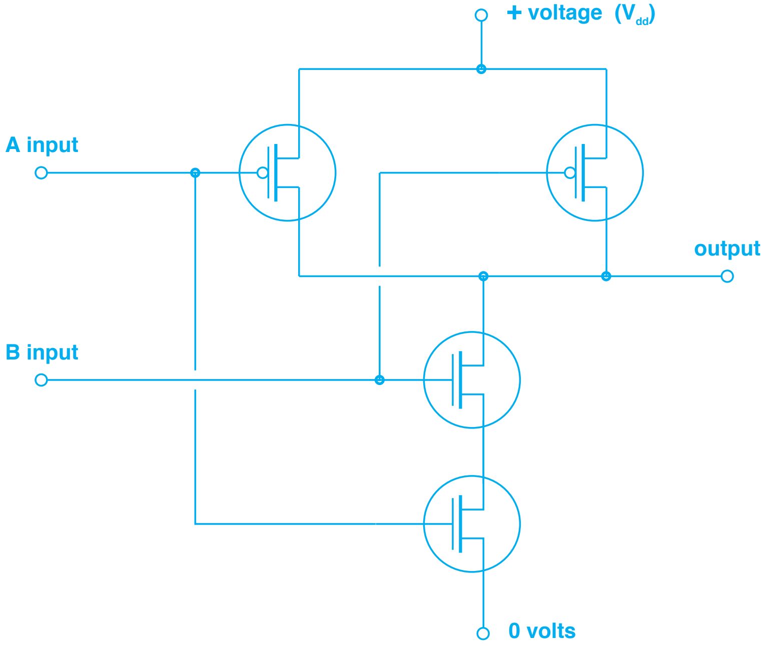


Figure 2.5 Example of four transistors interconnected in a circuit that implements a *nand* logic gate.

To understand how the circuit operates, observe that if both inputs represent logical one, the bottom two transistors will be turned on, which means the output will be connected to zero volts (logical zero). Otherwise, at least one of the top two transistors will be turned on, and the output will be connected to positive voltage (logical one). Of course, a circuit must be designed carefully to ensure that an output is never connected to positive voltage and zero volts simultaneously (or the transistors will be destroyed).

The diagram in Figure 2.5 uses a common convention: two lines that cross do not indicate an electrical connection unless a solid dot appears. The idea is similar to the way vertices and edges are drawn in a graph: two edges that cross do not indicate a vertex is present unless a dot (or circle) is drawn. In a circuit diagram, two lines that cross without a dot correspond to a situation in which there is no physical connection; we can imagine that the wires are positioned so an air gap exists between them (i.e., the wires do not touch). To help indicate that there is no connection, the lines are drawn with a slight space around the crossing point.

Now that we have seen an example of how a gate can be created out of transistors, we do not need to consider individual transistors again. Throughout the rest of the chapter, we will discuss gates without referring to their internal mechanisms. Later chapters discuss larger, more complex mechanisms that are composed of gates.

2.7 Symbols Used For Logic Gates

When they design circuits, engineers think about interconnecting logic gates rather than interconnecting transistors. Each gate is represented by a symbol, and engineers draw diagrams that show the interconnections among gates. Figure 2.6 shows the symbols for *nand*, *nor*, *inverter*, *and*, *or*, and *xor* gates. The figure follows standard terminology by using the term *inverter* for a gate that performs the Boolean *not* operation.

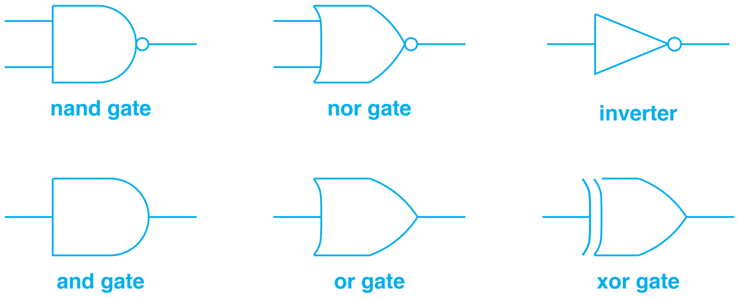


Figure 2.6 The symbols for commonly used gates. Inputs for each gate are shown on the left, and the output of the gate is shown on the right.

2.8 Example Interconnection Of Gates

The electronic parts that implement gates are classified as *Transistor-Transistor Logic* (*TTL*) because the output transistors in each gate are designed to connect directly to input transistors in other gates. In fact, an output can connect to several inputs†. For example, suppose a circuit is needed in which the output is true if a disk is spinning and the user presses a power-down button. Logically, the output is a Boolean *and* of two

†The technology limits the number of inputs that can be connected to a single output; we use the term *fanout* to specify the number of inputs that an output supplies.

inputs. We said, however, that some designs are limited to *nand*, *nor*, and *inverter* gates. In such cases, the *and* function can be created by directly connecting the output of a *nand* gate to the input of an *inverter*. Figure 2.7 illustrates the connection.

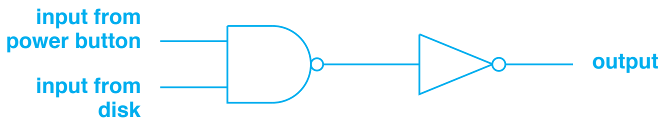


Figure 2.7 Illustration of gate interconnection. The output from one logic gate can connect directly to the input of another gate.

As another example of gate interconnection, consider the circuit in Figure 2.8 that shows three inputs.

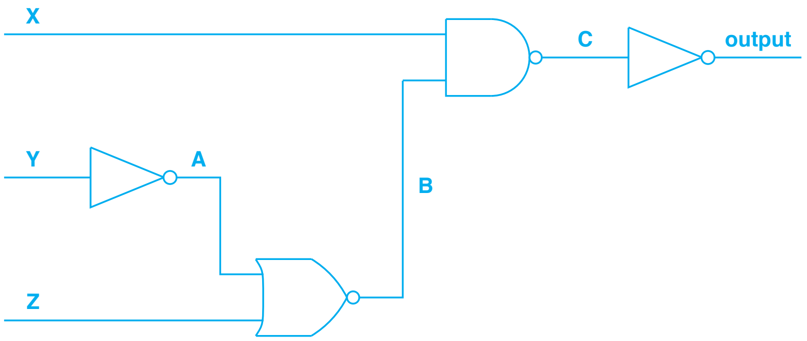


Figure 2.8 An example of a circuit with three inputs labeled *X*, *Y*, and *Z*. Internal interconnections are also labeled to allow us to discuss intermediate values.

What function does the circuit in the figure implement? There are two ways to answer the question: we can determine the Boolean formula to which the circuit corresponds, or we can enumerate the value that appears on each output for all eight possible combinations of input values. To help us understand the two methods, we have labeled each input and each intermediate connection in the circuit as well as the output.

To derive a Boolean formula, observe that input *Y* is connected directly to an inverter. Thus, the value at *A* corresponds to the Boolean function *not Y*. The *nor* gate takes inputs *not Y* (from the inverter) and *Z*, so the value at *B* corresponds to the Boolean function:

Z nor (not Y)

Because the combination of a *nand* gate followed by an inverter produces the Boolean *and* of the two inputs, the output value corresponds to:

X and (Z nor (not Y))

The formula can also be expressed as:

X and not (Z or (not Y)) (2.1)

Although we have described the use of Boolean expressions as a way of understanding circuits, Boolean expressions are also important in circuit design. An engineer can start a design by finding a Boolean expression that describes the desired behavior of a circuit. Writing such an expression can help a designer understand the problem and special cases. Once a correct expression has been found, the engineer can translate the expression into equivalent hardware gates.

The use of Boolean expressions to specify circuits has a significant advantage: a variety of tools are available that operate on Boolean expressions. Tools can be used to analyze an expression, minimize an expression†, and convert an expression into a diagram of interconnected gates. Automated minimization is especially useful because it can reduce the number of gates required. That is, tools exist that can take a Boolean expression as input, produce as output an equivalent expression that requires fewer gates, and then convert the output to a circuit diagram. We can summarize:

> *Tools exist that take a Boolean expression as input and produce an optimized circuit for the expression as output.*

A second technique used to understand a logic circuit consists of enumerating all possible inputs, and then finding the corresponding values at each point in the circuit. For example, because the circuit in Figure 2.8 has three inputs, eight possible combinations of input exist. We use the term *truth table* to describe the enumeration. Truth tables are often used when debugging circuits. Figure 2.9 contains the truth table for the circuit in Figure 2.8. The table lists all possible combination of inputs on wires *X*, *Y*, and *Z* along with the resulting values on the wires labeled *A*, *B*, *C*, and output.

X	Y	Z	A	B	C	output
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	0	1	0	1
1	1	1	0	0	1	0

Figure 2.9 A truth table for the circuit in Figure 2.8.

†Appendix 2 lists a set of rules used to minimize Boolean expressions.

The table in Figure 2.9 is generated by starting with all possible inputs, and then filling in the remaining columns one at a time. In the example, there are three inputs (X, Y, and Z) that can each be set to zero or one. Consequently, there are eight possible combinations of values in columns X, Y, and Z of the table. Once they have been filled in, the input columns can be used to derive other columns. For example, point A in the circuit represents the output from the first inverter, which is the inverse of input Y. Thus, column A can be filled in by reversing the values in column Y. Similarly, column B represents the *nor* of columns A and Z.

A truth table can be used to validate a Boolean expression — the expression can be computed for all possible inputs and compared to the values in the truth table. For example, the truth table in Figure 2.9 can be used to validate the Boolean expression (2.1) above and the equivalent expression:

X and Y and (not Z))

To perform the validation, one computes the value of the Boolean expression for all possible combinations of X, Y, and Z. For each combination, the value of the expression is compared to the value in the output column of the truth table.

2.9 A Digital Circuit For Binary Addition

How can logic circuits implement integer arithmetic? As an example, consider using gates to add two binary numbers. One can apply the technique learned in elementary school: align the two numbers in a column. Then, start with the least-significant digits and add each column of digits. If the sum overflows a given column, carry the high-order digit of the sum to the next column. The only difference is that computers represent integers in binary rather than decimal. For example, Figure 2.10 illustrates the addition of 20 and 29 carried out in binary.

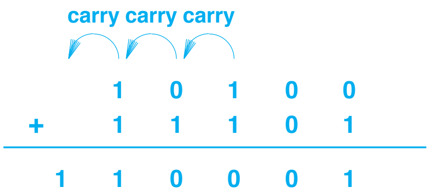


Figure 2.10 Example of binary addition using carry bits.

A circuit to perform the addition needs one module for each column (i.e., each bit in the operands). The module for the low-order bits takes two inputs and produces two outputs: a *sum bit* and a *carry bit*. The circuit, which is known as a *half adder*, contains an *and* gate and an *exclusive or* gate. Figure 2.11 shows how the gates are connected.

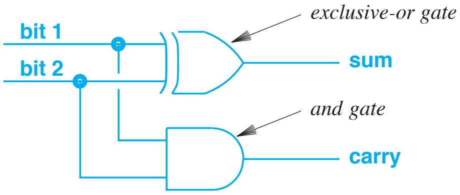


Figure 2.11 A half adder circuit that computes the sum and carry for two input bits.

Although a half adder circuit computes the low-order bit of the sum, a more complex circuit is needed for each of the other bits. In particular, each successive computation has three inputs: two input bits plus a carry bit from the column to the right. Figure 2.12 illustrates the necessary circuit, which is known as a *full adder*. Note the symmetry between the two input bits — either input can be connected to the *sum* of the circuit for the previous bit.

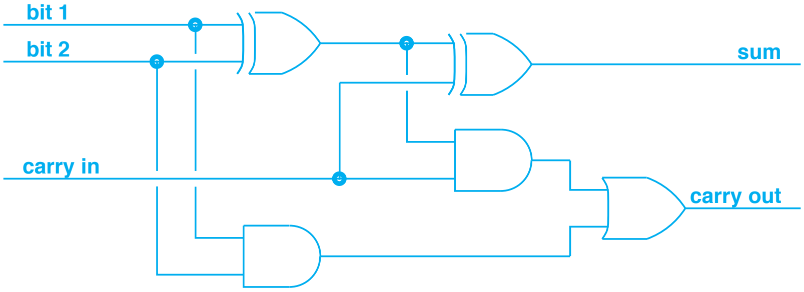


Figure 2.12 A full adder circuit that accepts a carry input as well as two input bits.

As the figure shows, a full adder consists of two half adder circuits plus one extra gate (a logical *or*). The *or* connects the carry outputs from the two half adders, and provides a carry output if either of the two half adders reports a carry.

Although a full adder can have eight possible input combinations, we only need to consider six when verifying correctness. To see why, observe that the full adder treats *bit 1* and *bit 2* symmetrically. Thus, we only need to consider three cases: both input bits are zeros, both input bits are ones, and one of the input bits is one while the other is zero. The presence of a carry input doubles the number of possibilities to six. An exercise suggests using a truth table to verify that the full adder circuit does indeed give correct output for each input combination.

2.10 Multiple Gates Per Integrated Circuit

Because the logic gates described above do not require many transistors, multiple gates that use TTL can be manufactured on a single, inexpensive electronic component. One popular set of TTL components that implement logic gates is known as the *7400 family*†; each component in the family is assigned a part number that begins with *74*. Physically, many of the parts in the 7400 family consist of a rectangular package approximately one-half inch long with fourteen copper wires (called *pins*) that are used to connect the part to a circuit; the result is known as a *14-pin Dual In-line Package* (*14-pin DIP*). More complex 7400-series chips require additional pins (e.g., some use a 16-pin DIP configuration).

To understand how multiple gates are arranged on a 7400-series chip, consider three examples. Part number 7400 contains four *nand* gates, part number 7402 contains four *nor* gates, and part number 7404 contains six inverters. Figure 2.13 illustrates how the inputs and outputs of individual logic gates connect to pins in each case.

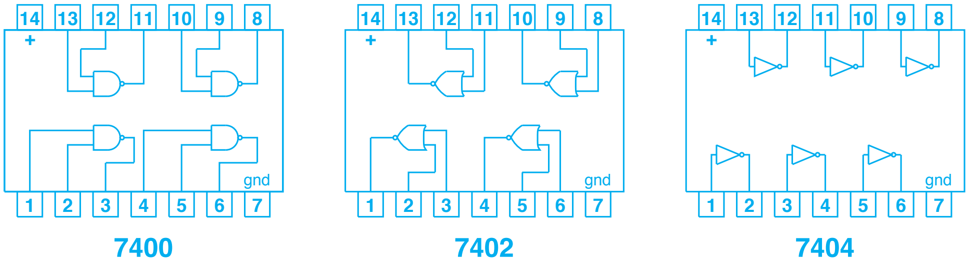


Figure 2.13 Illustration of the pin connections on three commercially available integrated circuits that implement logic gates.

Although the figure does not show gates connected to pins 14 and 7, the two pins are essential because they supply power needed to run the gates — as the labels indicate, pin 14 connects to plus five volts and pin 7 connects to ground (zero volts).

2.11 The Need For More Than Combinatorial Circuits

An interconnection of Boolean logic gates, such as the circuits described above, is known as a *combinatorial circuit* because the output is simply a Boolean combination of input values. In a combinatorial circuit, the output only changes when an input value changes. Although combinatorial circuits are essential, they are not sufficient — a computer requires circuits that can take action without waiting for inputs to change. For example, when a user presses a button to power on a computer, hardware must perform a sequence of operations, and the sequence must proceed without further input from the user. In fact, a user does not need to hold the power button continuously — the startup

†In addition to the logic gates described in this section, the 7400 family also includes more sophisticated mechanisms, such as flip-flops, counters, and demultiplexors, that are described later in the chapter.

sequence continues even after the user releases the button. Furthermore, pressing the same button again causes the hardware to initiate a shutdown sequence.

How can a power button act to power down as well as power up a system? How can digital logic perform a sequence of operations without requiring the input values to change? How can a digital circuit continue to operate after an input reverts to its initial condition? The answers involve additional mechanisms. Sophisticated arrangements of logic gates can provide some of the needed functionality. The rest requires a hardware device known as a *clock*. The next sections present examples of sophisticated circuits, and later sections explain clocks.

2.12 Circuits That Maintain State

In addition to electronic parts that contain basic Boolean gates, parts are also available that contain gates interconnected to maintain *state*. That is, electronic circuits exist in which the outputs are a function of the sequence of previous inputs as well as the current input. Such logic circuits are known as *sequential circuits*.

A *latch* is one of the most basic of sequential circuits. The idea of a latch is straightforward: a latch has an input and an output. In addition, a latch has an extra input called an *enable line*. As long as the enable line is set to logical one, the latch makes its output an exact copy of the input. That is, while the enable line is one, if the input changes, the output changes as well. Once the enable line changes to logical zero, however, the output freezes at its current value and does not change. Thus, the latch "remembers" the value the input had while the enable line was set, and keeps the output set to that value.

How can a latch be devised? Interestingly, a combination of Boolean logic gates is sufficient. Figure 2.14 illustrates a circuit that uses four *nand* gates to create a latch. The idea is that when the enable line is logical zero, the two *nand* gates on the right remember the current value of the output. Because the outputs of two *nand* gates feed back into each other's input, the output value will remain stable. When the enable line is logical one, the two gates on the left pass the data input (on the lower wire) and its inverse (on the higher wire) to the pair of gates on the right.

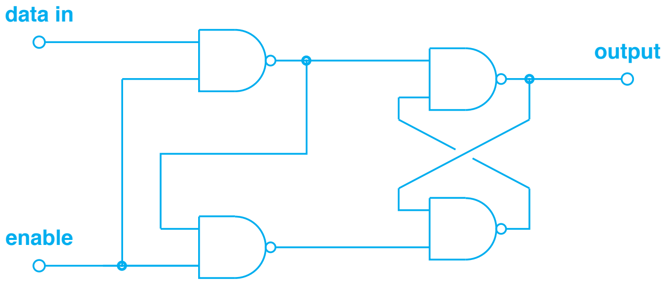


Figure 2.14 Illustration of four *nand* gates used to implement a one-bit latch.

2.13 Propagation Delay

To understand the operation of a latch, one must know that each gate has a *propagation delay*. That is, a delay occurs between the time an input changes and the output changes. During the propagation delay, the output remains at the previous value. Of course, transistors are designed to minimize delay, and the delay can be less than a microsecond, but a finite delay exists. To see how propagation delay affects circuits, consider the circuit in Figure 2.15.

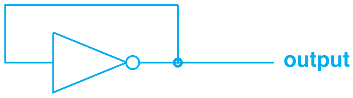


Figure 2.15 An inverter with the output connected back to the input.

As the figure shows, the output of an inverter is connected back to the input. It does not seem that such a connection makes sense because an inverter's output is always the opposite of its input. The Boolean expression for such a circuit is:

$$output = not(output)$$

which is a mathematical contradiction.

Propagation delay explains that the circuit works. At any time, if output is 0, the input to the inverter will be 0. After a propagation delay, the inverter will change the output to 1. Once the output becomes 1, another propagation delay occurs, and the output will become 0 again. Because the cycle goes on forever, we say that the circuit *oscillates* by generating an output that changes back and forth between 0 and 1 (known as a *square wave*). The concept of propagation delay explains the operation of the latch in Figure 2.14 — outputs remain the same until a propagation delay occurs.

2.14 Using Latches To Create A Memory

We will see that processors include a set of *registers* that serve as short-term storage units. Typically, registers hold values that are used in computation (e.g., two values that will be added together). Each register holds multiple bits; most computers have 32-bit or 64-bit registers. The circuit for a register illustrates an important principle of digital hardware design:

> *A circuit to handle multiple bits is constructed by physically replicating a circuit that handles one bit.*

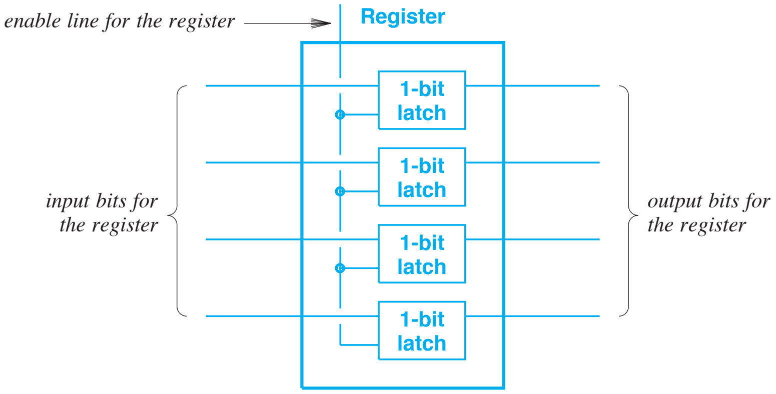

enable line for the register
Register
1-bit latch
1-bit latch
1-bit latch
1-bit latch
input bits for the register
output bits for the register

Figure 2.16 A 4-bit register formed from four 1-bit latches.

To understand the principle, consider Figure 2.16 which shows how a 4-bit register circuit can be constructed from four 1-bit latches†. In the figure, the enable lines of all four latches are connected together to form an enable input for the register. Although the hardware consists of four independent circuits, connecting the enable lines means the four latches act in unison. When the enable line is set to logical one, the register accepts the four input bits and sets the four outputs accordingly. When the enable line becomes zero, the outputs remain fixed. That is, the register has stored whatever value was present on its inputs, and the output value will not change until the enable line becomes one again.

The point is:

> *A register, one of the key components in a processor, is a hardware mechanism that uses a set of latches to store a digital value.*

2.15 Flip-Flops And Transition Diagrams

A *flip-flop* is another circuit in which the output depends on previous inputs as well as the current input. There are various forms. One form acts exactly like the power switch on a computer: the first time its input becomes *1*, the flip-flop turns the output on, and the second time the input becomes *1*, the flip-flop turns the output off. Like a push-button switch used to control power, a flip-flop does not respond to a continuous input — the input must return to *0* before a value of 1 will cause the flip-flop to change state. That is, whenever the input transitions from 0 to 1, the flip-flop changes its output from the current state to the opposite. Figure 2.17 shows a sequence of inputs and the resulting output.

†Although the diagram only shows a 4-bit register, the registers used in typical processors store 32 bits or 64 bits.

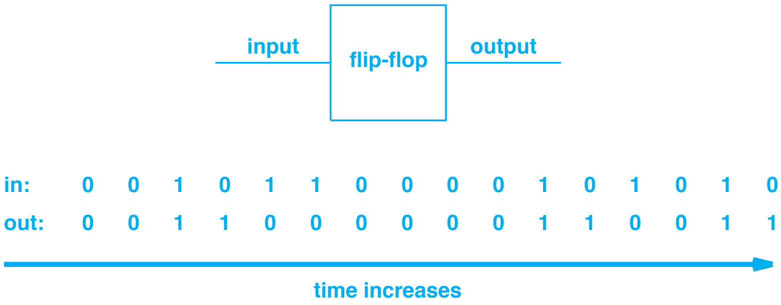

input
flip-flop
output
in: 0 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0
out: 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1
time increases

Figure 2.17 Illustration of how one type of flip-flop reacts to a sequence of inputs. The flip-flop output changes when the input transitions from 0 to 1 (i.e., from zero volts to five volts).

Because it responds to a sequence of inputs, a flip-flop is not a simple combinatorial circuit. A flip-flop cannot be constructed from a single gate. However, a flip-flop can be constructed from a pair of latches.

To understand how a flip-flop works, it is helpful to plot the input and output in graphical form as a function of time. Engineers use the term *transition diagram* for such a plot. In most digital circuits, transitions are coordinated by a clock, which means that transitions only occur at regular intervals. Figure 2.18 illustrates a transition diagram for the flip-flop values from Figure 2.17. The line labeled *clock* in Figure 2.18 shows where clock pulses occur; each input transition is constrained to occur on one of the clock pulses. For now, it is sufficient to understand the general concept; later sections explain clocks.

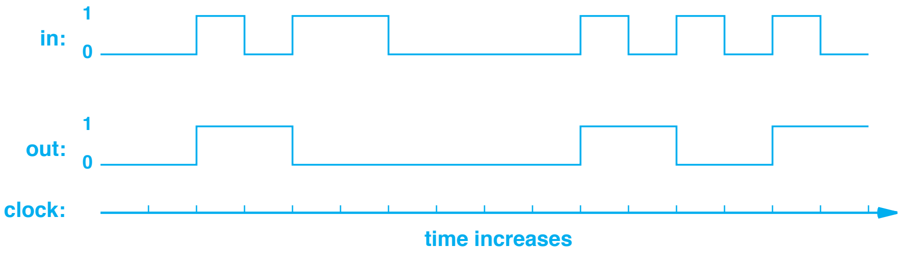


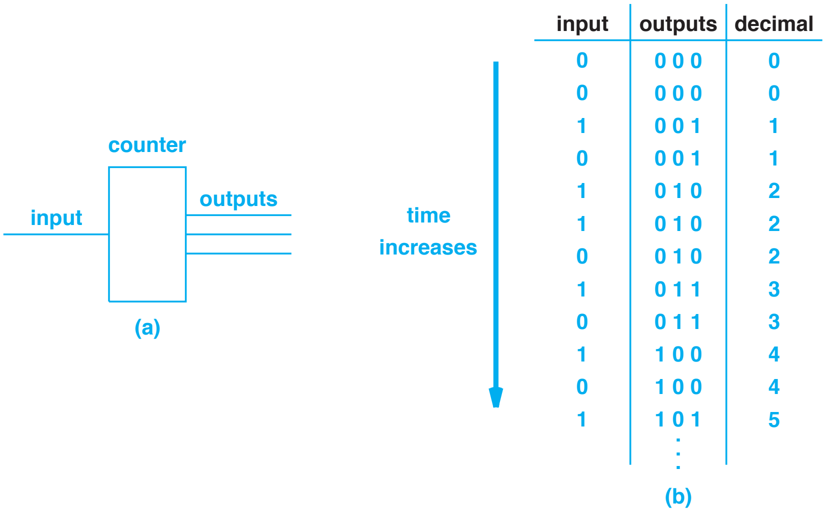
Figure 2.18 Illustration of a transition diagram that shows how a flip flop reacts to the series of inputs in Figure 2.17. Marks along the x-axis indicate times; each corresponds to one clock tick.

We said that a flip-flop changes output each time it encounters a one bit. In fact, the transition diagram shows the exact details and timing that are important to circuit designers. In the example, the transition diagram shows that the flip-flop is only triggered when the input *rises*. That is, the output does not change until the input transitions from zero to one. Engineers say that the output transition occurs on the *rising edge* of the input change; circuits that transition when the input changes from one to zero are said to occur on the *falling edge*.

In practice, additional details complicate flip-flops. For example, most flip-flops include an additional input named *reset* that places the output in a 0 state. In addition, several other variants of flip-flops exist. For example, some flip-flops provide a second output that is the inverse of the main output (in some circuits, having the inverse available results in fewer gates).

2.16 Binary Counters

A single flip-flop only offers two possible output values: 0 or 1. However, a set of flip-flops can be connected in series to form a binary *counter* that accumulates a numeric total. Like a flip-flop, a counter has a single input. Unlike a flip-flop, however, a counter has multiple outputs. The outputs count how many input pulses have been detected by giving a numerical total in binary†. We think of the outputs as starting at zero and adding one each time the input transitions from 0 to 1. Thus, a counter that has three output lines can accumulate a total between 0 and 7. Figure 2.19 illustrates a counter, and shows how the outputs change when the input changes.



input	outputs	decimal
0	0 0 0	0
0	0 0 0	0
1	0 0 1	1
0	0 0 1	1
1	0 1 0	2
1	0 1 0	2
0	0 1 0	2
1	0 1 1	3
0	0 1 1	3
1	1 0 0	4
0	1 0 0	4
1	1 0 1	5
	⋮	

Figure 2.19 Illustration of (a) a binary counter, and (b) a sequence of input values and the corresponding outputs. The column labeled *decimal* gives the decimal equivalent of the outputs.

†Chapter 3 considers data representation in more detail. For now, it is sufficient to understand that the outputs represent a number.

In practice, an electronic part that implements a binary counter has several additional features. For example, a counter has an additional input used to reset the count to zero, and may also have an input that temporarily stops the counter (i.e., ignores the input and freezes the output). More important, because it has a fixed number of output pins, each counter has a maximum value it can represent. When the accumulated count exceeds the maximum value, the counter resets the output to zero and uses an additional output to indicate that an *overflow* occurred.

2.17 Clocks And Sequences

Although we have seen the basic building blocks of digital logic, one additional feature is absolutely essential for a digital computer: automatic operation. That is, a computer must be able to execute a sequence of instructions without any inputs changing. The digital logic circuits discussed previously all use the property that they respond to changes in one or more of their inputs; they do not perform any function until an input changes. How can a digital logic circuit perform a series of steps?

The answer is a mechanism known as a *clock* that allows hardware to take action without requiring the input to change. In fact, most digital logic circuits are said to be *clocked*, which means that the clock signal, rather than changes in the inputs, controls and synchronizes the operation of individual components and subassemblies to ensure that they work together as intended (e.g., to guarantee that later stages of a circuit wait for the propagation delay of previous stages).

What is a clock? Unlike the common definition of the term, hardware engineers use the term *clock* to refer to an electronic circuit that oscillates at a regular rate; the oscillations are converted to a sequence of alternating ones and zeros. Although a clock can be created from an inverter†, most clock circuits use a quartz crystal, which oscillates naturally, to provide a signal at a precise frequency. The clock circuit amplifies the signal and changes it from a sine wave to a square wave. Thus, we think of a clock as emitting an alternating sequence of 0 and 1 values at a regular rate. The speed of a clock is measured in *Hertz* (*Hz*), the number of times per second the clock cycles through a 1 followed by a 0. Most clocks in high-speed digital computers operate at speeds ranging from one hundred megahertz (100 MHz) to several gigahertz (GHz). For example, at present, the clock used by a typical processor operates at approximately 3 GHz.

It is difficult for a human to imagine circuits changing at such high rates. To make the concept clear, let's consider a clock is available that operates at an extremely slow rate of 1 Hz. Such a clock might be used to control an interface for a human. For example, if a computer contains an LED that flashes on and off to indicate that the computer is active, a slow clock is needed to control the LED. Note that a clock rate of 1 Hz means the clock completes an entire cycle in one second. That is, the clock emits a logical 1 for one-half cycle followed by a logical zero for one-half cycle. If a circuit arranges to turn on an LED whenever the clock emits a logical 1, the LED will remain on for one-half second, and then will be off for one-half second.

†See Figure 2.15 on page 24.

How does an alternating sequence of 0 and 1 values make digital circuits more powerful? To understand, we will consider a simple clocked circuit. Suppose that during startup, a computer must perform the following sequence of steps:

- Test the battery
- Power on and test the memory
- Start the disk spinning
- Power up the screen
- Read the boot sector from disk into memory
- Start the CPU

To simplify the explanation, we will assume that each step requires at most one second to complete before the next step can be started. Thus, we desire a circuit that, once it has been started, will perform the six steps in sequence, at one-second intervals with no further changes in input.

For now, we will focus on the essence of the circuit, and consider how it can be started later. A circuit to handle the task of performing six steps in sequence can be built from three building blocks: a clock, a binary counter, and a device known as a *decoder/demultiplexor*†, which is often abbreviated *demux*. We have already considered a counter, and will assume that a clock is available that generates digital output at a rate of exactly one cycle per second. The last component, a decoder/demultiplexor, is a single integrated circuit that uses a binary value to map an input to a set of outputs. We will use the decoding function to select an output. That is, a decoder takes a binary value as input, and uses the value to choose an output. Only one output of a decoder is on at any time; all others are off — when the input lines represent the value *i* in binary, the decoder selects the i^{th} output. Figure 2.20 illustrates the concept.

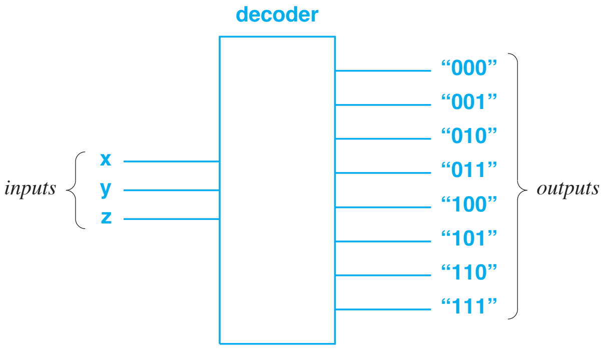


Figure 2.20 Illustration of a decoder with three input lines and eight output lines. When inputs x, y, and z have the values 0, 1, and 1, the fourth output from the top is selected.

†An alternate spelling of *demultiplexer* is also used.

When used as a decoder, the device merely selects one of its outputs; when used as a demultiplexor, the device takes an extra input which it passes to the selected output. Both the decoder function and the more complex demultiplexor function can be constructed from Boolean gates.

A decoder provides the last piece needed for our simplistic sequencing mechanism — when we combine a clock, counter, and decoder, the resulting circuit can execute a series of steps. For example, Figure 2.21 shows the interconnection in which the output of a clock is used as input to a binary counter, and the output of a binary counter is used as input to a decoder.

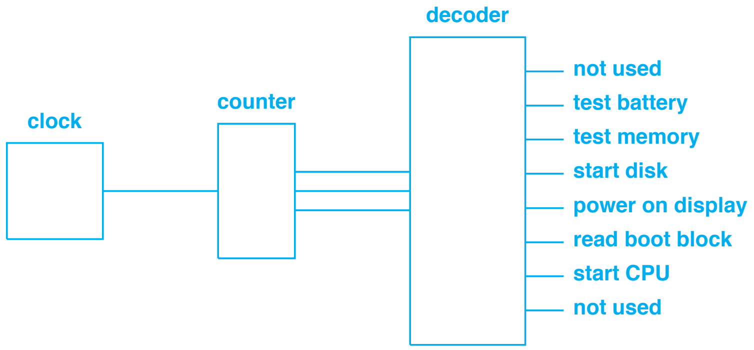


Figure 2.21 An illustration of how a clock can be used to create a circuit that performs a sequence of six steps. Output lines from the counter connect directly to input lines of the decoder.

To understand how the circuit operates, assume that the counter has been reset to zero. Because the counter output is 000, the decoder selects the topmost output, which is not used (i.e., not connected). Operation starts when the clock changes from logical 0 to logical 1. The counter accumulates the count, which changes its output to 001. When its input changes, the decoder selects the second output, which is labeled *test battery*. Presumably, the second output wire connects to a circuit that performs the necessary test. The second output remains selected for one second. During the second, the clock output remains at logical 1 for one-half second, and then reverts to logical 0 for one-half second. When the clock output changes back to logical 1, the counter output lines change to 010, and the decoder selects the third output, which is connected to circuitry that tests memory.

Of course, details are important. For example, some decoder chips make a selected output 0 and other outputs 1. Electrical details also matter. To be compatible with other devices, the clock must use five volts for logical 1, and zero volts for logical 0. Furthermore, to be directly connected, the output lines of the binary counter must use the same binary representation as the input lines of the decoder. Chapter 3 discusses data representation in more detail; for now, we assume the output and input values are compatible.

2.18 The Important Concept Of Feedback

The simplistic circuit in Figure 2.21 lacks an important feature: there is no way to control operation (i.e., to start or stop the sequence). Because a clock runs forever, the counter in the figure counts from zero through its maximum value, and then starts again at zero. As a result, the decoder will repeatedly cycle through its outputs, with each output being held for one second before moving on to the next.

Few digital circuits perform the same series of steps repeatedly. How can we arrange to stop the sequence after the six steps have been executed? The solution lies in a fundamental concept: *feedback*. Feedback lies at the heart of complex digital circuits because it allows the results of processing to affect the way a circuit behaves. In the computer startup sequence, feedback is needed for each of the steps. If the disk cannot be started, for example, the boot sector cannot be read from the disk.

We have already seen feedback used to maintain a data value in the latch circuit of Figure 2.14 because the output from each of the right-most *nand* gates feeds back as an input to the other gate. For another example of feedback, consider how we might use the final output of the decoder, call it *F*, to stop the sequence. An easy solution consists of using the value of F to prevent clock pulses from reaching the counter. That is, instead of connecting the clock output directly to the counter input, we insert logic gates that only allow the counter input to continue when F has the value 0. In terms of Boolean algebra, the counter input should be:

CLOCK *and* (*not* F)

That is, as long as F is false, the counter input should be equal to the clock; when F is true, however, the counter input changes to (and remains) zero. Figure 2.22 shows how two inverters and a *nand* gate can be used to implement the necessary function.

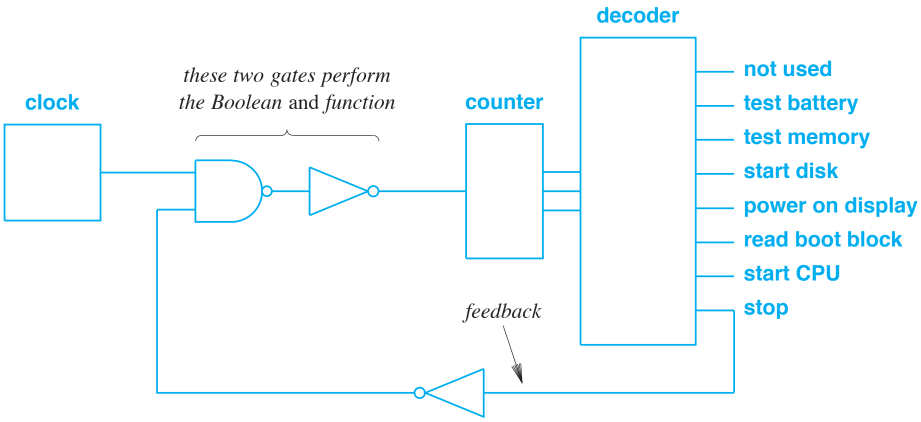


Figure 2.22 A modification of the circuit in Figure 2.21 that includes feedback to stop processing after one pass through each output.

The feedback in Figure 2.22 is fairly obvious because there is an explicit physical connection between the last output and the combinatorial circuit on the input side. The figure also makes it easy to see why feedback mechanisms are sometimes called *feedback loops*†.

2.19 Starting A Sequence

Figure 2.22 shows that it is possible to use feedback to terminate a process. However, the circuit is still incomplete because it does not contain a mechanism that allows the sequence to start. Fortunately, adding a starting mechanism is trivial. To understand why, recall that a counter contains a separate input line that resets the count to zero. All that is needed to make our circuit start is another input (e.g., from a button that a user pushes) connected to the counter reset.

When a user pushes the button, the counter resets to zero, which causes the counter's output to become 000. When it receives an input of all zeros, the decoder turns on the first output, and turns off the last output. When the last output turns off, the *nand* gate allows the clock pulses through, and the counter begins to run.

Although it does indeed start the sequence, allowing a user to reset the counter can cause problems. For example, consider what happens if a user becomes impatient during the startup sequence and presses the button a second time. Once the counter resets, the sequence starts again from the beginning. In some cases, performing an operation twice simply wastes time. In other cases, however, repeating an operation causes problems (e.g., some disk drives require that only one command be issued at a time). Thus, a production system uses complex combinatorial logic to prevent a sequence from being interrupted or restarted before it completes.

Although it only contains a few components, the example demonstrates an important concept: a set of Boolean logic gates and a clock are sufficient to allow the execution of a sequence of logical steps. The point is:

> *The example circuit shows that Boolean logic gates and a clock make it possible to build a circuit which, when started, performs a logical sequence of steps and then halts.*

Only one additional concept is needed before we can create a general-purpose computer: programmability. Chapter 6 extends our discussion of hardware by showing how the basic components described here can be used to build a *programmable* processor that uses a program in memory to determine the sequence of operations.

†A feedback loop is also present among the gates used to construct a flip-flop.

2.20 Iteration In Software Vs. Replication In Hardware

As we think about hardware, it will be important to remember a significant difference between the way software and hardware handle operations that must be applied to a set of items. In software, a fundamental paradigm for handling multiple items consists of *iteration* — a programmer writes code that repeatedly finds the next item in a set and applies the operation to the item. Because the underlying system only applies the operation to one item at a time, a programmer must specify the number of items. Iteration is so essential to programming that most programming languages provide statements (e.g., a *for loop*) that allow the programmer to express the iteration clearly.

Although hardware can be built to perform iteration, doing so is difficult and the resulting hardware is clumsy. Instead, the fundamental hardware paradigm for handling multiple items consists of *replication* — a hardware engineer creates multiple copies of a circuit, and allows each copy to act on one item. All copies perform at the same time. For example, to compute a Boolean operation on a pair of thirty-two bit values, a hardware engineer designs a circuit for a pair of bits, and then replicates the circuit thirty-two times. Thus, to compute the Boolean *exclusive or* of two thirty-two bit integers, a hardware designer can use thirty-two *xor* gates.

Replication is difficult for programmers to appreciate because replication is antithetical to good programming — a programmer is taught to avoid duplicating code. In the hardware world, however, replication has three distinct advantages: elegance, speed, and correctness. Elegance arises because replication avoids the extra hardware needed to select an individual item, move it into place, and move the result back. In addition to avoiding the delay involved in moving values and results, replication increases performance by allowing multiple operations to be performed simultaneously. For example, thirty-two inverters working at the same time can invert thirty-two bits in exactly the same amount of time that it takes one inverter to invert a single bit. Such speedup is especially significant if a computer can operate on sixty-four bits at the same time. The notion of parallel operation appears throughout the text; a later chapter explains how parallelism applies on a larger scale.

The third advantage of replication focuses on high reliability. Reliability is increased because replication makes hardware easier to validate. For example, to validate that a thirty-two bit operation works correctly, a hardware engineer only needs to validate the circuit for a single bit — the remaining bits will work the same because the same circuit is replicated. As a result, hardware is much more reliable than software. Even the legal system holds product liability standards higher for hardware than for software — unlike software that is often sold “as is” without a warranty, hardware (e.g., an integrated circuit) is sold within a legal framework that requires fitness for the intended purpose. We can summarize:

> *Unlike software, which uses iteration, hardware uses replication. The advantages of replication are increased elegance, higher speed, and increased reliability.*

2.21 Gate And Chip Minimization

We have glossed over many of the underlying engineering details. For example, once they choose a general design and the amount of replication that will be used, engineers seek ways to minimize the amount of hardware needed. There are two issues: minimizing gates and minimizing integrated circuits. The first issue involves general rules of Boolean algebra. For example, consider the Boolean expression:

not (*not* z)

A circuit to implement the expression consists of two inverters connected together. Of course, we know that two *not* operations are the identity function, so the expression can be replaced by z. That is, a pair of directly connected inverters can be removed from a circuit without affecting the result.

As another example of Boolean expression optimization, consider the expression:

x *nor* (*not* x)

Either *x* will have the value 1, or *not x* will have the value 1, which means the *nor* function will always produce the same value, a logical 0. Therefore, the entire expression can be replaced by the value 0. In terms of a circuit, it would be foolish to use a *nor* gate and an inverter to compute the expression because the circuit resulting from the two gates will always be logical zero. Thus, once an engineer writes a Boolean expression formula, the formula can be analyzed to look for instances of subexpressions that can be reduced or eliminated without changing the result.

Fortunately, sophisticated design tools exist that help engineers minimize gates. Such tools take a Boolean expression as an input. The design tool analyzes the expression and produces a circuit that implements the expression with a minimum number of gates. The tools do not merely use Boolean *and*, *or*, and *not*. Instead, they understand the gates that are available (e.g., *nand*), and define the circuit in terms of available electronic parts.

Although Boolean formulas can be optimized mathematically, further optimization is needed because the overall goal is minimization of integrated circuits. To understand the situation, recall that many integrated circuits contain multiple copies of a given type of gate. Thus, minimizing the number of Boolean operations may not optimize a circuit if the optimization increases the types of gates required. For example, suppose a Boolean expression requires four *nand* gates, and consider an optimization that reduces the requirements to three gates: two *nand* gates and a *nor* gate. Unfortunately, although the total number of gates is lower, the optimization increases the number of integrated circuits required because a single 7400 integrated circuit contains four *nand* gates, but two integrated circuits are required if an optimization includes both *nand* and *nor* gates.

2.22 Using Spare Gates

Consider the circuit in Figure 2.22 carefully†. Assuming the clock, counter, and decoder each require one integrated circuit, how many additional integrated circuits are required? The obvious answer is two: one is needed for the *nand* gate (e.g., a 7400)

†Figure 2.22 can be found on page 31.

and another for the two inverters (e.g., a 7404). Surprisingly, it is possible to implement the circuit with only one additional integrated circuit. To see how, observe that although the 7400 contains four *nand* gates, only one is needed. How can the spare gates be used? The trick lies in observing that *nand* of 1 and 0 is 1, and *nand* of 1 and 1 is 0. That is,

1 *nand* x

is equivalent to:

not x

To use a *nand* gate as an inverter, an engineer simply connects one of the two inputs to logical one (i.e., five volts). A spare *nand* gate can be used as an inverter.

2.23 Power Distribution And Heat Dissipation

In addition to planning digital circuits that correctly perform the intended function and minimizing the number of components used, engineers must contend with the underlying power and cooling requirements†. For example, although the diagrams in this chapter only depict the logical inputs and outputs of gates, every gate consumes power. The amount of power used by a single integrated circuit is insignificant. However, because hardware designers tend to use replication instead of iteration, complex digital systems contain many circuits. An engineer must calculate the total power required, construct the appropriate power supplies, and plan additional wiring to carry power to each chip.

The laws of physics dictate that any device that consumes power will generate heat. The amount of heat generated is proportional to the amount of power consumed, so an integrated circuit generates a minimal amount of heat. Because a digital system uses hundreds of circuits that operate in a small, enclosed space, the total heat generated can be significant. Unless engineers plan a mechanism to dissipate heat, high temperatures will cause the circuits to fail. For small systems, engineers add holes to the chassis that allow hot air to escape and be replaced by cooler air from the surrounding room. For intermediate systems, such as personal computers, fans are added to move air from the surrounding room through the system more quickly. For the largest digital systems, cool air is insufficient — a refrigeration system with liquid coolant must be used (e.g., circuits in the Cray 2 supercomputer were directly immersed in a liquid coolant).

2.24 Timing And Clock Zones

Our quick tour of digital logic omits another important aspect that engineers must consider: *timing*. A gate does not act instantly. Instead, a gate takes time to *settle* (i.e., to change the output once the input changes). In our examples, timing is irrelevant be-

†Chapter 20 considers power in more detail.

cause the clock runs at the incredibly slow rate of 1 Hz and all gates settle in less than a microsecond. Thus, the gates settle long before the clock pulses.

In practice, timing is an essential aspect of engineering because digital circuits are designed to operate at high speed. To ensure that a circuit will operate correctly, an engineer must calculate the time required for all gates to settle.

Engineers must also calculate the time required to propagate signals throughout an entire system, and must ensure that the system does not fail because of *clock skew*. To understand clock skew, consider Figure 2.23 that illustrates a circuit board with a clock that controls three of the integrated circuits in the system.

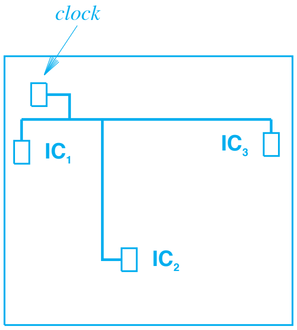


Figure 2.23 Illustration of three integrated circuits in a digital system that are controlled by a single clock. The length of wire between the clock and an integrated circuit determines when a clock signal arrives.

In the figure, the three integrated circuits are physically distributed (presumably, other integrated circuits occupy the remaining space). Unfortunately, a finite time is required for a signal from the clock to reach each of the circuits, and the time is proportional to the length of wire between the clock and a given circuit. As a result, the clock signal will arrive at some of the integrated circuits sooner than it arrives at others. As a rule of thumb, a signal requires one nanosecond to propagate across one foot of wire. Thus, for a system that measures eighteen inches across, the clock signal can reach locations near the clock a nanosecond before the signal reaches the farthest location. Obviously, clock skew can cause a problem if parts of the system must operate before other parts. An engineer needs to calculate the length of each path and design a layout that avoids the problem of clock skew.

As a consequence of clock skew, engineers seldom use a single global clock to control a system. Instead, multiple clocks are used, with each clock controlling one part of the system. Often, clocks that run at the highest rates are used in the smallest physical areas. We use the term *clock zone* to refer to the region that a given clock controls. The idea is not limited to physically large systems — integrated circuits, such as CPUs, have become so large and complex that multiple clock zones are used on a chip.

Although using multiple clock zones avoids the problems of clock skew, multiple clocks introduce another problem, *clock synchronization*: digital logic at the boundary between two clock zones must be engineered to accommodate both zones. Usually, such accommodation means the circuit slows down and takes multiple clock cycles to move data.

2.25 Clockless Logic

As chips increase in size and complexity, the problem of clock skew and the division of a system into clock zones has become increasingly important. In many systems, the boundary between clock zones forms a bottleneck because logic circuits at the boundary must wait multiple clock cycles before the output from one clock zone can be forwarded to another clock zone. The problem of zone synchronization has become so important that researchers have devised an alternative approach: *clockless logic*. In essence, a clockless system uses two wires instead of one to represent a Boolean value. The use of two wires means that an output can indicate the end of a bit unambiguously without depending on a clock. Figure 2.24 lists the four possible combinations of values on two wires and their meanings.

Wire 1	Wire 2	Meaning
0	0	Reset before starting a new bit
0	1	Transfer a 0 bit
1	0	Transfer a 1 bit
1	1	Undefined (not used)

Figure 2.24 Meaning of signals on two wires when clockless logic is used to transfer bits from one chip to another.

The idea is that the sender sets both wires to zero volts between each bit to *reset* the receiver. After the reset, the sender transfers a logical 0 or a logical 1. A receiver knows when a bit arrives because exactly one of the two wires is high (e.g., 5 volts).

Why use clockless logic? In addition to eliminating the problem of clock zone coordination and allowing higher speed data transfer among chips, the clockless approach can use less power. Clocked circuits need to propagate the clock signal continuously, even when parts of the circuit are inactive. Clockless logic can avoid the overhead of propagating clock signals.

Does the clockless approach work in practice? Yes. By designing an entire processor that uses clockless logic, ARM, Inc. has demonstrated that the approach scales to large, complex circuits. Thus, the clockless approach has potential. Currently, most chip designers still use the clocked approach.

2.26 Circuit Size And Moore's Law

Most digital circuits are built from *Integrated Circuits* (*ICs*), a technology that permits many transistors to be placed on a single silicon chip along with wiring that interconnects them. The idea is that the components on an IC form a useful circuit.

ICs are often created by using *Complementary Metal Oxide Semiconductor* (*CMOS*) technology. Silicon is doped with impurities to give it negative or positive ionization. The resulting substances are known as *N-type silicon* or *P-type silicon*. When arranged in layers, N-type and P-type silicon form transistors.

IC manufacturers do not create a single IC at a time. Instead, a manufacturer creates a round *wafer* that is between twelve and eighteen inches in diameter and contains many copies of a given IC design. Once the wafer has been created, the vendor cuts out the individual chips, and packages each chip in a plastic case along with pins that connect to the chip.

ICs come in a variety of shapes and sizes; some have only eight external connections (i.e., *pins*), and others have hundreds of pins†. Some ICs contain dozens of transistors, others contain millions.

Depending on the number of transistors on the chip, ICs can be divided into four broad categories that Figure 2.25 lists.

Name	Example Use
Small Scale Integration (SSI)	Basic Boolean gates
Medium Scale Integration (MSI)	Intermediate logic, such as counters
Large Scale Integration (LSI)	Small, embedded processors
Very Large Scale Integration (VLSI)	Complex processors

Figure 2.25 A classification scheme used for integrated circuits.

For example, integrated 7400, 7402, and 7404 circuits described in this chapter are classified as SSI. A binary counter, flip-flop, or demultiplexor is classified as MSI.

The definition of VLSI keeps changing as manufacturers devise new ways to increase the density of transistors per square area. Gordon Moore, a cofounder of Intel Corporation, is attributed with having observed that the density of silicon circuits, measured in the number of transistors per square inch, would double every year. The observation, known as *Moore's Law*, was revised in the 1970s, when the rate slowed to doubling every eighteen months.

As the number of transistors on a single chip increased, vendors took advantage of the capability to add more and more functionality. Some vendors created *multicore* CPU chips by placing multiple copies of their CPU (called a *core*) on a single chip, and then providing interconnections among the cores. Other vendors took a *System on Chip* (*SoC*) approach in which a single chip contains processors, memories, and interfaces for I/O devices, all interconnected to form a complete system. Finally, memory manufac-

†Engineers use the term *pinout* to describe the purpose of each pin on a chip.

turers have created chips with larger and larger amounts of main memory called *Dynamic Ram* (*DRAM*).

In addition to general-purpose ICs that are designed and sold by vendors, it has become possible to build special-purpose ICs. Known as *Application Specific Integrated Circuits* (*ASICs*), the ICs are designed by a private company, and then the designs are sent to a vendor to be manufactured. Although designing an ASIC is expensive and time-consuming — approximately two million dollars and nearly two years — once the design is completed, copies of the ASIC are inexpensive to produce. Thus, companies choose ASIC designs for products where standard chips do not meet the requirements and the company expects a large volume of the product to be produced.

2.27 Circuit Boards And Layers

Most digital systems are built using a *Printed Circuit Board* (*PCB*) that consists of a fiberglass board with thin metal strips attached to the surface and holes for mounting integrated circuits and other components. In essence, the metal strips on the circuit board form the wiring that interconnects components.

Can a circuit board be used for complex interconnections that require wires to cross? Interestingly, engineers have developed *multilayer* circuit boards that solve the problem. In essence, a multilayer circuit board allows wiring in three dimensions — when a wire must cross another, the designer can arrange to pass the wire up to a higher layer, make the crossing, and then pass the wire back down.

It may seem that a few layers will suffice for any circuit. However, large complex circuits with thousands of interconnections may need additional layers. It is not uncommon for engineers to design circuit boards that have eighteen layers; the most advanced boards can have twenty-four layers.

2.28 Levels Of Abstraction

As this chapter illustrates, it is possible to view digital logic at various levels of abstraction. At the lowest level, a transistor is created from silicon. At the next level, multiple transistors are used along with components, such as resistors and diodes, to form gates. At the next level, multiple gates are combined to form intermediate scale units, such as flip flops. In later chapters, we will discuss more complex mechanisms, such as processors, memory systems, and I/O devices, that are each constructed from multiple intermediate scale units. Figure 2.26 summarizes the levels of abstraction.

The important point is that moving up the levels of abstraction allows us to hide more details and talk about larger and larger building blocks without giving internal details. When we describe processors, for example, we can consider how a processor works without examining the internal structure at the level of gates or transistors.

Abstraction	Implemented With
Computer	Circuit board(s)
Circuit board	Processor, memory, and bus adapter chips
Processor	VLSI chip
VLSI chip	Many gates
Gate	Many transistors
Transistor	Semiconductor implemented in silicon

Figure 2.26 An example of levels of abstraction in digital logic. An item at one level is implemented using items at the next lower level.

An important consequence of abstraction arises in the diagrams architects and engineers use to describe digital systems. As we have seen, schematic diagrams can represent the interconnection of transistors, resistors, and diodes. Diagrams can also be used to represent an interconnection among gates. In later chapters, we will use high-level diagrams that represent the interconnection of processors and memory systems. In such diagrams, a small rectangular box will represent a processor or a memory without showing the interconnection of gates. When looking at an architectural diagram, it will be important to understand the level of abstraction and to remember that a single item in a high-level diagram can correspond to an arbitrarily large number of items at a lower-level abstraction.

2.29 Summary

Digital logic refers to the pieces of hardware used to construct digital systems such as computers. As we have seen, Boolean algebra is an important tool in digital circuit design — there is a direct relationship between Boolean functions and the gates used to implement combinatorial digital circuits. We have also seen that Boolean logic values can be described using truth tables.

A clock is a mechanism that emits pulses at regular intervals to form a signal of alternating ones and zeros. A clock allows a digital circuit output to be a function of time as well as of its logic inputs. A clock can also be used to provide synchronization among multiple parts of a circuit.

Although we think of digital logic from a mathematical point of view, building practical circuits involves understanding the underlying hardware details. In particular, besides basic correctness, engineers must contend with problems of power distribution, heat dissipation, and clock skew.

EXERCISES

2.1 Use the Web to find the number of transistors on a VLSI chip and the physical size of the chip. If the entire die was used, how large would an individual transistor be?

2.2 Digital logic circuits used in smart phones and other battery-powered devices do not run on five volts. Look at the battery in your smart phone or search the Web to find out what voltage is being used.

2.3 Design a circuit that uses *nand*, *nor* and *inverter* gates to provide the *exclusive or* function.

2.4 Write a truth table for the full adder circuit in Figure 2.12.

2.5 Use the Web to read about flip-flops. List the major types and their characteristics.

2.6 Create the circuit for a decoder from *nand*, *nor*, and *inverter* gates.

2.7 Look at Web sources, such as Wikipedia, to answer the following question: when a chip manufacturer boasts that it uses a seven nanometer chip technology, what does the manufacturer mean?

2.8 What is the maximum number of output bits a counter chip can have if the chip has sixteen pins? (Hint: the chip needs power and ground connections.)

2.9 If a decoder chip has five input pins (not counting power and ground), how many output pins will it have?

2.10 Design a circuit that takes three inputs, A, B, and C, and generates three outputs. The circuit would be trivial, except that you may only use two inverters. You may use arbitrary other chips (e.g., *nand*, *nor*, and *exclusive or*).

2.11 Assume a circuit has a spare *nor* gate. Can any useful functions be created by connecting one of the inputs to logical one? To logical zero? Explain.

2.12 Read about clockless logic. Where is it being used?

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

3

Data And Program Representation

3.1 Introduction

The previous chapter introduces digital logic, and describes basic hardware building blocks that are used to create digital systems. This chapter continues the discussion of fundamentals by explaining how digital systems use binary representations to encode programs and data. We will see that representation is important for programmers as well as for hardware engineers because software must understand the format that the underlying hardware uses, and the format affects the speed with which the hardware can perform operations, such as addition.

3.2 Digital Logic And The Importance Of Abstraction

As we have seen, digital logic circuits contain many low-level details. The circuits use transistors and electrical voltage to perform basic operations. The main point of digital logic, however, is *abstraction* — we want to hide the underlying details and use high-level abstractions whenever possible. For example, we have seen that each input or output of a 7400-series digital logic chip is restricted to two possible conditions: zero volts or five volts. When computer architects use logic gates to design computers, however, they do not think about such details. Instead, they use abstract designations of *logical 0* and *logical 1* from Boolean algebra. Abstracting means that complex digital systems, such as memories and processors, can be described without thinking about individual transistors or voltages. More important, abstraction means that a design can be used with a battery-operated device, such as a smart phone, that uses lower voltages to reduce power consumption.

To a programmer, the most important abstractions are the items visible to software: the representations used for data and programs. The next sections consider data representation, and discuss how it is visible to programs; later sections describe how instructions are represented.

3.3 Definitions Of Bit And Byte

All data representation builds on digital logic. We use the abstraction *binary digit* (*bit*) to describe a digital entity that can have two possible values, and assign the mathematical names *0* and *1* for the two values.

Multiple bits are used to represent more complex data items. For example, each computer system defines a *byte* to be the smallest data item larger than a bit that the hardware can manipulate.

How big is a byte? The size of a byte is not standard across all computer systems. Instead, the size is chosen by the architect who designs the computer. Early computer designers experimented with a variety of byte sizes, and some special-purpose computers still use unusual byte sizes. For example, an early computer manufactured by CDC corporation used a six-bit byte, and a computer manufactured by BB&N used a ten-bit byte. However, most modern computer systems define a byte to contain eight bits — the size has become so widely accepted that engineers usually assume a byte size equal to eight bits, unless told otherwise. The point is:

> *Although computers have been designed with other size bytes, current computer industry practice defines a byte to contain eight bits.*

3.4 Byte Size And Possible Values

The number of bits per byte is especially important to programmers because memory is organized as a sequence of bytes. The size of the byte determines the maximum numerical value that can be stored in one byte. A byte that contains k bits can represent one of 2^k values (i.e., exactly 2^k unique strings of 1s and 0s exist that have length k). Thus, a six-bit byte can represent 64 possible values, and an eight-bit byte can represent 256 possible values. As an example, consider the eight possible combinations that can be achieved with three bits. Figure 3.1 illustrates the combinations.

000	010	100	110
001	011	101	111

Figure 3.1 The eight unique combinations that can be assigned to three bits.

What does a given pattern of bits represent? The most important thing to understand is that the bits themselves have no intrinsic meaning — the interpretation of the value is determined by the way hardware and software use the bits. For example, a string of bits could represent an alphabetic character, a string of characters, an integer, a floating point number, an audio recording (e.g., a song), a video, or a computer program.

In addition to items a computer programmer understands, computer hardware can be designed in which a set of bits can represent the status of three peripheral devices. For example:

- The first bit has the value 1 if a keyboard is connected.
- The second bit has the value 1 if a camera is connected.
- The third bit has the value 1 if a printer is connected.

Alternatively, hardware can be designed in which a set of three bits represent the current status of three pushbutton switches: the *i*th bit is 1 if a user is currently pushing switch *i*. The point is:

> *Bits have no intrinsic meaning — all meaning is imposed by the way bits are interpreted.*

3.5 Binary Weighted Positional Representation

One of the most common abstractions used to associate a meaning with each combination of bits interprets them as a numeric value. For example, an integer interpretation is taken from mathematics: bits are values in a positional number system that uses base two. To understand the interpretation, remember that in base ten, the possible digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, each position represents a power of 10, and the number 123 represents 1 times 10^2 plus 2 times 10^1 plus 3 times 10^0. In the binary system, the possible digits are 0 and 1, and each bit position represents a power of two. That is, the positions represent successive powers of two: 2^0, 2^1, 2^2, and so on. Figure 3.2 illustrates the positional concept for binary numbers.

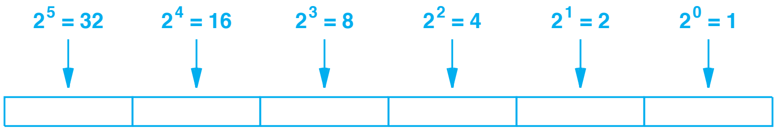


Figure 3.2 The value associated with each of the first six bit positions when using a positional interpretation in base two.

As an example, consider the binary number:

$$0\,1\,0\,1\,0\,1$$

According to the figure, the value can be interpreted as:

$$0\,1\,0\,1\,0\,1 \;=\; 0\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 \;=\; 21$$

We will discuss more about specific forms of integer representation (including negative numbers) later in the chapter. For now, it is sufficient to observe an important consequence of conventional positional notation: the binary numbers that can be represented in k bits start at zero instead of one. If we use the positional interpretation illustrated in Figure 3.2, the binary numbers that can be represented with three bits range from zero through seven. Similarly, the binary numbers that can be represented with eight bits range from zero through two hundred fifty-five. We can summarize:

> *A set of k bits can be interpreted to represent a binary integer. When conventional positional notation is used, the values that can be represented with k bits range from 0 through 2^k-1.*

Because it is an essential skill in the design of both software and hardware, anyone working in those fields should know the basics. Figure 3.3 lists the decimal equivalents of binary numbers that hardware and software designers should know. The table includes entries for 2^{32} and 2^{64} (an incredibly large number). Although smaller values in the table should be memorized, hardware and software designers only need to know the order of magnitude of the larger entries. Fortunately, is it easy to remember that 2^{32} contains ten decimal digits and 2^{64} contains twenty.

3.6 Bit Ordering

The positional notation in Figure 3.2 may seem obvious. After all, when writing decimal numbers, we always write the least significant digit on the right and the most significant digit on the left. Therefore, when writing binary, it makes sense to write the *Least Significant Bit* (*LSB*) on the right and the *Most Significant Bit* (*MSB*) on the left. When digital logic is used to store an integer, however, the concepts of "right" and "left" no longer make sense. Therefore, a computer architect must specify exactly how bits are stored, and which are the least and most significant.

The idea of bit ordering is especially important when bits are transferred from one location to another. For example, when a numeric value is moved between a register and memory, the bit ordering must be preserved. Similarly, when sending data across a network, the sender and receiver must agree on the bit ordering. That is, the two ends must agree whether the LSB or the MSB will be sent first.

Power Of 2	Decimal Value	Decimal Digits
0	1	1
1	2	1
2	4	1
3	8	1
4	16	2
5	32	2
6	64	2
7	128	3
8	256	3
9	512	3
10	1024	4
11	2048	4
12	4096	4
15	16384	5
16	32768	5
20	1048576	7
30	1073741824	10
32	4294967296	10
64	18446744073709551616	20

Figure 3.3 Decimal values for commonly used powers of two.

3.7 Hexadecimal Notation

Although a binary number can be translated to an equivalent decimal number, programmers and engineers sometimes find the decimal equivalent difficult to understand. For example, if a programmer needs to test the fifth bit from the right, using the binary constant 010000 makes the correspondence between the constant and the bit much clearer than the equivalent decimal constant 16.

Unfortunately, long strings of bits are as unwieldy and difficult to understand as a decimal equivalent. For example, to determine whether the sixteenth bit is set in the following binary number, a human needs to count individual bits:

1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1

To aid humans in expressing binary values, a compromise has been reached: a positional numbering system with a larger base. If the base is chosen to be a power of two, translation to binary is trivial. Base eight (known as *octal*) has been used, but base sixteen (known as *hexadecimal*) has become especially popular.

Hexadecimal representation offers two advantages. First, because the representation is substantially more compact than binary, the resulting strings are shorter. Second, because sixteen is a power of two, conversion between binary and hexadecimal is straightforward and does not involve a complex arithmetic calculation (i.e., a human can perform the transformation easily and quickly, without the need for a calculator or other tools).

In essence, hexadecimal encodes each group of four bits as a single hex† digit between zero and fifteen. Figure 3.4 lists the sixteen hex digits along with the binary and decimal equivalent of each. The figure and the examples that follow use uppercase letters *A* through *F* to represent hex digits above nine. Some programmers and some programming languages use lowercase letters *a* through *f* instead; the distinction is unimportant and programmers should be prepared to use either form.

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

Figure 3.4 The sixteen hexadecimal digits and their equivalent binary and decimal values. Each hex digit encodes four bits of a binary value.

As an example of hexadecimal encoding, Figure 3.5 illustrates how a binary string corresponds to its hexadecimal equivalent.

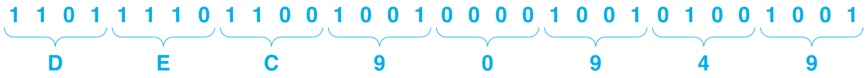


Figure 3.5 Illustration of the relationship between binary and hexadecimal. Each hex digit represents four bits.

†Programmers use the term *hex* as an abbreviation for *hexadecimal*.

3.8 Notation For Hexadecimal And Binary Constants

Because the digits used in binary, decimal, and hexadecimal number systems overlap, constants can be ambiguous. To solve the ambiguity, an alternate notation is needed. Mathematicians and some textbooks add a subscript to denote a base other than ten (e.g., 135_{16} specifies that the constant is hexadecimal). Computer architects and programmers tend to follow programming language notation: hex constants begin with prefix *0x*, and binary constants begin with prefix *0b*. Thus, to denote 135_{16}, a programmer writes 0x135. Similarly, the 32-bit constant from Figure 3.5 is written:

0xDEC90949

3.9 Character Sets

We said that bits have no intrinsic meaning, and that the hardware or software must determine what each bit represents. More important, more than one interpretation can be used — a set of bits can be created and used with one interpretation and later used with another.

As an example, consider character data that has both a numeric and symbolic interpretation. Each computer system defines a *character set*† to be a set of symbols that the computer and I/O devices agree to use. A typical character set contains uppercase and lowercase letters, digits, and punctuation marks. More important, computer architects often choose a character set such that each character fits into a byte (i.e., each of the bit patterns in a byte is assigned one character). Thus, a computer that uses an eight-bit byte has two hundred fifty-six (2^8) characters in its character set, and a computer that uses a six-bit byte has sixty-four (2^6) characters. In fact, the relationship between the byte size and the character set is so strong that many programming languages refer to a byte as a *character*.

What bit values are used to encode each character? The computer architect must decide. In the 1960s, for example, IBM Corporation chose the *Extended Binary Coded Decimal Interchange Code* (*EBCDIC*) representation as the character set used on IBM computers. CDC Corporation chose a six-bit character set for use on their computers. The two character sets were completely incompatible.

As a practical matter, computer systems connect to devices such as keyboards, printers, or modems, and such devices are often built by separate companies. To interoperate correctly, peripheral devices and computer systems must agree on which bit pattern corresponds to a given symbolic character. To help vendors build compatible equipment, the *American National Standards Institute* (*ANSI*) defined a character representation known as the *American Standard Code for Information Interchange* (*ASCII*). The ASCII character set specifies the representation of one hundred twenty-eight characters, including the usual letters, digits, and punctuation marks; additional values in an eight-bit byte can be assigned for special symbols. The standard is widely accepted.

†Names of character sets are pronounced, not spelled out. For example, EBCDIC is pronounced *ebb'se-dick*, and ASCII is pronounced *ass'key*.

Figure 3.6 lists the ASCII representation of characters by giving a hexadecimal value and the corresponding symbolic character. Of course, the hexadecimal notation is merely a shorthand notation for a binary string. For example, the lowercase letter *a* has hexadecimal value 0x61, which corresponds to the binary value 0b01100001.

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0A lf	0B vt	0C np	0D cr	0E so	0F si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1A sub	1B esc	1C fs	1D gs	1e rs	1F us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 ’
28 (	29)	2A *	2B +	2C ,	2D –	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 ‘	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F del

Figure 3.6 The ASCII character set. Each entry shows a hexadecimal value and the graphical representation for printable characters and the meaning for others.

We said that a conventional computer uses eight-bit bytes, and that ASCII defines one hundred twenty-eight characters (i.e., a seven-bit character set). Thus, when ASCII is used on a conventional computer, one-half of the byte values are unassigned (decimal values 128 through 255). How are the additional values used? In some cases, they are not — peripheral devices that accept or deliver characters merely ignore the eighth bit in a byte. In other cases, the computer architect or a programmer extends the character set (e.g., by adding punctuation marks for alternate languages).

3.10 Unicode

Although a seven-bit character set and an eight-bit byte work well for English and some European languages, they do not suffice for all languages. Chinese, for example, contains thousands of symbols and glyphs. To accommodate such languages, extensions and alternatives have been proposed.

One of the widely accepted extended character sets is named *Unicode*. Unicode extends ASCII and is intended to accommodate all languages, including languages from the Far East. Originally designed as a sixteen-bit character set, later versions of Unicode have been extended to accommodate larger representations. Thus, future computers and I/O devices may base their character set on Unicode.

3.11 Unsigned Integers, Overflow, And Underflow

The positional representation of binary numbers illustrated in Figure 3.2† is said to produce *unsigned integers*. That is, each of 2^k combinations of bits is associated with a nonnegative numeric value. Because the unsigned integers used in a computer have finite size, operations like addition and subtraction can have unexpected results. For example, subtracting a positive k-bit unsigned integer from a smaller positive k-bit unsigned integer can yield a negative (i.e., signed) result. Similarly, adding two k-bit unsigned integers can produce a value that requires more than k bits to represent.

Hardware to perform unsigned binary arithmetic handles the problem in an interesting way. First, the hardware produces a result by using *wraparound* (i.e., the hardware adds two k-bit integers, and takes the k low-order bits of the answer). Second, the hardware sets *overflow* or *underflow* conditions to indicate whether the result exceeded k bits or was negative‡. For example, an overflow indicator corresponds to the value that would appear in the k+1st bit (i.e., the value commonly known as *carry*). Figure 3.7 illustrates an addition with three-bit arithmetic that results in a carry.

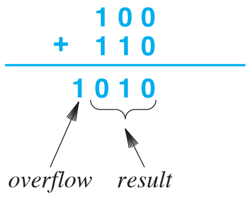


Figure 3.7 Illustration of addition with unsigned integers that produces overflow. The overflow indicator, which tells whether wraparound occurred, is equal to the carry bit.

3.12 Numbering Bits And Bytes

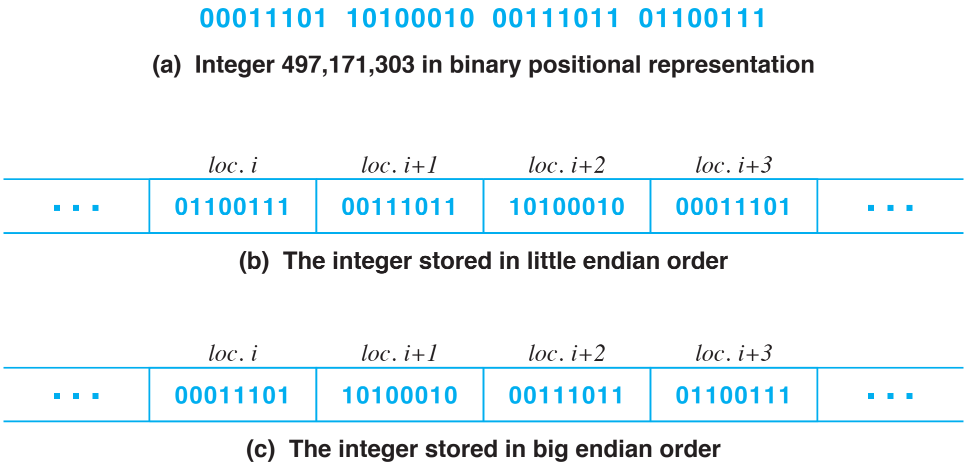
How should a set of bits be numbered? If we view the set as a string, it makes sense to start numbering from the left, but if we view the set as a binary number, it makes sense to start numbering from the right (i.e., from the numerically least significant bit). Numbering is especially important when data is transferred over a network because the sending and receiving computers must agree on whether the least-significant or most-significant bit will be transferred first.

†Figure 3.2 can be found on page 47.

‡The term *underflow* denotes a value that is less than the representation can hold. A negative result from unsigned integer arithmetic is classified as an underflow because negative values cannot be represented.

The issue of numbering becomes more complicated if we consider data items that span multiple bytes. For example, consider an integer that consists of thirty-two bits. If the computer uses eight-bit bytes, the integer will span four bytes, which can be transferred starting with the least-significant or the most-significant byte.

We use the term *little endian* to characterize a system that stores and transmits bytes of an integer from least significant to most significant, and the term *big endian* to characterize a system that stores and transmits bytes of an integer from most significant to least significant. Similarly, we use the terms *bit little endian* and *bit big endian* to characterize systems that transfer bits within a byte starting at the least-significant bit and most-significant bit, respectively. We can think of the bytes of an integer as being stored in an array, and the endianness determines the direction in memory. Figure 3.8 uses an example integer to illustrate the two byte orders, showing positional representation and the arrangement of bytes in memory using both little endian order and big endian order.



00011101 10100010 00111011 01100111

(a) Integer 497,171,303 in binary positional representation

	loc. i	*loc. i+1*	*loc. i+2*	*loc. i+3*	
. . .	01100111	00111011	10100010	00011101	. . .

(b) The integer stored in little endian order

	loc. i	*loc. i+1*	*loc. i+2*	*loc. i+3*	
. . .	00011101	10100010	00111011	01100111	. . .

(c) The integer stored in big endian order

Figure 3.8 (a) Integer 497,171,303 expressed as a 32-bit binary value, with spaces used to mark groups of eight bits, (b) the integer stored in successive memory locations using little endian order, and (c) the integer stored in successive memory locations using big endian order.

The big endian representation may seem appealing because it mimics the order humans use to write numbers. Surprisingly, little endian order has several advantages for computing. For example, little endian allows a programmer to use a single memory address to refer to all four bytes of an integer, the two low-order bytes, or only the lowest-order byte.

3.13 Signed Binary Integers

The positional representation described in Section 3.5 has no provision for negative numbers. To accommodate negative numbers, we need an alternative. Three interpretations have been used:

- *Sign Magnitude*. With sign-magnitude representation, bits of an integer are divided into a sign bit (1 if the number is negative and 0 otherwise) and a set of bits that gives the absolute value (i.e., the magnitude) of the integer. The magnitude field follows the positional representation illustrated in Figure 3.2.
- *One's Complement*. The set of bits is interpreted as a single field. A positive integer uses the positional representation illustrated in Figure 3.2 with the restriction that for an integer of *k* bits, the maximum positive value is 2^{k-1}. To form a negative of any value, invert each bit (i.e., change from 0 to 1 or vice versa). The most significant bit tells the sign of the integer (1 for negative integers, 0 otherwise).
- *Two's Complement*. The set of bits is interpreted as a single field. A positive integer uses the positional representation illustrated in Figure 3.2 with the restriction that for an integer of *k* bits, the maximum positive value is $2^{k-1}-1$. Thus, positive integers have the same representation as in one's complement. To form a negative number, start with a positive number, subtract one, and then invert each bit. As with one's complement, the most significant bit tells the sign of the integer (1 for negative integers, 0 otherwise).

Each interpretation has interesting quirks. For example, the sign-magnitude interpretation makes it possible to create a value of *negative zero*, even though the concept does not correspond to a valid mathematical concept. The one's complement interpretation provides two values for zero: all zero bits and the complement, all one bits. Finally, the two's complement interpretation includes one more negative value than positive values (to accommodate zero).

Which interpretation is best? Programmers can debate the issue because each interpretation works well in some cases. However, programmers cannot choose because computer architects make the decision and build hardware accordingly. Each of the three representations has been used in at least one computer. Many hardware architectures use the two's complement scheme. There are two reasons. First, two's complement makes it possible to build low-cost, high-speed hardware to perform arithmetic operations. Second, as the next section explains, hardware for two's complement arithmetic can also handle unsigned arithmetic.

3.14 An Example Of Two's Complement Numbers

We said that k bits can represent 2^k possible combinations. Unlike the unsigned representation in which the combinations correspond to a continuous set of integers starting at zero, two's complement divides the combinations in half. Each combination in the first half (zero through $2^{k-1}-1$) is assigned the same value as in the unsigned representation. Combinations in the second half, each of which has the high-order bit equal to one, correspond to negative integers. Thus, at exactly one-half of the way through the possible combinations, the value changes from the largest possible positive integer to the negative integer with the largest absolute value.

An example will clarify the two's complement assignment. To keep the example small, we will consider a four-bit integer. Figure 3.9 lists the sixteen possible bit combinations and the decimal equivalent when using unsigned, sign magnitude, one's complement, and two's complement representations.

Binary String	Unsigned (positional) Interpretation	Sign Magnitude Interpretation	One's Complement Interpretation	Two's Complement Interpretation
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	−0	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

Figure 3.9 The decimal value assigned to each combination of four bits when using unsigned, sign-magnitude, one's complement, and two's complement interpretations.

As noted above, unsigned and two's complement have the advantage that except for overflow, the same hardware operations work for both representations. For example, adding one to the binary value 1001 produces 1010. In the unsigned interpretation, adding one to nine produces ten; in the two's complement interpretation, adding one to negative seven produces negative six.

The important point is:

> *A computer can use a single hardware circuit to provide unsigned or two's complement integer arithmetic; software running on the computer can choose an interpretation for each integer.*

3.15 Sign Extension

Although Figure 3.9 shows four-bit binary strings, the ideas can be extended to an arbitrary number of bits. Many computers include hardware for multiple integer sizes (e.g., a single computer can offer sixteen bit, thirty-two bit, and sixty-four bit representations), and allow a programmer to choose one of the sizes for each integer data item.

If a computer does contain multiple sizes of integers, a situation can arise in which a value is copied from a smaller-size integer to a larger-size integer. For example, consider copying a value from a sixteen-bit integer to a thirty-two-bit integer. What should be placed in the extra bits? In two's complement, the solution consists of copying the least significant bits and then extending the sign bit — if the original value is positive, extending the high-order bit fills the most significant bits of the larger number with zeros; if the original value is negative, extending the high-order bit fills the most significant bits of the larger number with ones. In either case, the integer with more bits will be interpreted to have the same numeric value as the integer with fewer bits†.

We can summarize:

> *Sign extension: in two's complement arithmetic, when an integer Q composed of k bits is copied to an integer of more than k bits, the additional high-order bits are made equal to the top bit of Q. Extending the sign bit ensures that the numeric value of the two will be the same if each is interpreted as a two's complement value.*

Because two's complement hardware gives correct results when performing arithmetic operations on unsigned values, it may seem that software could use the hardware to support all unsigned operations. However, sign extension provides an exception to the rule: the hardware will always perform sign extension, which may have unexpected results. For example, if an unsigned integer is copied to a larger unsigned integer, the copy will not have the same numeric value if the high-order bit is 1. The point is:

> *Because two's complement hardware performs sign extension, copying an unsigned integer to a larger unsigned integer can change the value.*

†Because division and multiplication by powers of two can be implemented with shift operations, sign extension occurs during a right-shift operation, which results in the correct value. Thus, shifting integer -14 right one bit results in -7, and shifting integer 14 right one bit results in 7.

3.16 Floating Point

In addition to hardware that performs signed and unsigned integer arithmetic, general-purpose computers provide hardware that performs arithmetic on *floating point* values. Floating point representation used in computers derives from *scientific notation* in which each value is represented by a *mantissa* and an *exponent*. For example, scientific notation expresses the value –12345 as -1.2345×10^4. Similarly, a chemist might write a well-known constant, such as Avogadro's number, as:

$$6.023 \times 10^{23}$$

Unlike conventional scientific notation, the floating point representation used in computers is based on binary. Thus, a floating point value consists of a bit string that is divided into three fields: a bit to store the sign, a group of bits that stores a mantissa, and a third group of bits that stores an exponent. Unlike conventional scientific notation, everything in floating point is based on powers of two. For example, the mantissa uses a binary positional representation to store a value, and the exponent is an integer that specifies a power of 2 rather than a power of 10. In scientific notation, we think of an exponent as specifying how many digits to shift the decimal point; in floating point, the exponent specifies how many bits to shift the binary point.

To further optimize space, many floating point representations include optimizations:

- The value is normalized.
- The most significant bit of the mantissa is implicit.
- The exponent is biased to simplify magnitude comparison.

The first two optimizations are related. A floating point number is *normalized* by adjusting the exponent to eliminate leading zeros from the mantissa. In decimal, for example, 0.003×10^4 can be normalized to 3×10^1. Interestingly, normalizing a binary floating point number always produces a leading bit of 1 (except in the special case of the number zero). Therefore, to increase the number of bits of precision in the mantissa, floating point representations do not need to store the most significant bit of the mantissa when a value is stored in memory. Instead, when a floating point number computation is required, the hardware concatenates a 1 bit onto the mantissa.

An example will clarify the concepts. The example we will use is IEEE† standard 754, which is widely used in the computer industry. The standard specifies both *single precision* and *double precision* numbers. According to the standard, a single precision value occupies thirty-two bits, and a double precision value occupies sixty-four bits. Figure 3.10 illustrates how the IEEE standard divides a floating point number into three fields.

†IEEE stands for Institute of Electrical and Electronics Engineers, an organization that creates standards used in electronic digital systems.

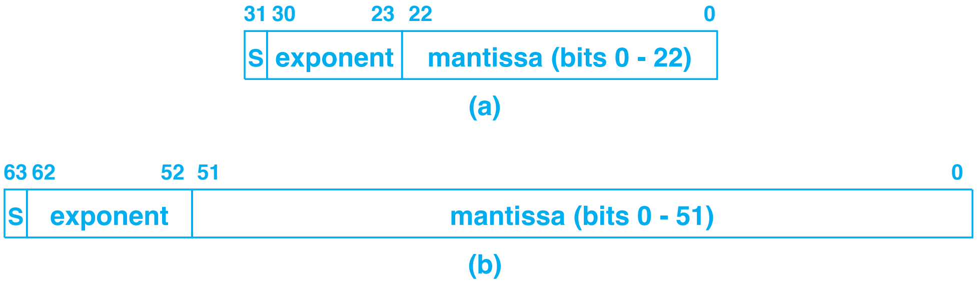


Figure 3.10 The format of (a) a single precision and (b) a double precision floating point number according to IEEE Standard 754, with the lowest bit in each field labeled. Fields consist of a sign bit, exponent, and mantissa.

Bit numbering in the figure follows the IEEE standard, in which the least significant bit is assigned bit number zero. In single precision, for example, the twenty-three rightmost bits, which constitute a mantissa, are numbered zero through twenty-two. The next eight bits, which constitute an exponent, are numbered twenty-three through thirty, and the most significant bit, which contains a sign, is bit number thirty-one. For double precision, the mantissa occupies fifty-two bits and the exponent occupies eleven bits.

3.17 Range Of IEEE Floating Point Values

The IEEE standard for single precision floating point allows normalized values in which the exponent ranges from negative one hundred twenty-six through one hundred twenty-seven. Thus, the approximate range of values that can be represented is:

$$2^{-126} \text{ to } 2^{127}$$

which, in decimal, is approximately:

$$10^{-38} \text{ to } 10^{38}$$

The IEEE standard for double precision floating point provides an enormously larger range than single precision. The range is:

$$2^{-1022} \text{ to } 2^{1023}$$

which, in decimal, is approximately:

$$10^{-308} \text{ to } 10^{308}$$

To make magnitude comparison fast, the IEEE standard specifies that an exponent field stores the exponent (a power of two) plus a *bias constant*. The bias constant used with single precision is one hundred twenty-seven, and the bias constant used with double precision is one thousand twenty-three†. For example, to store an exponent of three, the exponent field in a single precision value is assigned the value one hundred thirty, and an exponent of negative five is represented by one hundred twenty-two.

As an example of floating point, consider how 6.5 is represented. In binary, 6 is 110, and .5 is a single bit following the binary point, giving us 110.1 (binary). If we use binary scientific notation and normalize the value, 6.5 can be expressed:

$$1.101 \times 2^2$$

To express the value as an IEEE single precision floating point number, the sign bit is zero, and the exponent must be biased by 127, making it 129. In binary, 129 is:

$$10000001$$

To understand the value in the mantissa, recall that the leading 1 bit is not stored, which means that instead of 1101 followed by zeros, the mantissa is stored as:

$$10100000000000000000000$$

Figure 3.11 shows how the fields combine to form a single-precision IEEE floating point representation of 6.5.

S	exponent (23 – 30)	mantissa (bits 0 – 22)
0	1 0 0 0 0 0 0 1	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 3.11 The value 6.5 (decimal) represented as a single-precision IEEE floating point constant.

3.18 Special Values

Like most floating point representations, the IEEE standard follows the implicit leading bit assumption — a mantissa is assumed to have a leading one bit that is not stored. Of course, any representation that strictly enforces the assumption of a leading one bit is useless because the representation cannot store the value zero. To handle zero, the IEEE standard makes an exception — when all bits are zero, the implicit assumption is ignored, and the stored value is taken to be zero.

The IEEE standard includes two other special values that are reserved to represent positive and negative infinity: the exponent contains all ones and the mantissa contains all zeros. The point of including values for infinity is that some digital systems do not

†The bias constant is always $2^{k-1}-1$, where k is the number of bits in the exponent field.

have facilities to handle errors such as arithmetic overflow. On such systems, it is important that a value be reserved so that the software can determine that a floating point operation failed.

3.19 Binary Coded Decimal Representation

Most computers employ the binary representations for integers and floating point numbers described above. Because the underlying hardware uses digital logic, binary digits of 0 and 1 map directly onto the hardware. As a result, hardware can compute binary arithmetic efficiently and all combinations of bits are valid. However, two disadvantages arise from the use of binary representations. First, the range of values is a power of two rather than a power of ten (e.g., the range of an unsigned 32-bit integer is zero to 4,294,967,295). Second, floating point values are rounded to binary fractions rather than decimal fractions.

The use of binary fractions has some unintended consequences, and their use does not suffice for all computations. For example, consider a bank account that stores U.S. dollars and cents. We usually represent cents as hundredths of dollars, writing 5.23 to denote five dollars and 23 cents. Surprisingly, one hundredth (i.e., one cent) cannot be represented exactly as a binary floating point number because it turns into a repeating binary fraction. Therefore, if binary floating point arithmetic is used for bank accounts, individual pennies are rounded, making the totals inaccurate. In a scientific sense, the inaccuracy is bounded, but humans demand that banks keep accurate records — they become upset if a bank preserves significant digits of their account but loses pennies.

To accommodate banking and other computations where decimal is required, a *Binary Coded Decimal* (*BCD*) representation is used. Some computers (notably on IBM mainframes) have hardware to support BCD arithmetic; on other computers, software performs all arithmetic operations on BCD values.

Although a variety of BCD formats have been used, the essence is always the same: a value is represented as a string of decimal digits. The simplest case consists of a character string in which each byte contains the character for a single digit. However, the use of character strings makes computation inefficient and takes more space than needed. As an example, if a computer uses the ASCII character set, the integer 123456 is stored as six bytes with values†:

0x31 0x32 0x33 0x34 0x35 0x36

If a character format is used, each ASCII character (e.g., 0x31) must be converted to an equivalent binary value (e.g., 0x01) before arithmetic can be performed. Furthermore, once an operation has been performed, the digits of the result must be converted from binary back to the character format. To make computation more efficient, modern BCD systems represent digits in binary rather than as characters. Thus, 123456 could be represented as:

0x01 0x02 0x03 0x04 0x05 0x06

†Although our examples use ASCII, BCD is typically used on IBM computers that employ the EBCDIC character set.

Although the use of a binary representation has the advantage of making arithmetic faster, it also has a disadvantage: a BCD value must be converted to character format before it can be displayed or printed. The general idea is that because arithmetic is performed more frequently than I/O, keeping a binary form will improve overall performance.

3.20 Signed, Fractional, And Packed BCD Representations

Our description of BCD omits many details found in commercial systems. For example, an implementation may limit the size of a BCD value. To handle fractions, BCD must either include an explicit decimal point or the representation must specify the location of the decimal point. Furthermore, to handle signed arithmetic, a BCD representation must include a sign. Interestingly, one of the most widely used BCD conventions places the sign byte at the right-hand end of the BCD string. Thus -123456 might be represented by the sequence:

0x01 0x02 0x03 0x04 0x05 0x06 0x2D

where 0x2D is a value used to indicate a minus sign. The advantage of placing the sign on the right arises because no scanning is required when arithmetic is performed — all bytes except the last byte of the string correspond to decimal digits.

The final detail used with BCD encodings arises from the observation that using a byte for each digit is inefficient. Each digit only requires four bits, so placing one digit in each eight-bit byte wastes half of each byte. To reduce the storage space needed for BCD, a *packed* representation is used in which each digit occupies a *nibble* (i.e., four bits). With a packed version of BCD, the integer -123456 can be represented in four bytes:

0x01 0x23 0x45 0x6D

where the last nibble contains the value 0xD to indicate that the number is negative†.

3.21 Data Aggregates

So far, we have only considered the representation for individual data items such as characters, integers, or floating point numbers. Most programming languages allow a programmer to specify *aggregate* data structures that contain multiple data items, such as *arrays*, *records*, or *structs*. How are such values stored? In general, an aggregate value occupies contiguous bytes. Thus, on a computer that uses an eight-bit byte, a data aggregate that consists of three sixteen-bit integers occupies six contiguous bytes as Figure 3.12 illustrates.

†To aid with BCD arithmetic, the x86 architecture has a condition code bit that indicates whether 4-bit addition overflows.

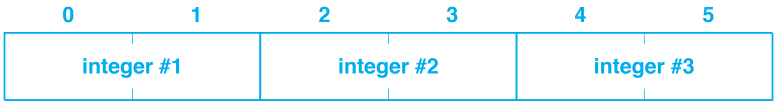


Figure 3.12 A data aggregate consisting of three sixteen-bit integers arranged in successive bytes of memory numbered 0 through 5.

We will see later that some memory systems do not permit arbitrary data types to be contiguous. Thus, we will reconsider data aggregates when we discuss memory architecture.

3.22 Program Representation

Modern computers are classified as *stored program computers* because programs as well as data are placed in memory. We will discuss program representation and storage in the next chapters, including the structure of instructions the computer understands and their storage in memory. For now, it is sufficient to understand that each computer defines a specific set of operations and a format in which each is stored. On some computers, for example, each instruction is the same size as other instructions; on other computers, the instruction size varies. We will see that on a typical computer, an instruction occupies multiple bytes. Thus, the bit and byte numbering schemes that the computer uses for data values also apply to instructions.

3.23 Summary

The underlying digital hardware has two possible values, logical 0 and logical 1. We think of the two values as defining a bit (binary digit), and use bits to represent data and programs. Each computer defines a byte size, and most current systems use eight bits per byte.

A set of bits can be used to represent a character from the computer's character set, an unsigned integer, a single or double precision floating point value, or a computer program. Representations are chosen carefully to maximize the flexibility and speed of the hardware while keeping the cost low. The two's complement representation for signed integers is particularly popular because a single piece of hardware can be constructed that performs operations on either two's complement integers or unsigned integers. In cases where decimal arithmetic is required, computers use Binary Coded Decimal values in which a number is represented by a string that specifies individual decimal digits.

Organizations, such as ANSI and IEEE, have created standards for representation; such standards allow hardware manufactured by two separate organizations to interoperate and exchange data.

EXERCISES

3.1 Give a mathematical proof that a string of k bits can represent 2^k possible values (hint: argue by induction on the number of bits).

3.2 What is the value of the following binary string in hexadecimal?

1101 1110 1010 1101 1011 1110 1110 1111

3.3 Write a computer program that determines whether the computer on which it is running uses big endian or little endian representation for integers.

3.4 Write a computer program that prints a string of zeros and ones that represents the bits of an integer. Place a blank between each bit, and add an extra space after every four bits.

3.5 Write a computer program that determines whether the computer on which it is running uses one's complement, two's complement, or (possibly) some other representation for signed integers.

3.6 Write a computer program that determines whether the computer on which it is running uses the ASCII or EBCDIC character set.

3.7 Write a computer program that takes a set of integers as input and for each integer prints the two's complement, one's complement, and sign-magnitude representation of the integer.

3.8 Write a C program that prints a table of all possible eight-bit binary values and the two's complement interpretation of each.

3.9 Write a computer program that adds one to the largest possible positive integer and uses the result to determine whether the computer implements two's complement arithmetic.

3.10 Write a computer program to display the value of a byte in hexadecimal, and apply the program to an array of bytes. Add an extra space after every four bytes to make the output easier to read.

3.11 Extend the hexadecimal dump program in the previous exercise to also print the character representation of any printable character. For characters that do not have a printable representation, arrange for the program to print a period.

3.12 A programmer computes the sum of two unsigned 32-bit integers. Can the resulting sum be less than either of the two values? Explain.

3.13 Suppose you are given a computer with hardware that can only perform 32-bit arithmetic, and are asked to create functions that add and subtract 64-bit integers. How can you perform 64-bit computations with 32-bit hardware? (To simplify the problem, limit your answer to unsigned arithmetic.)

3.14 The C Programming language allows a programmer to specify constants in decimal, binary, hexadecimal, and octal. Write a program that declares 0, 5, 65, 128, and -1 and -256 in decimal, binary, hexadecimal, and octal, and uses printf to show that the values are correct. Which is the easiest representation?

3.15 Create a form of Binary Coded Decimal similar to the one described in the text, and write a computer program that uses the form to add two arbitrary length integers.

3.16 Extend the previous program to include multiplication.

3.17 The financial industry uses a "bankers" rounding algorithm. Read about the algorithm, and implement a program that uses decimal arithmetic to compute the sum of the two decimal values with both bankers rounding and conventional rounding.

Part II

Processors

The Engines That Drive Computation

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

4

The Variety Of Processors And Computational Engines

4.1 Introduction

Previous chapters describe the basic building blocks used to construct computer systems: digital logic and representations used for data types such as characters, integers, and floating point numbers. This chapter begins an investigation of one of three key elements of any computer system: a processor. The chapter introduces the general concept, describes the variety of processors, and discusses the relationship between clock rate and processing rate. The next chapters extend the basic description by explaining instruction sets, addressing modes, and the functions of a general-purpose CPU.

4.2 The Two Basic Architectural Approaches

Early in the history of computers, architects experimenting with new designs considered how to organize the hardware. Two basic approaches emerged that are named for the groups who proposed them:

- Harvard Architecture
- Von Neumann Architecture

We will see that the two share ideas, and only differ in how programs and data are stored and accessed.

4.3 The Harvard And Von Neumann Architectures

The term *Harvard Architecture*† refers to a computer organization with four principal components: a processor, an instruction memory, a data memory, and I/O facilities, organized as Figure 4.1 illustrates.

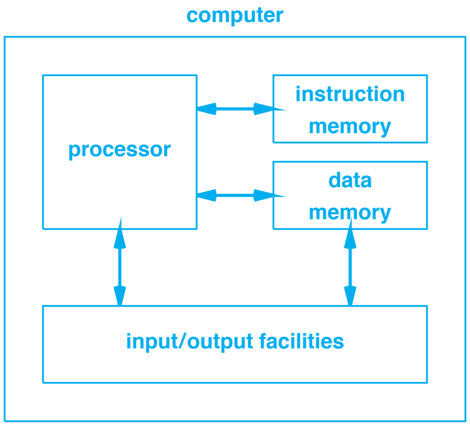


Figure 4.1 Illustration of the Harvard Architecture that uses two memories, one to hold programs and another to store data.

Although it includes the same basic components, a *Von Neumann Architecture*‡ uses a single memory to hold both programs and data. Figure 4.2 illustrates the approach.

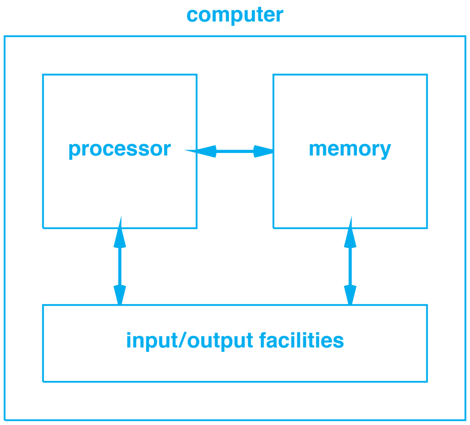


Figure 4.2 Illustration of the Von Neumann Architecture. Both programs and data can be stored in the same memory.

†The name arises because the approach was first used on the *Harvard Mark I* relay computer.

‡The name is taken from John Von Neumann, a mathematician who first proposed the architecture.

The chief advantage of the Harvard Architecture arises from its ability to have one memory unit optimized to store programs and another memory unit optimized to store data. The chief disadvantage arises from inflexibility: when purchasing a computer, an owner must choose the size of the instruction memory and the size of data memory. Once the computer has been purchased, an owner cannot use part of the instruction memory to store data nor can he or she use part of the data memory to store programs. Although it has fallen out of favor for general-purpose computers, the Harvard Architecture is still sometimes used in small embedded systems and other specialized designs.

Unlike the Harvard Architecture, the Von Neumann Architecture offers complete flexibility: at any time, an owner can change how much of the memory is devoted to programs and how much to data. The approach has proven to be so valuable that it has become widely adopted:

> *Because it offers flexibility, the Von Neumann Architecture, which uses a single memory to hold both programs and data, has become pervasive: almost all computers follow the Von Neumann approach.*

We say a computer that follows the Von Neumann Architecture employs a *stored program* approach because a program is stored in memory. More, important programs can be loaded into memory just like other data items.

Except when noted, the remainder of the text implicitly assumes a Von Neumann Architecture. There are two primary exceptions in Chapters 6 and 12. Chapter 6, which explains data paths, uses a simplified Harvard Architecture in the example. Chapter 12, which explains caching, discusses the motivation for using separate instruction and data caches.

4.4 Definition Of A Processor

The remainder of this chapter considers the processor component present in both the Harvard and Von Neumann Architectures. The next sections define the term and characterize processor types. Later sections explore the subcomponents of complex processors.

Although programmers tend to think of a conventional computer and often use the term *processor* as a synonym for the *Central Processing Unit* (*CPU*), computer architects have a much broader meaning that includes the processors used to control the engine in an automobile, processors in hand-held remote control devices, and specialized video processors used in graphics equipment. To an architect, a *processor* refers to a digital device that can perform a computation involving multiple steps. Individual processors are not complete computers; they are merely one of the building blocks that an architect uses to construct a computer system. Thus, although it can compute more than the combinatorial Boolean logic circuits we examined in Chapter 2, a processor need not be large or fast. In particular, some processors are significantly less powerful than the general-purpose CPU found in a typical PC. The next sections help clarify the definition by examining characteristics of processors and explaining some of the ways they can be used.

4.5 The Range Of Processors

Because processors span a broad range of functionality and many variations exist, no single description adequately captures all the properties of processors. Instead, to help us appreciate the many designs, we need to divide processors into categories according to functionality and intended use. For example, we can use four categories to explain whether a processor can be adapted to new computations. The categories are listed in order of flexibility:

- Fixed logic
- Selectable logic
- Parameterized logic
- Programmable logic

A *fixed logic processor*, which is the least flexible, performs a single task. More important, all the functionality needed to perform the operation is built in when the processor is created, and the functionality cannot be altered without changing the underlying hardware†. For example, a fixed logic processor can be designed to compute a function, such as *sine(x)*, or to perform a graphics operation needed in a video game.

A *selectable logic processor* has slightly more flexibility than a fixed logic processor. In essence, a selectable logic processor contains facilities needed to perform more than one function; the exact function is specified when the processor is invoked. For example, a selectable logic processor might be designed to compute either *sine(x)* or *cosine(x)*.

A *parameterized logic processor* adds additional flexibility. Although it only computes a predetermined function, the processor accepts a set of parameters that control the computation. For example, consider a parameterized processor that computes a hash function, *h(x)*. The hash function uses two constants, *p* and *q*, and computes the hash of *x* by computing the remainder of *x* when multiplied by *p* and divided by *q*. For example, if *p* is 167 and *q* is 163, *h(26729)* is the remainder of 4463743 divided by 163, or 151‡. A parameterized processor for such a hash function allows constants *p* and *q* to be changed each time the processor is invoked. That is, in addition to the input, *x*, the processor accepts additional parameters, *p* and *q*, that control the operation.

A *programmable logic processor* offers the most flexibility because it allows the sequence of steps to be changed each time the processor is invoked — the processor can be given a program to run, typically by placing the program in memory.

†Engineers use the term *hardwired* for functionality that cannot be changed without altering the underlying wiring.

‡Hashing is often applied to strings. In the example, number 26729 is the decimal value of the two characters in the string “hi” when treated as an unsigned short integer.

4.6 Hierarchical Structure And Computational Engines

A large processor, such as a modern, general-purpose CPU, is so complex that no human can understand the entire processor as a single unit. To control the complexity, computer architects use a hierarchical approach in which subparts of the processor are designed and tested independently before being combined into the final design.

Some of the independent subparts of a large processor are so sophisticated that they fit our definition of a processor — the subpart can perform a computation that involves multiple steps. For example, a general-purpose CPU that has instructions for *sine* and *cosine* might be constructed by first building and testing a trigonometry processor, and then combining the trigonometry processor with other pieces to form the final CPU.

How do we describe a subpiece of a large, complex processor that acts independently and performs a computation? Some engineers use the term *computational engine*. The term *engine* usually implies that the subpiece fills a specific role and is less powerful than the overall unit. For example, Figure 4.3 illustrates a CPU that contains several engines.

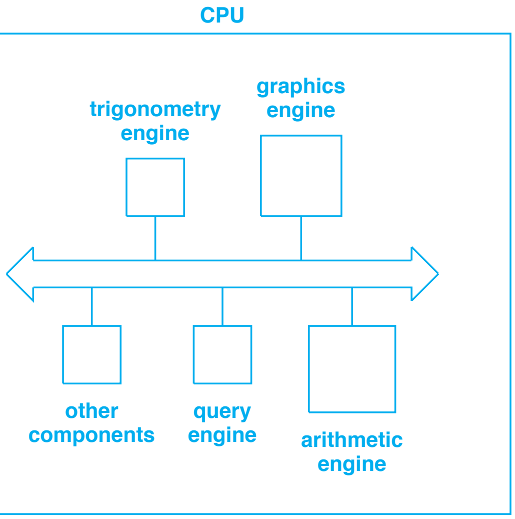


Figure 4.3 An example of a CPU that includes multiple components. The large arrow in the center of the figure indicates a central interconnect mechanism that the components use to coordinate.

The CPU in the figure includes a special-purpose *graphics engine*. Graphics engines, sometimes called *graphics accelerators*, are common because video game software is popular and many computers need a graphics engine to drive the graphics display at high speed. For example, a graphics engine might include facilities to repaint the surface of a graphical figure after it has been moved (e.g., in response to a joystick movement).

The CPU illustrated in Figure 4.3 also includes a *query engine*. Query engines and closely related *pattern engines* are used in database processors. A query engine examines a database record at high speed to determine if the record satisfies the query; a pattern engine examines a string of bits to determine if the string matches a specified pattern (e.g., to test whether a document contains a particular word). In either case, a CPU has enough capability to handle the task, but a special-purpose processor can perform the task much faster.

4.7 Structure Of A Conventional Processor

Although the imaginary CPU described in the previous section contains many engines, most processors do not. Two questions arise. First, what engine(s) are found in a conventional processor? Second, how are the engines interconnected? This section answers the questions broadly, and later sections give more detail.

Although a practical processor contains many subcomponents with complex interconnections among them, we can view a processor as having five conceptual units:

- Controller
- Arithmetic Logic Unit (ALU)
- Local data storage (typically, registers)
- Internal interconnection(s)
- External interface(s) (I/O buses)

Figure 4.4 illustrates the concept.

Controller. The controller forms the heart of a processor. Controller hardware has overall responsibility for program execution. That is, the controller steps through the program and coordinates the actions of all other hardware units to perform the specified operations.

Arithmetic Logic Unit (ALU). We think of the ALU as the main computational engine in a processor. The ALU performs all computational tasks, including integer arithmetic, operations on bits (e.g., left or right shift), and Boolean (logical) operations (e.g., Boolean *and*, *or*, *exclusive or*, and *not*). However, an ALU does not perform multiple steps or initiate activities. Instead, the ALU only performs one operation at a time, and relies on the controller to specify exactly what operation to perform on the operand values.

Local Data Storage. A processor must have at least some local storage to hold data values such as operands for arithmetic operations and the result. As we will see, local storage usually takes the form of hardware *registers* — values must be loaded into the hardware registers before they can be used in computation.

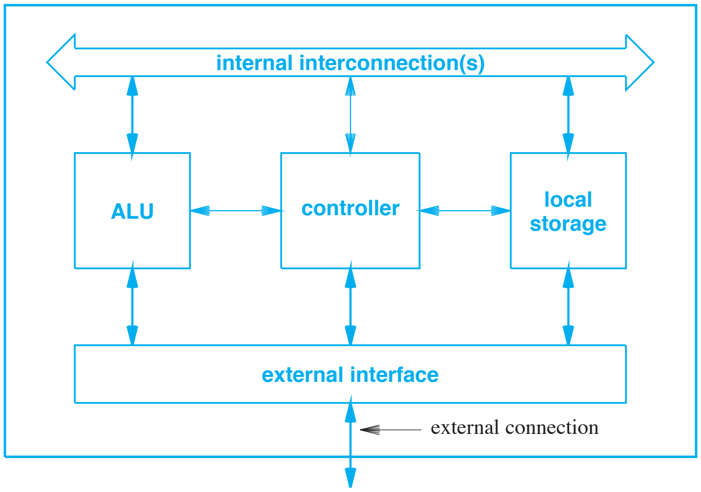


Figure 4.4 The five major units found in a conventional processor. The external interface connects to the rest of the computer system.

Internal Interconnection(s). A processor contains one or more hardware mechanisms that are used to transfer values between the other hardware units. For example, the interconnection hardware is used to move data values from the local storage to the ALU or to move results from the ALU to local storage. Architects sometimes use the term *data path* to describe an internal interconnection.

External Interface(s). The external interface unit handles all communication between the processor and the rest of the computer system. In particular, the external interface manages communication between the processor and external memory and I/O devices.

4.8 Processor Categories And Roles

Understanding the range of processors is especially difficult for someone who has not encountered hardware design because processors can be used in a variety of roles. It may help if we consider the ways that hardware devices use processors and how processors function in each role. Here are four examples:

- Coprocessors
- Microcontrollers
- Embedded system processors
- General-purpose processors

Coprocessors. A coprocessor operates in conjunction with and under the control of another processor. Usually, a coprocessor consists of a special-purpose processor that performs a single task at high speed. For example, some CPUs use a coprocessor known as a *floating point accelerator* to speed the execution of arithmetic operations — when a floating point operation occurs, the CPU automatically passes the necessary values to the coprocessor, obtains the result, and then continues execution. In architectures where a running program does not know which operations are performed directly by the CPU and which operations are performed by a coprocessor, we say that the operation of a coprocessor is *transparent* to the software. Typical coprocessors use fixed or selectable logic, which means that the functions the coprocessor can perform are determined when the coprocessor is designed.

Microcontrollers. A microcontroller consists of a programmable device dedicated to the control of a physical system. For example, microcontrollers run physical systems such as the engine in a modern automobile, the landing gear on an airplane, and the automatic door in a grocery store. In many cases, a microcontroller performs a trivial function that does not require much traditional computation. Instead, a microcontroller tests sensors and sends signals to control devices. Figure 4.5 lists an example of the steps a typical microcontroller can be programmed to perform:

```
do forever {
    wait for the sensor to be tripped;
    turn on power to the door motor;
    wait for a signal that indicates the
        door is open;
    wait for the sensor to reset;
    delay ten seconds;
    turn off power to the door motor;
}
```

Figure 4.5 Example of the steps a microcontroller performs. In most cases, microcontrollers are dedicated to trivial control tasks.

Embedded System Processors. An embedded system processor runs sophisticated electronic devices such as a wireless router or smart phone. The processors used for embedded systems are usually more powerful than the processors that are used as microcontrollers, and often run a protocol stack used for communication. However, the processor may not contain all the functionality found on more general-purpose CPUs.

General-purpose Processors. General-purpose processors are the most familiar and need little explanation. For example, the CPU in a PC is a general-purpose processor.

4.9 Processor Technologies

How are processors created? In the 1960s, processors were created from digital logic circuits. Individual gates were connected together on a circuit board, which then plugged into a chassis to form a working computer. By the 1970s, large-scale integrated circuit technology arrived, which meant that the smallest and least powerful processors — such as those used for microcontrollers — could each be implemented on a single integrated circuit. As integrated circuit technology improved and the number of transistors on a chip increased, a single chip became capable of holding more powerful processors. Today, many of the most powerful general-purpose processors consist of a single integrated circuit.

4.10 Stored Programs

We said that a processor performs a computation that involves multiple steps. Although some processors have the series of steps built into the hardware, most do not. Instead, they are *programmable* (i.e., they rely on a mechanism known as *programming*). That is, the sequence of steps to be performed comprise a program that is placed in a location the processor can access; the processor accesses the program and follows the specified steps.

Computer programmers are familiar with conventional computer systems that use main memory as the location that holds a program. The program is loaded into memory each time a user runs the application. The chief advantage of using main memory to hold programs lies in the ability to change the program. The next time a user runs a program after it has been changed, the altered version will be used.

Although our conventional notion of programming works well for general-purpose processors, other types of processors use alternative mechanisms that are not as easy to change. For example, the program for a microcontroller usually resides in hardware known as *Read Only Memory* (*ROM*†). In fact, a ROM that contains a program may reside on an integrated circuit along with a microcontroller that runs the program. For example, the microcontroller used in an automobile may reside on a single integrated circuit that also contains the program the microcontroller runs.

The important point is that programming is a broad notion:

> *To a computer architect, a processor is classified as programmable if, at some level of detail, the processor is separate from the program it runs. To a user, it may appear that the program and processor are integrated, and it may not be possible to change the program without replacing the processor.*

†Later chapters describe memory in more detail.

4.11 The Fetch-Execute Cycle

How does a programmable processor access and perform steps of a program? The data path description in Chapter 6 explains the basic idea. Although the details vary among processors, all programmable processors follow the same fundamental paradigm. The underlying mechanism is known as the *fetch-execute cycle*.

To implement fetch-execute, a processor has an instruction pointer that automatically moves through the program in memory, performing each step. That is, each programmable processor executes two basic functions repeatedly. Algorithm 4.1 presents the two fundamental steps†.

Algorithm 4.1

```
Repeat forever {

        Fetch: access the next step of the program from the
                  location in which the program has been stored.

        Execute: perform the step of the program.

}
```

Algorithm 4.1 The Fundamental Steps Of The Fetch-Execute Cycle

The important point is:

> *At some level, every programmable processor implements a fetch-execute cycle.*

Several questions arise. Exactly how is the program represented in memory, and how is such a representation created? How does a processor identify the next step of a program? What are the possible operations that can be performed during the execution phase of the fetch-execute cycle? How does the processor perform each operation? The next chapters will answer each of these questions in more detail. The remainder of this chapter concentrates on three questions: how fast does a processor operate, how does a processor begin with the first step of a program, and what happens when the processor reaches the end of a program?

†Note that the algorithm presented here is a simplified form; when we discuss I/O, we will see how the algorithm is extended to handle device interrupts.

4.12 Program Translation

An important question for programmers concerns how a program is converted to the form a processor expects. A programmer uses a *High Level Language* (*HLL*) to create a computer program. We say the programmer writes *source code*. The programmer uses a tool to translate the source code into the representation that a processor expects.

Although a programmer invokes a single tool, such as *gcc*, multiple steps are required to perform the translation. First, a *preprocessor* expands macros, producing a modified source program. The modified source program becomes input to a *compiler*, which translates the program into assembly language. Although it is closer to the form needed by a processor, assembly language can be read by humans. An *assembler* translates the assembly language program into a relocatable object program that contains a combination of binary code and references to external library functions. A *linker* processes the relocatable object program by replacing external function references with the code for the functions. To do so, the linker extracts the name of a function, searches one or more libraries to find binary code for the function. Figure 4.6 illustrates the translation steps and the software tool that performs each step.

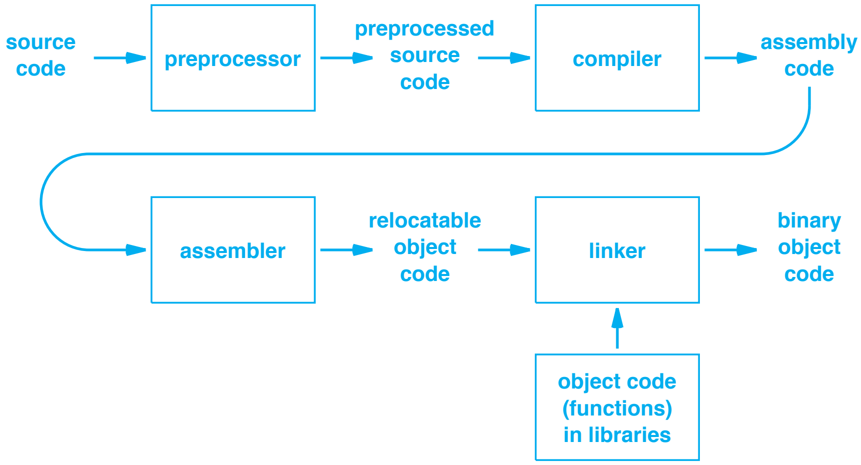


Figure 4.6 The steps used to translate a source program to the binary object code representation used by a processor.

4.13 Clock Rate And Instruction Rate

One of the primary questions about processors concerns speed: how fast does the fetch-execute cycle operate? The answer depends on the processor, the technology used to store a program, and the time required to execute each instruction. On one hand, a processor used as a microcontroller to actuate a physical device (e.g., an electric door) can be relatively slow because a response time under one-tenth of a second seems fast to a human. On the other hand, a processor used in the highest-speed computers must be as fast as possible because the goal is maximum performance.

As we saw in Chapter 2, most processors use a clock to control the rate at which the underlying digital logic operates. Anyone who has purchased a computer knows that sales personnel push customers to purchase a fast clock with the argument that a higher clock rate will result in higher performance. Although a higher clock rate usually means higher processing speed, it is important to realize that the clock rate does not give the rate at which the fetch-execute cycle proceeds. In particular, in most systems, the time required for the *execute* portion of the cycle depends on the instruction being executed. We will see later that operations involving memory access or I/O can require significantly more time (i.e., more clock cycles) than those that do not. The time also varies among basic arithmetic operations: integer multiplication or division requires more time than integer addition or subtraction. Floating point computation is especially costly because floating point operations usually require more clock cycles than equivalent integer operations. Floating point multiplication or division stands out as especially costly — a single floating point division can require orders of magnitude more clock cycles than an integer addition.

For now, it is sufficient to remember the general principle:

> *The fetch-execute cycle may not proceed at a fixed rate because the time taken to execute an instruction depends on the operation being performed. An operation such as multiplication requires more time than an operation such as addition.*

4.14 Control: Getting Started And Stopping

So far, we have discussed a processor running a fetch-execute cycle without giving details. We now need to answer two basic questions. How does the processor start running the fetch-execute cycle? What happens after the processor executes the last step in a program?

The issue of program termination is the easiest to understand: processor hardware is not designed to stop. Instead, the fetch-execute cycle continues indefinitely. Of course, a processor can be permanently halted, but such a sequence is only used to power down a computer — in normal operations, the processor continues to execute one instruction after another.

In some cases, a program uses a loop to delay. For example, a microcontroller may need to wait for a sensor to indicate an external condition has been met before proceeding. The processor does not merely stop to wait for the sensor. Instead, the program contains a loop that repeatedly tests the sensor. Thus, from a hardware point of view, the fetch-execute cycle continues.

The notion of an indefinite fetch-execute cycle has a direct consequence for programming: software must be planned so a processor always has a next step to execute. In the case of a dedicated system such as a microcontroller that controls a physical device, the program consists of an infinite loop — when it finishes the last step of the program, the processor starts again at the first step. In the case of a general-purpose computer, an operating system is always present. The operating system can load an application into memory, and then direct the processor to run the application. To keep the fetch-execute cycle running, the operating system must arrange to regain control when the application finishes. When no application is running, the operating system enters a loop to wait for input (e.g., from a touch screen, keyboard, or mouse).

To summarize:

> *Because a processor runs the fetch-execute cycle indefinitely, a system must be designed to ensure that there is always a next step to execute. In a dedicated system, the same program executes repeatedly; in a general-purpose system, an operating system runs when no application is running.*

4.15 Starting The Fetch-Execute Cycle

How does a processor start the fetch-execute cycle? The answer is complex because it depends on the underlying hardware. For example, some processors have a hardware *reset*. On such processors, engineers arrange for a combinatorial circuit to apply voltage to the reset line until all system components are ready to operate. When voltage is removed from the reset line, the processor begins executing a program from a fixed location. Some processors start executing a program found at location zero in memory once the processor is reset. In such systems, the designer must guarantee that a valid program is placed in location zero before the processor starts.

The steps used to start a processor are known as a *bootstrap*. In an embedded environment, the program to be run usually resides in *Read Only Memory* (*ROM*). On a conventional computer, the hardware reads a copy of the operating system from an I/O device, such as a disk, and places the copy into memory before starting the processor. In either case, hardware assist is needed for bootstrap because a signal must be passed to the processor that causes the fetch-execute cycle to begin.

Many devices have a *soft power switch*, which means that the power switch does not actually turn power on or off. Instead, the switch acts like a sensor — the processor can interrogate the switch to determine its current position. Booting a device that has a softswitch is no different than booting other devices. When power is first applied (e.g., when a battery is installed), the processor boots to an initial state. The initial state consists of a loop that interrogates the soft power switch. Once the user presses the soft power switch, the hardware completes the bootstrap process.

4.16 Summary

A processor is a digital device that can perform a computation involving multiple steps. Processors can use fixed, selectable, parameterized or programmable logic. The term *engine* identifies a processor that is a subpiece of a more complex processor.

Processors are used in various roles, including coprocessors, microcontrollers, embedded processors, and general-purpose processors. Although early processors were created from discrete logic, a modern processor is implemented as a single VLSI chip.

A processor is classified as programmable if at some level, the processor hardware is separate from the sequence of steps that the processor performs; from the point of view of the end user, however, it might not be possible to change the program without replacing the processor. All programmable processors follow a fetch-execute cycle; the time required for one cycle depends on the operation performed. Because fetch-execute processing continues indefinitely, a designer must construct a program in such a way that the processor always has an instruction to execute.

A set of software programs are used to translate a source program, written by a programmer, into the binary representation that a processor requires. The set includes a preprocessor, compiler, assembler, and linker.

EXERCISES

4.1 Neither Figure 4.1 nor Figure 4.2 has *storage* as a major component. Where does storage (e.g., flash or an electro-mechanical disk) fit into the figures?

4.2 Consider the System-on-Chip (SoC) approach described in Chapter 2. Besides a processor, memory, and I/O facilities, what does an SoC need?

4.3 Consult Wikipedia to learn about early computers. How much memory did the Harvard Mark I computer have, and what year was it created? How much memory did the IBM 360/20 computer have, and what year was it created?

4.4 Although CPU manufacturers brag about the graphics accelerators on their chips, some video game designers choose to keep the graphics hardware separate from the processor. Explain one possible motivation for keeping it separate.

4.5 Imagine a smart phone that employs a Harvard Architecture. If you purchase such a phone, what would you need to specify that you do not normally specify?

4.6 What aspect of a Von Neumann Architecture makes it more vulnerable to hackers than a Harvard Architecture?

4.7 If you have access to *gcc*, read the man page to learn the command line argument that allows you to run only the preprocessor and place the preprocessed program in a file that can be viewed. What changes are made in the source program?

4.8 Extend the previous exercise by placing the assembly language output from the compiler in a file that can be viewed.

4.9 Write a computer program that compares the difference in execution times between an integer division and a floating point division. To test the program, execute each operation 100,000 times, and compare the difference in running times.

Chapter Contents

5

Processor Types And Instruction Sets

5.1 Introduction

The previous chapter introduces a variety of processors and explains the fetch-execute cycle that programmable processors use. This chapter continues the discussion by focusing on the set of operations that a processor can perform. The chapter explains various approaches computer architects have chosen, and discusses the advantages and disadvantages of each. The next chapters extend the discussion by describing the various ways processors access operands.

5.2 Mathematical Power, Convenience, And Cost

What operations should a processor offer? From a mathematical point of view, a wide variety of computational models provide equivalent computing power. In theory, as long as a processor offers a few basic operations, the processor has sufficient power to compute any computable function†.

Programmers understand that although only a minimum set of operations are necessary, a minimum is neither convenient nor practical. That is, the set of operations is designed for convenience rather than for mere functionality. For example, it is possible to compute a quotient by repeated subtraction. However, a program that uses repeated subtraction to compute a quotient runs slowly. Thus, most processors operations include hardware for each basic arithmetic operation: addition, subtraction, multiplication, and division.

†In a mathematical sense, only three operations are needed to compute any computable function: add one, subtract one, and branch if a value is nonzero.

To a computer architect, choosing a set of operations that the processor will perform represents a tradeoff. On the one hand, adding an additional arithmetic operation, such as multiplication or division, provides convenience for the programmer. On the other hand, each additional operation adds more hardware and makes the processor design more difficult. Adding hardware also increases engineering considerations such as chip size, power consumption, and heat dissipation. Thus, because a smart phone is designed to conserve battery power, the processors used in smart phones typically have fewer built-in operations than the processors used in powerful mainframe computers.

The point is that when considering the set of operations a given processor provides, we need to remember that the choice represents a complex tradeoff:

> *The set of operations a processor provides represents a tradeoff among the cost of the hardware, the convenience for a programmer, and engineering considerations such as power consumption.*

5.3 Instruction Set Architecture

When an architect designs a programmable processor, the architect must make two key decisions:

- Instruction set: the set of operations the processor provides
- Instruction representation: the format for each operation

We use the term *instruction set* to refer to the set of operations the hardware recognizes, and refer to each operation as an *instruction*. We assume that on each iteration of its fetch-execute cycle, a processor *executes* one instruction.

The definition of an instruction set specifies all details about instructions, including an exact specification of actions the processor takes when it executes the instruction. Thus, the instruction set defines values on which each instruction operates and the results the instruction produces. The definition specifies allowable values (e.g., the division instruction requires the divisor to be nonzero) and error conditions (e.g., what happens if an addition results in an overflow).

The term *instruction representation* (*instruction format*) refers to the binary representation that the hardware uses for instructions. The instruction representation is important because it defines a key interface: the interface between software that generates instructions and places them in memory and the hardware that executes the instructions. The software (e.g., the compiler, linker, and loader) must create an image in memory that uses exactly the same instruction format that the processor hardware expects.

We say that the definition of an instruction set and the corresponding representation define an *Instruction Set Architecture* (*ISA*). That is, an ISA defines both syntactic and semantic aspects of the instruction set. IBM Corporation pioneered the approach in the 1960s when it developed an ISA for its System/360 line of computers — with minor exceptions, all computers in the line shared the same basic instruction set, but individual models differed widely (approximately a 1:30 ratio) in the size of memory, processor speed, and cost.

5.4 Opcodes, Operands, And Results

Conceptually, each instruction contains three parts that specify: the exact operation to be performed, the value(s) to use, and where to place the result(s). The following paragraphs define the idea more precisely.

Opcode. The term *opcode* (short for *operation code*) refers to the exact operation to be performed. An opcode is a number; when the instruction set is designed, each operation must be assigned a unique opcode. For example, integer addition might be assigned opcode five, and integer subtraction might be assigned opcode twelve.

Operands. The term *operand* refers to a value that is needed to perform an operation. The definition of an instruction set specifies the exact number of operands for each instruction, and the possible values (e.g., the addition operation takes two signed integers).

Results. In some architectures, one or more of the operands specify where the processor should place results of an instruction (e.g., the result of an arithmetic operation); in others, the location of the result is determined automatically.

5.5 Typical Instruction Format

Each instruction is represented as a binary string. On most processors, an instruction begins with a field that contains the opcode, followed by fields that contain the operands. Figure 5.1 illustrates the general format.

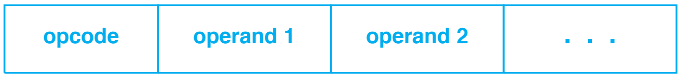


Figure 5.1 The general instruction format that many processors use. The opcode at the beginning of an instruction determines exactly which operands follow.

5.6 Variable-Length Vs. Fixed-Length Instructions

The question arises: should each instruction be the same size (i.e., occupy the same number of bytes) or should the length depend on the quantity and type of the operands? For example, consider integer arithmetic operations. Addition or subtraction operates on two values, but negation operates on a single value. Furthermore, a processor can handle multiple sizes of operands (e.g., a processor can have an instruction that adds a pair of sixteen-bit integers as well as an instruction that adds a pair of thirty-two bit integers). Should one instruction be shorter than another?

We use the term *variable-length* to characterize an instruction set that includes multiple instruction sizes, and the term *fixed-length* to characterize an instruction set in which every instruction is the same size. Programmers expect variable-length instructions because software usually allocates space according to the size of each object (e.g., if the strings “Hello” and “bye” appear in a program, a compiler will allocate 5 and 3 bytes, respectively). From a hardware point of view, however, variable-length instructions require complex hardware to fetch and decode. By comparison, fixed-length instructions require less complex hardware. Fixed-length instructions allow processor hardware to operate at higher speed because the hardware can compute the location of the next instruction easily. Thus, many processors force all instructions to be the same size, even if some instructions can be represented in fewer bits than others. The point is:

> *Although it may seem inefficient to a programmer, using fixed-length instructions can make processor hardware less complex and faster.*

How does a processor that uses fixed-length instructions handle cases where an instruction does not need all operands? For example, how does a fixed-length instruction set accommodate both addition and negation? Interestingly, the hardware is designed to ignore fields that are not needed for a given operation. Thus, an instruction set may specify that in some instructions, specific bits are *unused*†. To summarize:

> *When a fixed-length instruction set is employed, some instructions contain extra fields that the hardware ignores. The unused fields should be viewed as part of a hardware optimization, not as an indication of a poor design.*

5.7 General-Purpose Registers

As we have seen, a *register* is a small, high-speed hardware storage device found in a processor. A register has a fixed size (e.g., 32 or 64 bits) and supports two basic operations: *fetch* and *store*. We will see later that registers can operate in a variety of roles, including as an *instruction pointer* (also called a *program counter*) that gives the address of the next instruction to execute. For now, we will restrict our attention to a simple case that is well known to programmers: *general-purpose registers* that are used as a temporary storage mechanism. A processor usually has a small number of general-purpose registers (e.g., thirty-two), and each register is usually the size of an integer. For example, on a processor that provides thirty-two bit arithmetic, each general-purpose register holds thirty-two bits. As a result, a general-purpose register

†Some hardware requires unused bits to be zero.

can hold an operand needed for an arithmetic instruction or the result of such an instruction.

In many architectures, general-purpose registers are numbered from 0 through N−1. The processor provides instructions that can store a value into (or fetch a value from) a specified register. General-purpose registers have the same semantics as memory: a *fetch* operation returns the value specified in the previous *store* operation. Similarly, a *store* operation replaces the contents of the register with a new value.

5.8 Floating Point Registers And Register Identification

Processors that support floating point arithmetic often use a separate set of registers to hold floating point values. Confusion can arise because both general-purpose registers and floating point registers are usually numbered starting at zero — the instruction determines which registers are used. For example, if registers 3 and 6 are specified as operands for an integer instruction, the processor will extract the operands from the general-purpose registers. However, if registers 3 and 6 are specified as operands for a floating point instruction, the floating point registers will be used.

5.9 Programming With Registers

Many processors require operands to be placed in general-purpose registers before an instruction is executed. Some processors also place the results of an instruction in a general-purpose register. Thus, to add two integers in variables X and Y and place the result in variable Z, a programmer must create a series of instructions that move values to the corresponding registers. For example, if general-purpose registers 3, 6, and 7 are available, the program might contain four instructions that perform the following steps:

- Load a copy of variable X from memory into register 3
- Load a copy of variable Y from memory into register 6
- Add the value in register 3 to the value in register 6, and place the result in register 7
- Store a copy of the value in register 7 to variable Z in memory

We will see that moving a value between memory and a register is relatively expensive, so performance is optimized by leaving values in registers if the value will be used again. Because a processor only contains a small number of registers, a programmer (or compiler) must decide which values to keep in the registers at any time; other values are kept in memory†. The process of choosing which values the registers contain is known as *register allocation*.

Many details complicate register allocation. One of the most common arises if an instruction generates a large result, called an *extended value*. For example, integer multiplication can produce a result that contains twice as many bits as either operand.

†The term *register spilling* refers to moving a value from a register back into memory to make the register available for a new value.

Some processors offer facilities for *double precision* arithmetic (e.g., if a standard integer is thirty-two bits wide, a double precision integer occupies sixty-four bits).

To handle extended values, the hardware treats registers as consecutive. On such processors, for example, an instruction that loads a double precision integer into register 4 will place half the integer in register 4 and the other half in register 5 (i.e., the value of register 5 will change even though the instruction contains no explicit reference). When choosing registers to use, a programmer must plan for instructions that place extended data values in consecutive registers.

5.10 Register Banks

An additional hardware detail complicates register allocation: some architectures divide registers into multiple *banks*, and require the operands for an instruction to come from separate banks. For example, on a processor that uses two register banks, an integer *add* instruction may require the two operands to be from separate banks.

To understand register banks, we must examine the underlying hardware. In essence, register banks allow the hardware to operate faster because each bank has a separate physical access mechanism and the mechanisms operate simultaneously. Thus, when the processor executes an instruction that accesses two operands in registers, both operands can be obtained at the same time. Figure 5.2 illustrates the concept.

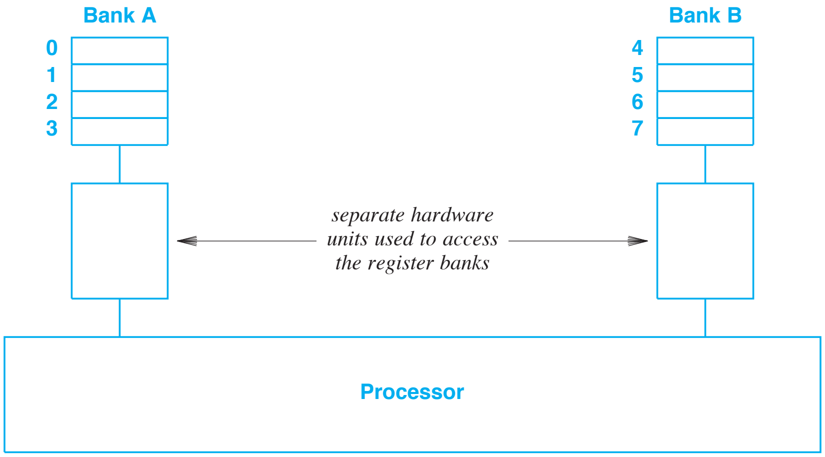


Figure 5.2 Illustration of eight registers divided into two banks. Hardware allows the processor to access both banks at the same time.

Register banks have an interesting consequence for programmers: it may not be possible to keep data values in registers permanently. To understand why, consider the following assignment statements that are typical of those used in a conventional programming language, and assume we want to implement the statements on a processor that has two register banks as Figure 5.2 illustrates.

```
R ← X + Y
S ← Z - X
T ← Y + Z
```

To perform the first addition, X and Y must be in separate register banks. Let's assume X is in a register in bank A, and Y is in a register in bank B. For the subtraction, Z must be in the opposite register bank than X (i.e., Z must be in a register in bank B). For the third assignment, Y and Z must be in different banks. Unfortunately, the first two assignments mean that Y and Z are located in the same bank. Thus, there is no possible assignment of X, Y, and Z to registers that works with all three instructions. We say that a *register conflict* occurs.

What happens when a register conflict arises? The programmer must either reassign registers or insert an instruction to copy values. For example, we could insert an extra instruction that copies the value of Z into a register in bank A before the final addition is performed.

5.11 Complex And Reduced Instruction Sets

Computer architects divide instruction sets into two broad categories that are used to classify processors†:

- Complex Instruction Set Computer (CISC)
- Reduced Instruction Set Computer (RISC)

A *CISC processor* usually includes many instructions (typically hundreds), and each instruction can perform an arbitrarily complex computation. Intel's x86 instruction set is classified as CISC because a processor provides hundreds of instructions, including complex instructions that require a long time to execute (e.g., one instruction manipulates graphics in memory and others compute the *sine* and *cosine* functions).

In contrast to CISC, a *RISC processor* is constrained. Instead of arbitrary instructions, a RISC design strives for a minimum set that is sufficient for all computation (e.g., thirty-two instructions). Instead of allowing a single instruction to compute an arbitrary function, each instruction performs a basic computation. To achieve the highest possible speed, RISC designs constrain instructions to be a fixed size. Finally, as the next section explains, a RISC processor is designed to execute an instruction in one clock cycle‡. Arm Limited and MIPS Corporation have each created RISC architectures with limited instructions that can be executed in one clock cycle. The ARM designs are especially popular in smart phones and other low-power devices.

We can summarize:

†Instead of using the full name, most engineers use the acronyms, which are pronounced *sisk* and *risk*.

‡Recall from Chapter 2 that a clock, which pulses at regular intervals, is used to control digital logic.

> *A processor is classified as CISC if the instruction set contains instructions that perform complex computations that can require long times; a processor is classified as RISC if it contains a small number of instructions that can each execute in one clock cycle.*

5.12 RISC Design And The Execution Pipeline

We said that a RISC processor executes one instruction per clock cycle. In fact, a more accurate version of the statement is: a RISC processor is designed so the processor can *complete* one instruction on each clock cycle. To understand the subtle difference, it is important to know how the hardware works. We said that a processor performs a fetch-execute cycle by first fetching an instruction and then executing the instruction. In fact, the processor divides the fetch-execute cycle into several steps, typically:

- Fetch the next instruction
- Decode the instruction and fetch operands from registers
- Perform the arithmetic operation specified by the opcode
- Perform memory read or write, if needed
- Store the result back to the registers

To enable high speed, RISC processors contain parallel hardware units that each perform one step listed above. The hardware is arranged in a multistage *pipeline*†, which means the results from one hardware unit are passed to the next hardware unit. Figure 5.3 illustrates a pipeline.

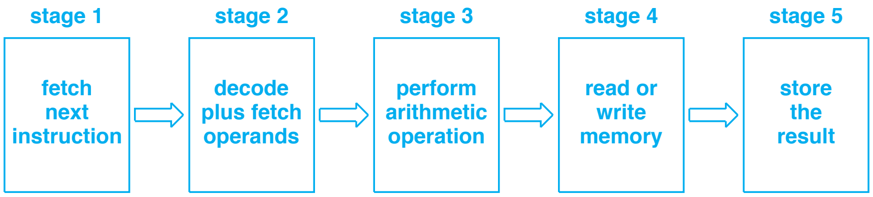
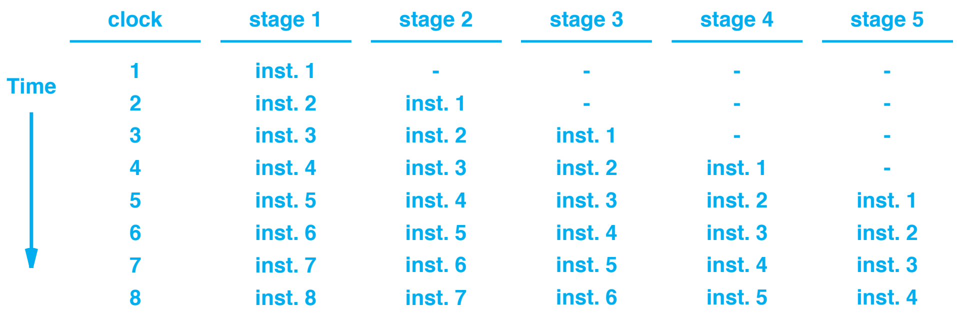


Figure 5.3 An example pipeline of the five hardware stages that are used to perform the fetch-execute cycle.

In the figure, an instruction moves left to right through the pipeline. The first stage fetches the instruction, the next stage examines the opcode, and so on. Whenever the clock ticks, all stages simultaneously pass the instruction to the right. Thus, instructions move through the pipeline like an assembly line: at any time, the pipeline contains five instructions.

†The terms *instruction pipeline* and *execution pipeline* are used interchangeably to refer to the multistage pipeline used in the fetch-execute cycle.

The speed of a pipeline arises because all stages can operate in parallel — while the fourth stage executes an instruction, the third stage fetches the operands for the next instruction. Thus, a stage never needs to delay because an instruction is always ready on each clock cycle. Figure 5.4 illustrates how a set of instructions pass through a five-stage pipeline.



clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. 1	-	-	-	-
2	inst. 2	inst. 1	-	-	-
3	inst. 3	inst. 2	inst. 1	-	-
4	inst. 4	inst. 3	inst. 2	inst. 1	-
5	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1
6	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2
7	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3
8	inst. 8	inst. 7	inst. 6	inst. 5	inst. 4

Figure 5.4 Instructions passing through a five-stage pipeline. Once the pipeline is filled, each stage is busy on each clock cycle.

The figure clearly illustrates that although a RISC processor cannot perform all the steps needed to fetch and execute an instruction in one clock cycle, parallel hardware allows the processor to finish one instruction per clock cycle. We can summarize:

> *Although a RISC processor cannot perform all steps of the fetch-execute cycle in a single clock cycle, an instruction pipeline with parallel hardware provides approximately the same performance: once the pipeline is full, one instruction completes on every clock cycle.*

5.13 Pipelines And Instruction Stalls

We say that the instruction pipeline is *transparent* to programmers because the instruction set does not contain any explicit references to the pipeline. That is, the hardware is constructed so the results of a program are the same whether or not a pipeline is present. Although transparency can be an advantage, it can also be a disadvantage: a programmer who does not understand the pipeline can inadvertently introduce inefficiencies.

To understand the effect of programming choices on a pipeline, consider a program that contains two successive instructions that perform an addition and subtraction on operands and results located in registers that we will label A, B, C, D, and E:

Instruction K: C ← add A B
Instruction K+1: D ← subtract E C

Although instruction K can proceed through the pipeline from beginning to end, instruction K+1 encounters a problem because operand C is not available in time. That is, the hardware must wait for instruction K to finish before fetching the operands for instruction K+1. We say that a stage of the pipeline *stalls* to wait for the operand to become available. Figure 5.5 illustrates what happens during a pipeline stall.

	clock	stage 1 fetch instruction	stage 2 fetch operands	stage 3 ALU operation	stage 4 access memory	stage 5 write results
	1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4
Time	2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3
	3	inst. K+2	(inst. K+1)	inst. K	inst. K-1	inst. K-2
	4	(inst. K+2)	(inst. K+1)	–	inst. K	inst. K-1
	5	(inst. K+2)	(inst. K+1)	–	–	inst. K
	6	(inst. K+2)	inst. K+1	–	–	–
	7	inst. K+3	inst. K+2	inst. K+1	–	–
	8	inst. K+4	inst. K+3	inst. K+2	inst. K+1	–
	9	inst. K+5	inst. K+4	inst. K+3	inst. K+2	inst. K+1
	10	inst. K+6	inst. K+5	inst. K+4	inst. K+1	inst. K+2

Figure 5.5 Illustration of a pipeline stall. Instruction K+1 cannot proceed until an operand from instruction K becomes available.

The figure shows a normal pipeline running until clock cycle 3, when Instruction K+1 has reached stage 2. Recall that stage 2 fetches operands from registers. In our example, one of the operands for instruction K+1 is not available, and will not be available until instruction K writes its results into a register. The pipeline must stall until instruction K completes. In the code above, because the value of C has not been computed, stage 2 cannot fetch the value for C. Thus, stages 1 and 2 remain stalled during clock cycles 4 and 5. During clock cycle 6, stage 2 can fetch the operand, and pipeline processing continues.

The rightmost column in Figure 5.5 shows the effect of a stall on performance: the final stage of the pipeline does not produce any results during clock cycles 6, 7, and 8. If no stalls had occurred, instruction K+1 would have completed during clock cycle 6, but the stall means the instruction does not complete until clock cycle 9.

To describe the delay between the cause of a stall and the time at which output stops, we say that a *bubble* passes through the pipeline. Of course, the bubble is only apparent to someone observing the pipeline's performance because correctness is not affected. That is, an instruction always passes directly to the next stage as soon as one stage completes, which means all instructions are executed in the order specified.

5.14 Other Causes Of Pipeline Stalls

In addition to waiting for operands, a pipeline can stall when the processor executes any instruction that delays processing or disrupts the normal flow. For example, a stall can occur when a processor:

- Accesses external storage
- Invokes a coprocessor
- Branches to a new location
- Calls a subroutine

The most sophisticated processors contain additional hardware to avoid stalls. For example, some processors contain two copies of a pipeline, which allows the processor to start decoding the instruction that will be executed if a branch is taken as well as the instruction that will be executed if a branch is not taken. The two copies operate until a branch instruction can be executed. At that time, the hardware knows which copy of the pipeline to follow; the other copy is ignored. Other processors contain special shortcut hardware that passes a copy of a result back to a previous pipeline stage.

5.15 Consequences For Programmers

To achieve maximum speed, a program for a RISC architecture must be written to accommodate an instruction pipeline. For example, a programmer should avoid introducing unnecessary branch instructions. Similarly, instead of referencing a result register immediately in the following instruction, the reference can be delayed. As an example, Figure 5.6 shows how code can be rearranged to run faster.

```
C ← add A B              C ← add A B
D ← subtract E C         F ← add G H
F ← add G H              M ← add K L
J ← subtract I F         D ← subtract E C
M ← add K L              J ← subtract I F
P ← subtract M N         P ← subtract M N
      (a)                      (b)
```

Figure 5.6 (a) A list of instructions, and (b) the instructions reordered to run faster in a pipeline. Reducing pipeline stalls increases speed.

In the figure, the optimized program separates references from computation. For example, in the original program, the second instruction references value C, which is produced by the previous instruction. Thus, a stall occurs between the first and second instructions. Moving the subtraction to a later point in the program allows the processor to continue to operate without a stall.

Of course, a programmer can choose to view a pipeline as an automatic optimization instead of a programming burden. Fortunately, most programmers do not need to perform pipeline optimizations manually. Instead, compilers for high-level languages perform the optimizations automatically.

> *Rearranging code sequences can increase the speed of processing when the hardware uses an instruction pipeline; programmers view the reordering as an optimization that can increase speed without affecting correctness.*

5.16 Programming, Stalls, And No-Op Instructions

In some cases, the instructions in a program cannot be rearranged to prevent a stall. In such cases, programmers can document stalls so anyone reading the code will understand that a stall occurs. Such documentation is especially helpful if a program is modified because the programmer who performs the modification can reconsider the situation and attempt to reorder instructions to prevent a stall.

How should programmers document a stall? One technique is obvious: insert a comment that explains the reason for a stall. However, most code is generated by compilers and is only read by humans when problems occur or special optimization is required. In such cases, another technique can be used: in places where stalls occur, insert extra instructions in the code. The extra instructions show where items can be inserted without affecting the pipeline. Of course, the extra instructions must be innocuous — they must not change the values in registers or otherwise affect the program. In most cases, the hardware provides the solution: a *no-op*. That is, an instruction that does absolutely nothing except occupy time. The point is:

> *Most processors include a no-op instruction that does not reference data values, compute a result, or otherwise affect the state of the computer. No-op instructions can be inserted to document locations where an instruction stall occurs.*

5.17 Forwarding

As mentioned above, some hardware has special facilities to improve instruction pipeline performance. For example, an ALU can use a technique known as *forwarding* to solve the problem of successive arithmetic instructions passing results.

To understand how forwarding works, consider the example of two instructions where operands *A*, *B*, *C*, *D*, and *E* are in registers:

Instruction K:	C ← add A B
Instruction K+1:	D ← subtract E C

We said that such a sequence causes a stall on a pipelined processor. However, a processor that implements forwarding can avoid the stall by arranging for the hardware to detect the dependency and automatically pass the value for *C* from instruction *K* directly to instruction *K + 1*. That is, a copy of the output from the ALU in instruction *K* is forwarded directly to the input of the ALU in instruction *K + 1*. As a result, instructions continue to fill the pipeline, and no stall occurs.

5.18 Types Of Operations

When computer architects discuss instruction sets, they divide the instructions into a few basic categories. Figure 5.7 lists one possible division.

- Integer arithmetic instructions
- Floating point arithmetic instructions
- Logical instructions (also called Boolean operations)
- Data access and transfer instructions
- Conditional and unconditional branch instructions
- Processor control instructions
- Graphics instructions

Figure 5.7 An example of categories used to classify instructions. A general-purpose processor includes instructions in all the categories.

5.19 Program Counter, Fetch-Execute, And Branching

Recall from Chapter 4 that every processor implements a basic fetch-execute cycle. During the cycle, control hardware in the processor automatically moves through instructions — once it finishes executing one instruction, the processor automatically moves past the current instruction in memory before fetching the next instruction. To implement the fetch-execute cycle and a move to the next instruction, the processor uses a special-purpose internal register known as an *instruction pointer* or *program counter*†.

When a fetch-execute cycle begins, the program counter contains the address of the instruction to be executed. After an instruction has been fetched, the program counter is updated to the address of the next instruction. The update of the program counter during each fetch-execute cycle means the processor will automatically move through successive instructions in memory. Algorithm 5.1 specifies how the fetch-execute cycle moves through successive instructions.

Algorithm 5.1

```
Assign the program counter an initial program address.  Repeat
forever {

        Fetch: access the next step of the program from the loca-
        tion given by the program counter.

        Set an internal address register, A, to the address beyond
        the instruction that was just fetched.

        Execute: Perform the step of the program.

        Copy the contents of address register A to the program
        counter.

}
```

Algorithm 5.1 The Fetch-Execute Cycle

The algorithm allows us to understand how branch instructions work. There are two cases: absolute and relative. An *absolute branch* computes a memory address, and the address specifies the location of the next instruction to execute. Typically, an absolute branch instruction is known as a *jump*. During the execute step, a *jump* instruction computes an address and loads the value into the internal register *A* that Algorithm 5.1 specifies. At the end of the fetch-execute cycle, the hardware copies the value into the program counter, which means the address will be used to fetch the next instruction. For example, the absolute branch instruction:

†The two terms are equivalent.

```
jump 0x05DE
```

causes the processor to load *0x05DE* into the internal address register, which is copied into the program counter before the next instruction is fetched. In other words, the next instruction fetch will occur at memory location *0x05DE*.

Unlike an absolute branch instruction, a *relative branch instruction* does not specify an exact memory address. Instead, a relative branch computes a positive or negative increment for the program counter. For example, the instruction:

```
br +8
```

specifies branching to a location that is eight bytes beyond the current location (i.e., beyond the current value of the program counter).

To implement relative branching, a processor adds the operand in the branch instruction to the program counter, and places the result in internal address register *A*. For example, if the relative branch computes *-12*, the next instruction to be executed will be found at an address twelve bytes *before* the current instruction. A compiler might use a relative branch at the end of a short *while-loop*.

Most processors also provide an instruction to invoke a subroutine, typically *jsr* (*jump subroutine*). In terms of the fetch-execute cycle, a *jsr* instruction operates like a branch instruction with a key difference: before the branch occurs, the *jsr* instruction saves the value of the address register, *A*. When it finishes executing, a subroutine returns to the caller. To do so, the subroutine executes an absolute branch to the saved address. Thus, when the subroutine finishes, the fetch-execute cycle resumes at the instruction immediately following the *jsr*.

5.20 Subroutine Calls, Arguments, And Register Windows

High-level languages use a subroutine call instruction, such as *jsr*, to implement a procedure or function call. The calling program supplies a set of *arguments* that the subroutine uses in its computation. For example, the function call *cos(3.14159)* has the floating point constant *3.14159* as an argument.

One of the principal differences among processors arises from the way the underlying hardware passes arguments to a subroutine. Some architectures use memory — the arguments are stored on the stack in memory before the call, and the subroutine extracts the values from the stack when they are referenced. In other architectures, the processor uses either general-purpose or special-purpose registers to pass arguments.

Using either special-purpose or general-purpose registers to pass arguments is much faster than using a stack in memory because registers are part of the local storage in the processor itself. Because few processors provide special-purpose registers for argument passing, general-purpose registers are typically used. Unfortunately, general-purpose registers cannot be devoted exclusively to arguments because they are also needed for other computation (e.g., to hold operands for arithmetic operations). Thus, a programmer faces a tradeoff: using a general-purpose register to pass an argument can increase the speed of a subroutine call, but using the register to hold a data value can increase the speed of general computation. Thus, a programmer must choose which arguments to keep in registers and which to store in memory†.

Some processors include an optimization for argument passing known as a *register window*. Although a processor has a large set of general-purpose registers, the register hardware only exposes a subset of the registers at any time. The subset is known as a *window*. The window moves automatically each time a subroutine is invoked, and moves back when the subroutine returns. More important, the windows that are available to a program and subroutine overlap — some of the registers visible to the caller are visible to the subroutine. A caller places arguments in the registers that will overlap before calling a subroutine and the subroutine extracts values from the registers. Figure 5.8 illustrates the concept of a register window

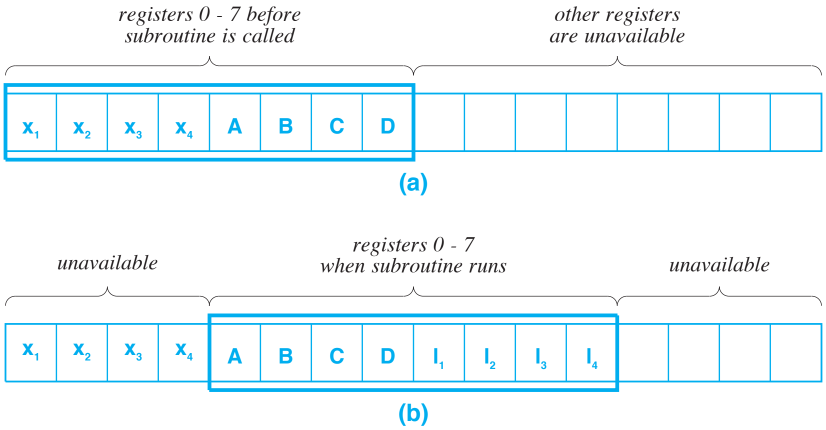

registers 0 - 7 before subroutine is called
other registers are unavailable
x_1
x_2
x_3
x_4
A
B
C
D
(a)
unavailable
registers 0 - 7 when subroutine runs
unavailable
x_1
x_2
x_3
x_4
A
B
C
D
l_1
l_2
l_3
l_4
(b)

Figure 5.8 Illustration of a register window (a) before a subroutine call, and (b) during the call. Values *A*, *B*, *C*, and *D* correspond to arguments that are passed.

In the figure, the hardware has 16 registers, but only 8 registers are visible at any time; others are unavailable. A program always references visible registers as numbers 0 through the window size minus one (0 through 7 in the example). When a subroutine is called, the hardware changes the set of registers that are visible by *sliding* the window. In the example, registers that were numbered 4 through 7 before the call become 0 through 3 after the call. Thus, the calling program places arguments *A* through *D* in registers 4 through 7, and the subroutine finds the arguments in registers 0 through 3. Registers with values x_i are only available to the calling program. The advantage of a register window approach is that registers that are not in the current window retain the values they had. So, when the called subroutine returns, the window will slide back, and registers with values x_i will be exactly the same as before the call.

†Appendix 3 describes the calling sequence used with an x86 architecture, and Appendix 4 explains how an ARM architecture passes some arguments in registers and some in memory.

The illustration in Figure 5.8 uses a small window size (eight registers) to simplify the diagram. In practice, processors that use a register window typically have larger windows. For example, the Sparc architecture has one hundred twenty-eight or one hundred forty-four physical registers and a window size of thirty-two registers; however, only eight of the registers in the window overlap (i.e., only eight registers can be used to pass arguments).

5.21 An Example Instruction Set

An example instruction set will help clarify the concepts described above. We have selected the MIPS processor as an example for two reasons. First, the MIPS processor is popular for use in embedded systems. Second, the MIPS instruction set is a classic example of the instruction set offered by a RISC processor. Figure 5.9 lists the instructions in the MIPS instruction set.

A MIPS processor contains thirty-two general-purpose registers, and most instructions require the operands and results to be in registers. For example, the *add* instruction takes three operands that are registers: the instruction adds the contents of the first two registers and places the result in the third.

In addition to the integer instructions that are listed in Figure 5.9, the MIPS architecture defines a set of floating point instructions for both single precision (i.e., thirty-two bit) and double precision (i.e., sixty-four bit) floating point values. The hardware provides a set of thirty-two floating point registers. Although they are numbered from zero to thirty-one, the floating point registers are completely independent of the general-purpose registers.

To handle double precision values, the floating point registers operate as pairs. That is, only an even-numbered floating point register can be specified as an operand or target in a floating point instruction — the hardware uses the specified register plus the next odd-numbered register as a combined storage unit to hold a double precision value. Figure 5.10 summarizes the MIPS floating point instruction set.

Instruction	Meaning
Arithmetic	
add	integer addition
subtract	integer subtraction
add immediate	integer addition (register + constant)
add unsigned	unsigned integer addition
subtract unsigned	unsigned integer subtraction
add immediate unsigned	unsigned addition with a constant
move from coprocessor	access coprocessor register
multiply	integer multiplication
multiply unsigned	unsigned integer multiplication
divide	integer division
divide unsigned	unsigned integer division
move from Hi	access high-order register
move from Lo	access low-order register
Logical (Boolean)	
and	logical *and* (two registers)
or	logical *or* (two registers)
and immediate	*and* of register and constant
or immediate	*or* of register and constant
shift left logical	shift register left N bits
shift right logical	shift register right N bits
Data Transfer	
load word	load register from memory
store word	store register into memory
load upper immediate	place constant in upper sixteen bits of register
move from coproc. register	obtain a value from a coprocessor
Conditional Branch	
branch equal	branch if two registers equal
branch not equal	branch if two registers unequal
set on less than	compare two registers
set less than immediate	compare register and constant
set less than unsigned	compare unsigned registers
set less than immediate	compare unsigned register and constant
Unconditional Branch	
jump	go to target address
jump register	go to address in register
jump and link	procedure call

Figure 5.9 An example instruction set. The table lists the instructions offered by the MIPS processor.

Instruction	Meaning
Arithmetic	
FP add	**floating point addition**
FP subtract	**floating point subtraction**
FP multiply	**floating point multiplication**
FP divide	**floating point division**
FP add double	**double-precision addition**
FP subtract double	**double-precision subtraction**
FP multiply double	**double-precision multiplication**
FP divide double	**double-precision division**
Data Transfer	
load word coprocessor	**load value into FP register**
store word coprocessor	**store FP register to memory**
Conditional Branch	
branch FP true	**branch if FP condition is true**
branch FP false	**branch if FP condition is false**
FP compare single	**compare two FP registers**
FP compare double	**compare two double precision values**

Figure 5.10 Floating point (FP) instructions defined by the MIPS architecture. Double precision values occupy two consecutive floating point registers.

5.22 Minimalistic Instruction Set

It may seem that the instructions listed in Figure 5.9 are insufficient and that additional instructions are needed. For example, the MIPS architecture does not include an instruction that copies the contents of a register to another register, nor does the architecture include instructions that can add a value in memory to the contents of a register. To understand the choices, it is important to know that the MIPS instruction set supports two principles: speed and minimalism. First, the basic instruction set has been designed carefully to ensure high speed (i.e., the architecture has the property that when a pipeline is used, one instruction can complete on every clock cycle). Second, the instruction set is *minimalistic* — it contains the fewest possible instructions that handle standard computation. Limiting the number of instructions forms a key piece of the design. Choosing thirty-two instructions means that an opcode only needs five bits and no combination of the bits is wasted.

One feature of the MIPS architecture, which is also used in other RISC processors, helps achieve minimalism: fast access to a zero value. In the case of MIPS, register 0 provides the mechanism — the register is reserved and always contains the value zero.

Thus, to test whether a register is zero, the value can be compared to register zero. Similarly, register zero can be used in any instruction. For example, to copy a value from one register to another, an *add* instruction can be used in which one of the two operands is register zero.

5.23 The Principle Of Orthogonality

In addition to the technical aspects of instruction sets discussed above, an architect must consider the aesthetic aspects of a design. In particular, an architect strives for *elegance*. Elegance relates to human perception: how does the instruction set appear to a programmer? How do instructions combine to handle common programming tasks? Are the instructions balanced (if the set includes right-shift, does it also include left-shift)? Elegance calls for subjective judgment. However, experience with a few instruction sets often helps engineers and programmers recognize and appreciate elegance.

One particular aspect of elegance, known as *orthogonality*, concentrates on eliminating unnecessary duplication and overlap among instructions. We say that an instruction set is *orthogonal* if each instruction performs a unique task. An orthogonal instruction set has important advantages for programmers: orthogonal instructions can be understood more easily, and a programmer does not need to choose among multiple instructions that perform the same task. *Orthogonality* is so important that it has become a general principle of processor design. We can summarize:

> *The* principle of orthogonality *specifies that each instruction should perform a unique task without duplicating or overlapping the functionality of other instructions.*

5.24 Condition Codes And Conditional Branching

On many processors, executing an instruction results in a *status*, which the processor stores in an internal hardware mechanism. A later instruction can use the status to decide how to proceed. For example, when it executes an arithmetic instruction, the ALU sets an internal register known as a *condition code* that contains bits to record whether the result is positive, negative, zero, or an arithmetic overflow occurred. A *conditional branch* instruction that follows the arithmetic operation can test one or more of the condition code bits, and use the result to determine whether to branch.

An example will clarify how a condition code mechanism is used†. To understand the paradigm, consider a program that tests for equality between two values. As a simple example, suppose the goal is to set register 3 to zero if the contents of register 4 are not equal to the contents of register 5. Figure 5.11 contains example code.

†Chapter 9 explains programming with condition codes and shows further examples.

```
        cmp   r4, r5    # compare regs. 4 & 5, and set condition code
        be    lab1      # branch to lab1 if cond. code specifies equal
        mov   r3, 0     # place a zero in register 3
  lab1: ...program continues at this point
```

Figure 5.11 An example of using a condition code. An ALU operation sets the condition code, and a later *conditional branch* instruction tests the condition code.

5.25 Summary

Each processor defines an instruction set that consists of operations the processor supports; the set is chosen as a compromise between programmer convenience and hardware efficiency. In some processors, each instruction is the same size, and in other processors size varies among instructions.

Most processors include a small set of general-purpose registers that are high-speed storage mechanisms. To program using registers, one loads values from memory into registers, performs a computation, and stores the result from a register into memory. To optimize performance, a programmer leaves values that will be used again in registers. On some architectures, registers are divided into banks, and a programmer must ensure that the operands for each instruction come from separate banks.

Processors can be classified into two broad categories of CISC and RISC depending on whether they include many complex instructions or a minimal set of instructions. RISC architectures use an instruction pipeline to ensure that one instruction can complete on each clock cycle. Programmers can optimize performance by rearranging code to avoid pipeline stalls.

To implement conditional execution (e.g., an *if-then-else*), many processors rely on a condition code mechanism — an ALU instruction sets the condition code, and a later instruction (a conditional branch) tests the condition code.

EXERCISES

5.1 When debugging a program, a programmer uses a tool that allows them to show the contents of memory. When the programmer points the tool to a memory location that contains an instruction, the tool prints three hex values with labels:

OC=0x43 OP1=0xff00 OP2=0x0324

What do the labels abbreviate?

5.2 If the arithmetic hardware on a computer requires operands to be in separate banks, what instruction sequence will be needed to compute the following?

```
A ← B - C
Q ← A * C
W ← Q + A
Z ← W - Q
```

5.3 Assume you are designing an instruction set for a computer that will perform the Boolean operations *and*, *or*, *not*, and *exclusive or*. Assign opcodes and indicate the number of operands for each instruction. When your instructions are stored in memory, how many bits will be needed to hold the opcode?

5.4 If a computer can add, subtract, multiply, and divide 16-bit integers, 32-bit integers, 32-bit floating point values, and 64-bit floating point values, how many unique opcodes will be needed? (Hint: assume one op code for each operation and each data size.)

5.5 A computer architect boasted that they were able to design a computer in which every instruction occupied exactly thirty-two bits. What is the advantage of such a design?

5.6 Classify the ARM architecture owned by ARM Limited, the SPARC architecture owned by Oracle Corporation, and the Intel Architecture owned by Intel Corporation as CISC or RISC.

5.7 Consider a pipeline of N stages in which stage *i* takes time t_i. Assuming no delay between stages, what is the total time (start to finish) that the pipeline will spend handling a single instruction?

5.8 Insert *nop* instructions in the following code to eliminate pipeline stalls (assume the pipeline illustrated in Figure 5.5).

```
loadi    r7, 10          # put 10 in register 7
loadi    r8, 15          # put 15 in register 8
loadi    r9, 20          # put 20 in register 5
addrr    r10, r7, r8     # add registers 7 8; put the result in register 10
movr     r12, r9         # copy register 9 to register 12
movr     r11, r7         # copy register 7 to register 11
addri    r14, r11, 27    # add 27 plus register 11; put the result in register 14
addrr    r13, r12, r11   # add registers 11 and 12; put the results in register 13
```

Chapter Contents

6

Data Paths And Instruction Execution

6.1 Introduction

Chapter 2 introduces digital logic and describes the basic hardware building blocks that are used to create digital systems. The chapter covers basic gates, and shows how gates are constructed from transistors. The chapter also describes the important concept of a clock, and demonstrates how a clock allows a digital circuit to perform a series of operations. Successive chapters describe how data is represented in binary and cover processors and instruction sets.

This chapter explains how digital logic circuits can be combined to construct a computer. The chapter reviews functional units, such as arithmetic-logic units and memory, and shows how the units are interconnected. Finally, the chapter explains how the units interact to perform computation. Later sections of the text expand the discussion by examining processors and memory systems in more detail.

6.2 Data Paths

The topic of how hardware can be organized to create a programmable computer is complex. Rather than look at all the details of a large design, architects begin by describing the major hardware components and their interconnection. At a high level, we are only interested in how instructions are read from memory and how an instruction is executed. Therefore, the high-level description ignores many details and only shows the interconnections across which data items move as instructions are executed. For example, when we consider the addition operation, we will see the data paths across which two operands travel to reach an *Arithmetic Logic Unit* (*ALU*) and a data path that carries the result to another unit. Our diagrams will not show other details, such as power and ground connections or control connections. Computer architects use the terms *data paths* to describe the idea and *data path diagram* to describe a figure that depicts the data paths.

To make the discussion of data paths clear, we will examine a simplified computer. The simplifications include:

- Our instruction set only contains four instructions
- We assume a program has already been loaded into memory
- We ignore startup and assume the processor is running
- We assume each data item and each instruction occupies exactly 32 bits
- We only consider integer arithmetic
- We completely ignore error conditions, such as arithmetic overflow

Although the example computer is extremely simple, the basic hardware units we examine are exactly the same as a conventional computer. Thus, the example is sufficient to illustrate the main hardware components, and the example interconnection is sufficient to illustrate how data paths are designed.

6.3 The Example Instruction Set

As the previous chapter describes, a new computer design must begin with the design of an *instruction set*. Once the details of instructions have been specified, a computer architect can design hardware that performs each of the instructions. To illustrate how hardware is organized, we will consider an imaginary computer that has the following properties:

- A set of sixteen *general-purpose registers*†
- A memory that holds instructions (i.e., a program)
- A separate memory that holds data items

Each register can hold a thirty-two bit integer value. The *instruction memory* contains a sequence of instructions to be executed. As described above, we ignore startup, and assume a program has already been placed in the instruction memory. The *data memory* holds data values. We will also assume that both memories on the computer are byte-addressable, which means that each byte of memory is assigned an address.

Figure 6.1 lists the four basic instructions that our imaginary computer implements.

†Hardware engineers often use the term *register file* to refer to the hardware unit that implements a set of registers; we will simply refer to them as *registers*.

Instruction	Meaning
add	Add the integers in two registers and place the result in a third register
load	Load an integer from the data memory into a register
store	Store the integer in a register into the data memory
jump	Jump to a new location in the instruction memory

Figure 6.1 Four example instructions, the operands each uses, and the meaning of the instruction.

The *add* instruction is the easiest to understand — the instruction obtains integer values from two registers, adds the values together, and places the result in a third register. For example, consider an *add* instruction that specifies adding the contents of registers 2 and 3 and placing the result in register 4. If register 2 contains 50 and register 3 contains 60, such an *add* instruction will place 110 in register 4 (i.e., the sum of the integers in registers 2 and 3).

In assembly language, such an instruction is specified by giving the instruction name followed by operands. For example, a programmer might code the *add* instruction described in the previous paragraph by writing:

```
add     r4, r2, r3
```

where the notation *rX* is used to specify register X. The first operand specifies the *destination register* (where the result should be placed), and the other two specify *source registers* (where the instruction obtains the values to sum).

The *load* and *store* instructions move values between the data memory and the registers. Like many commercial processors, our imaginary processor requires both operands of an *add* instruction to be in registers. Also like commercial computers, our imaginary processor has a large data memory, but only a few registers. Consequently, to add two integers that are in memory, the two values must be loaded into registers. The *load* instruction makes a copy of an integer in memory and places the copy in a register. The *store* instruction moves data in the opposite direction: it makes a copy of the value currently in a register and places the copy in an integer in memory.

One of the operands for a *load* or *store* specifies the register to be loaded or stored. The other operand is more interesting because it illustrates a feature found on many commercial processors: a single operand that combines two values. Instead of using a single constant to specify a memory address, memory operands contain two parts. One part specifies a register, and the other part specifies a constant that is often called an *offset*. When the instruction is executed, the processor reads the current value from the specified register, adds the offset, and uses the result as a memory address.

An example will clarify the idea. Consider a *load* instruction that loads register 1 from a value in memory. Such an instruction might be written as:

```
load    r1, 20(r3)
```

where the first operand specifies that the value should be loaded into register 1. The second operand specifies that the memory address is computed by adding the offset 20 to the current contents of register 3.

Why are processors designed with operands that specify a register plus an offset? Using such a form makes it easy and efficient to iterate through an array. The address of the first element is placed in a register, and bytes of the element can be accessed by using the constant part of the operand. To move to the next element of the array, the register is incremented by the element size. For now, we only need to understand that such operands are used, and consider how to design hardware that implements them.

As an example, suppose register 3 contains the value 10000, and the *load* instruction shown above specifies an offset of 20. When the instruction is executed, the hardware adds 10000 and 20, treats the result as a memory address, and loads the integer from location 10020 into register 1.

The fourth instruction, a *jump* controls the flow of execution by giving the processor an address in the instruction memory. Normally, our imaginary processor works like an ordinary processor by executing an instruction and then moving to the next instruction in memory automatically. When it encounters a *jump* instruction, however, the processor does not move to the next instruction. Instead, the processor uses the operand in the *jump* instruction to compute a memory address, and then starts executing at that address.

Like the *load* and *store* instructions, our *jump* instruction allows both a register and offset to be specified in its operand. For example, the instruction

```
jump    60(r11)
```

specifies that the processor should obtain the contents of register 11, add 60, treat the result as an address in the instruction memory, and make the address the next location where an instruction is executed. It is not important now to understand why processors contain a *jump* instruction — you only need to understand how the hardware handles the move to a new location in a program.

6.4 Instructions In Memory

We said that the instruction memory on our imaginary computer contains a set of instructions for the processor to execute, and that each instruction occupies thirty-two bits. A computer designer specifies the exact format of each instruction by specifying what each bit means. Figure 6.2 shows the instruction format for our imaginary computer.

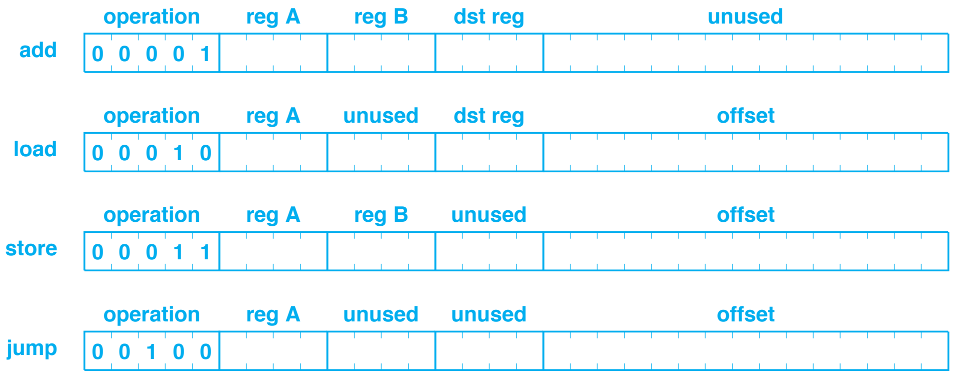


Figure 6.2 The binary representation for each of the four instructions listed in Figure 6.1. Each instruction is thirty-two bits long.

Look carefully at the fields used in each instruction. Each instruction has exactly the same format, even though some of the fields are not needed in some instructions. A uniform format makes it easy to design hardware that extracts the fields from an instruction.

The *operation* field in an instruction (sometimes called an *opcode* field) contains a value that specifies the operation. For our example, an *add* instruction has the operation field set to 1, a *load* instruction has the operation field set to 2, and so on. Thus, when it picks up an instruction, the hardware can use the operation field to decide which operation to perform.

The three fields with the term *reg* in their name specify three registers. Only the *add* instruction needs all three registers; in other instructions, one or two of the register fields are not used. The hardware ignores the unused fields when executing an instruction other than *add*.

The order of operands in the instructions may seem unexpected and inconsistent with the code above. For example, the code for an *add* instruction has the destination (the register to contain the result) on the left, and the two registers to be added on the right. In the instruction, fields that specify the two registers to be added precede the field that specifies the destination. Figure 6.3 shows a statement written by a programmer and the instruction when it has been converted to bits in memory. We can summarize the point:

The order of operands in an assembly language program is chosen to be convenient to a programmer; the order of operands in an instruction in memory is chosen to make the hardware efficient.

```
add  r4, r2, r3
```

(a)

operation	reg A	reg B	dst reg	offset
0 0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(b)

Figure 6.3 (a) An example *add* instruction as it appears to a programmer, and (b) the instruction stored in memory.

In the figure, the field labeled *reg A* contains 2 to specify register 2, the field labeled *reg B* contains 3 to specify register 3, and the field labeled *dst reg* contains 4 to specify that the result should be placed in register 4.

When we examine the hardware, we will see that the binary representation used for instructions is not capricious — the format is chosen to simplify the hardware design. For example, if an instruction has an operand that specifies a memory address, the register in the operand is always assigned to the field labeled *reg A*. Thus, if the hardware must add the offset to a register, the register is always found in field *reg A*. Similarly, if a value must be placed in a register, the register is found in field *dst reg*.

6.5 Moving To The Next Instruction

Chapter 2 illustrates how a clock can be used to control the timing of a fixed sequence of steps. Building a computer requires one additional twist: instead of a fixed sequence of steps, a computer is *programmable* which means that although the computer has hardware to perform every possible instruction, the exact sequence of instructions to perform is not predetermined. Instead, a programmer stores a program in memory, and the processor moves through the memory, extracting and executing successive instructions one at a time. The next sections illustrate how digital logic circuits can be arranged to enable programmability.

What pieces of hardware are needed to execute instructions from memory? One key element is known as an *instruction pointer*. An instruction pointer consists of a register (i.e., a set of latches) in the processor that holds the memory address of the next instruction to execute. For example, if we imagine a computer with thirty-two-bit memory addresses, an instruction pointer will hold a thirty-two-bit value. To execute instructions, the hardware repeats the following three steps.

- Use the instruction pointer as a memory address and fetch an instruction
- Use bits in the instruction to control hardware that performs the operation
- Move the instruction pointer to the next instruction

One of the most important aspects of a processor that can execute instructions arises from the mechanism used to move to the next instruction. After it extracts an instruction from the instruction memory, the processor must compute the memory address of the instruction that immediately follows. Thus, once a given instruction has executed, the processor is ready to execute the next sequential instruction.

In our example computer, each instruction occupies thirty-two bits in memory. However, the memory is byte-addressable, which means that after an instruction is executed, hardware must increment the instruction pointer by four bytes (thirty two bits) to move to the next instruction. In essence, the processor must add four to the instruction pointer and place the result back in the instruction pointer. To perform the computation, the constant 4 and the current instruction pointer value are passed to a thirty-two-bit adder. Figure 6.4 illustrates the basic components used to increment an instruction pointer and shows how the components are interconnected.

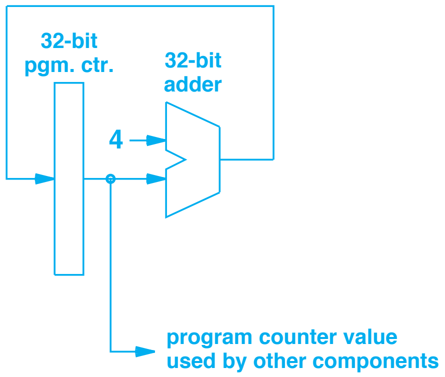


Figure 6.4 Hardware that increments a program counter.

The circuit in the figure appears to be an infinite loop that will simply run wild incrementing the program counter continuously. To understand why the circuit works, recall that a clock is used to control and synchronize digital circuits. In the case of the program counter, the clock only lets the increment occur after an instruction has executed. Although no clock is shown, we will assume that each component of the circuit is connected to the clock, and the component only acts according to the clock. Thus, the adder will compute a new value immediately, but the program counter will not be updated until the clock pulses. Throughout our discussion, we will assume that the clock pulses once per instruction.

Each line in the figure represents a *data path* that consists of multiple parallel wires. In the figure, each data path is thirty-two bits wide. That is, the adder takes two inputs, both of which are thirty-two bits. The value from the instruction pointer is obvious because the instruction pointer has thirty-two bits. The other input, marked with the label *4* represents a thirty-two-bit constant with the numeric value 4. That is, we imagine thirty-two wires that are all zero except the third wire. The adder computes the sum and produces a thirty-two-bit result.

6.6 Fetching An Instruction

The next step in constructing a computer consists of fetching an instruction from memory. For our simplistic example, we will assume that a dedicated *instruction memory* holds the program to be executed, and that a memory hardware unit takes an address as input and extracts a thirty-two bit data value from the specified location in memory. That is, we imagine a memory to be an array of bytes that has a set of input lines and a set of output lines. Whenever a value is placed on the input lines, the memory uses the value as input to a decoder, selects the appropriate bytes, and sets the output lines to the value found in the bytes. Figure 6.5 illustrates how the value in a program counter is used as an address for the instruction memory.

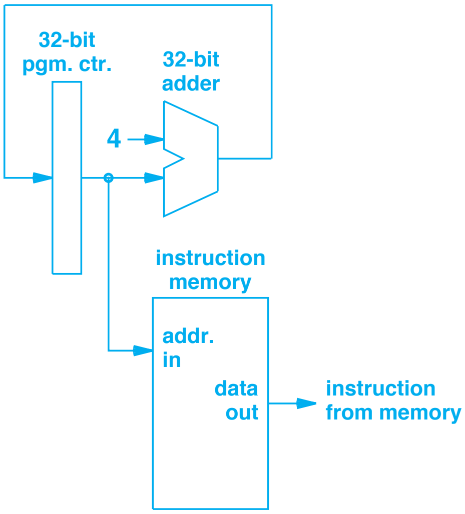


Figure 6.5 The data path used during *instruction fetch* in which the value in a program counter is used as a memory address.

6.7 Decoding An Instruction

When an instruction is fetched from memory, it consists of thirty-two bits. The next conceptual step in execution consists of *instruction decoding*. That is, the hardware separates fields of the instruction such as the operation, registers specified, and offset. Recall from Figure 6.2 how the bits of an instruction are organized. Because we used separate bit fields for each item, instruction decoding is trivial — the hardware simply separates the wires that carry bits for the operation field, each of the three register fields, and the offset field. Figure 6.6 illustrates how the output from the instruction memory is fed to an instruction decoder.

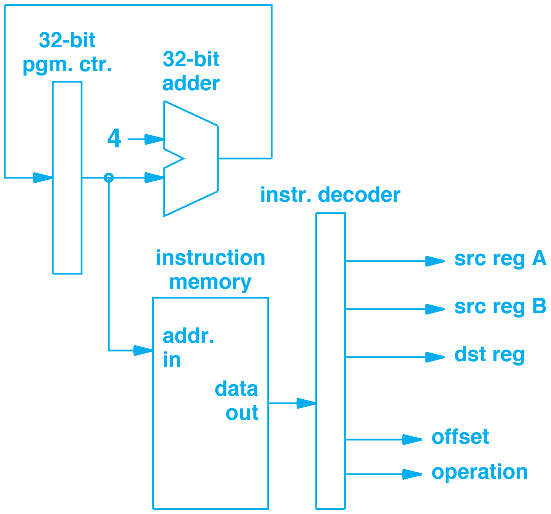


Figure 6.6 Illustration of an instruction decoder connected to the output of the instruction memory.

In the figure, individual outputs from the instruction decoder do not all have thirty-two bits. The operation consists of five bits, the outputs that correspond to registers consist of four bits each, the output labeled *offset* consists of fifteen bits. Thus, we can think of a line in the data path diagram as indicating one or more bits of data.

It is important to understand that the output from the decoder consists of fields from the instruction. For example, the path labeled *offset* contains the fifteen offset bits from the instruction. Similarly, the data path labeled *reg A* merely contains the four bits from the *reg A* field in the instruction. The point is that the data for *reg A* only specifies which register to use, and does not carry the value that is currently in the register. We can summarize:

Our example instruction decoder merely extracts bit fields from an instruction without interpreting the fields.

Unlike our imaginary computer, a real processor may have multiple instruction formats (e.g., the fields in an arithmetic instruction may be in different locations than the fields in a memory access instruction). Furthermore, a real processor may have variable length instructions. As a result, an instruction decoder may need to examine the operation to decide the location of fields. Nevertheless, the principle applies: a decoder extracts fields from an instruction and passes each field along a data path.

6.8 Connections To A Register Unit

The register fields of an instruction are used to select registers that are used in the instruction. In our example, a *jump* instruction uses one register, a *load* or *store* instruction uses two, and an *add* instruction uses three. Therefore, each of the three possible register fields must be connected to a register storage unit as Figure 6.7 illustrates.

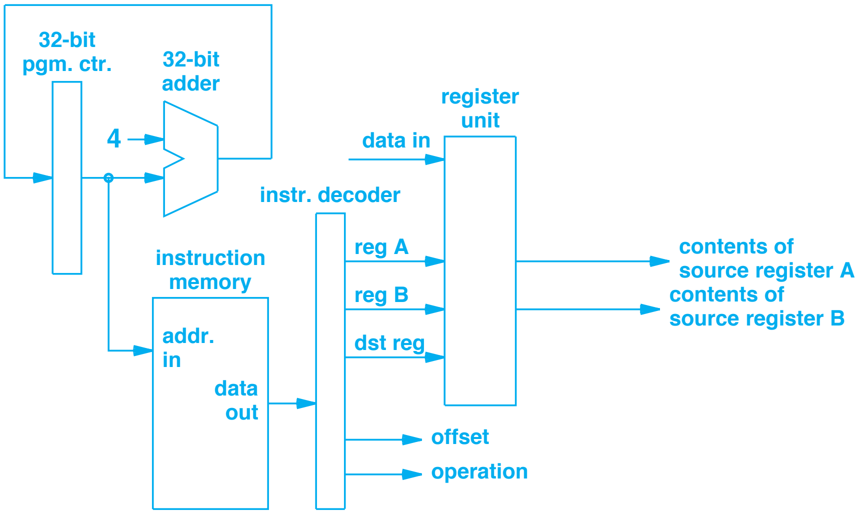


Figure 6.7 Illustration of a register unit attached to an instruction decoder.

6.9 Control And Coordination

Although all three register fields connect to the register unit, the unit does not always use all three. Instead, a register unit contains logic that determines whether a given instruction reads existing values from registers or writes data into one of the registers. In particular, the *load* and *add* instructions each write a result to a register, but the *jump* and *store* instructions do not.

It may seem that the operation portion of the instruction should be passed to the register unit to allow the unit to know how to act. To understand why the figure does not show a connection between remaining fields of the instruction and the register unit, remember that we are only examining data paths (i.e., the hardware paths along which data can flow). In an actual computer, each of the units illustrated in the figure will have additional connections that carry control signals. For example, each unit must receive a clock signal to ensure that it coordinates to take action at the correct time (e.g., to ensure that the data memory does not store a value until the correct address has been computed).

In practice, most computers use an additional hardware unit, known as a *controller*, to coordinate overall data movement and each of the functional units. A controller must have one or more connections to each of the other units, and must use the *operation* field of an instruction to determine how each unit should operate to perform the instruction. In the diagram, for example, a connection between the controller and register unit would be used to specify whether the register unit should fetch the values of one or two registers, and whether the unit should accept data to be placed in a register. For now, we will assume that a controller exists to coordinate the operation of all units.

6.10 Arithmetic Operations And Multiplexing

Our example set of instructions illustrates an important principle: hardware that is designed to re-use functional units. Consider arithmetic. Only the *add* instruction performs arithmetic explicitly. A real processor will have several arithmetic and logical instructions (e.g., *subtract, shift*, *logical and*, etc), and will use the *operation* field in the instruction to decide which the ALU should perform.

Our instruction set also has an implicit arithmetic operation associated with the *load*, *store*, and *jump* instructions. Each of those instructions requires an addition operation to be performed when the instruction is executed. Namely, the processor must add the offset value, which is found in the instruction itself, to the contents of a register. The resulting sum is then treated as a memory address.

The question arises: should a processor have a separate hardware unit to compute the sum needed for an address, or should a single ALU be used for both general arithmetic and address arithmetic? Such questions form the basis for key decisions in processor design. Separate functional units have the advantage of speed and ease of design. Re-using a functional unit for multiple purposes has the advantage of taking less power.

Our design illustrates re-use. Like many processors, our design contains a single *Arithmetic Logic Unit* (*ALU*) that performs all arithmetic operations†. For our sample instruction set, inputs to the ALU can come from two sources: either a pair of registers or a register and the offset field in an instruction. How can a hardware unit choose among multiple sources of input? The mechanism that accommodates two possible inputs is known as a *multiplexor*. The basic idea is that a multiplexor has *K* data inputs, one data output, and a set of control lines used to specify which input is sent to the output. To understand how a multiplexor is used, consider Figure 6.8, which shows a multiplexor between the register unit and ALU. When viewing the figure, remember that each line in our diagram represents a data path with thirty-two bits. Thus, each input to the multiplexor contains thirty-two bits as does the output. The multiplexor selects all thirty-two bits from one of the two inputs and sends them to the output.

†Incrementing the program counter is a special case.

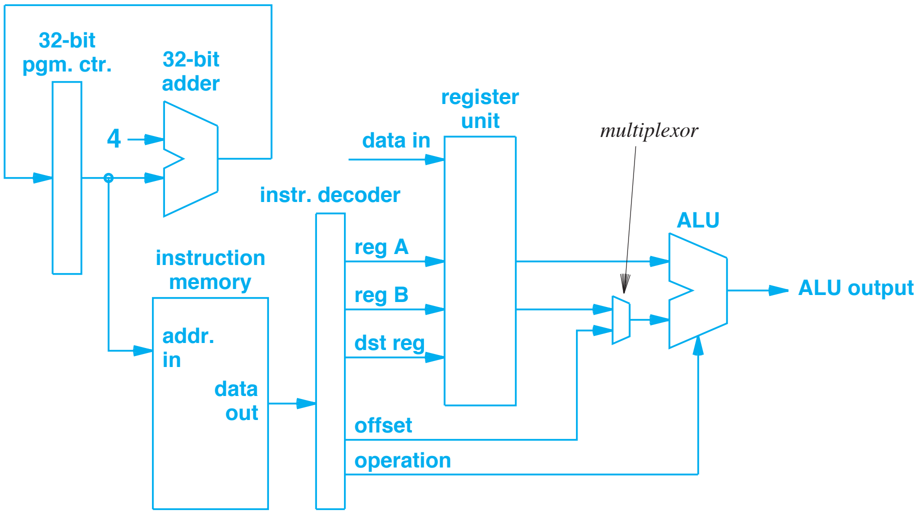


Figure 6.8 Illustration of a multiplexor used to select an input for the ALU.

In the figure, inputs to the multiplexor come from the register unit and the offset field in the instruction. How does the multiplexor decide which input to pass along? Recall that our diagram only shows the data path. In addition, the processor contains a controller, and all units are connected to the controller. When the processor executes an *add* instruction, the controller signals the multiplexor to select the input coming from the register unit. When the processor executes other instructions, the controller specifies that the multiplexor should select the input that comes from the offset field in the instruction.

Observe that the *operation* field of the instruction is passed to the ALU. Doing so permits the ALU to decide which operation to perform. In the case of an arithmetic or logical instruction (e.g., *add*, *subtract*, *right shift*, *logical and*), the ALU uses the operation to select the appropriate action. In the case of other instructions, the ALU performs addition.

6.11 Operations Involving Data In Memory

When it executes a *load* or *store* operation, the computer must reference an item in the data memory. For such operations, the ALU is used to add the offset in the instruction to the contents of a register, and the result is used as a memory address. In our simplified design, the memory used to store data is separate from the memory used to store instructions. Figure 6.9 illustrates the data paths used to connect a data memory.

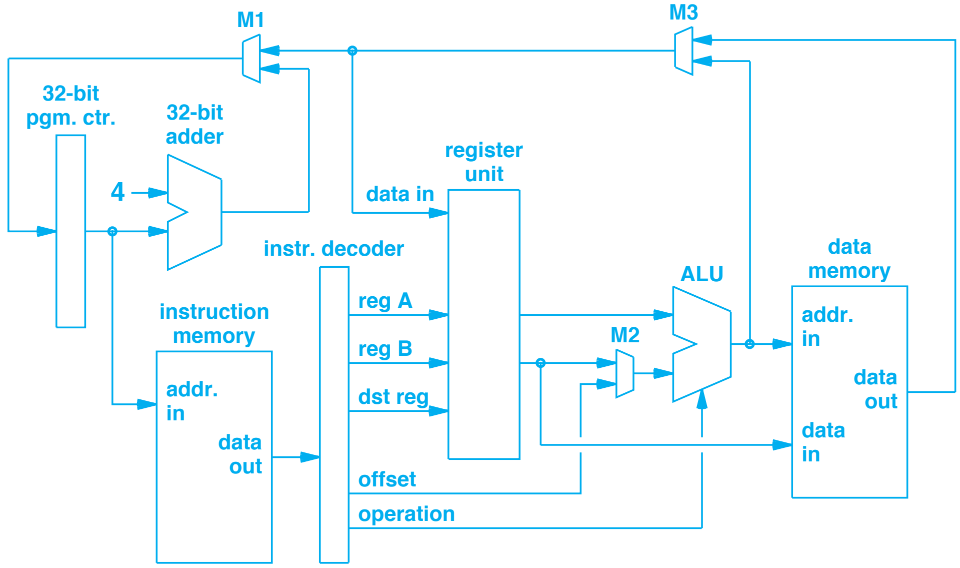


Figure 6.9 Illustration of data paths including data memory.

6.12 Example Execution Sequences

To understand how computation proceeds, consider the data paths that are used for each instruction. The following paragraphs explain the sequence. In each case, the program counter gives the address of an instruction, which is passed to the instruction memory. The instruction memory fetches the value from memory, and passes bits of the value to the instruction decoder. The decoder separates fields of the instruction and passes them to other units. The remainder of the operation depends on the instruction.

Add. For an *add* instruction, the register unit is given three register numbers, which are passed along paths labeled *reg A*, *reg B*, and *dst reg*. The register unit fetches the values in the first two registers, which are passed to the ALU. The register unit also prepares to write to the third register. The ALU uses the operation code to determine that addition is required. To allow the *reg B* output from the register unit to reach the ALU, the controller (not shown) must set multiplexor M2 to pass the value from the *B* register unit and to ignore the *offset* value from the decoder. The controller must set multiplexor M3 to pass the output from the ALU to the register unit's *data input*, and must set multiplexor M1 to ignore the output from the ALU. Once the output from the ALU reaches the input connection on the register unit, the register unit stores the value in the register specified by the path labeled *dst reg*, and the operation is complete.

Store. After a *store* instruction has been fetched from memory and decoded, the register unit fetches the values for registers *A* and *B*, and places them on its output lines. Multiplexor M2 is set to pass the *offset* field to the ALU and ignore the value of register *B*. The controller instructs the ALU to perform addition, which adds the offset and contents of register *A*. The resulting sum is passed to the data memory as an address. Meanwhile, the register *B* value (the second output of the register unit) is passed to the *data in* connection on the data memory. The controller instructs the data memory to perform a *write* operation, which writes the value of register *B* into the location specified by the value on the address lines, and the operation is complete.

Load. After a *load* instruction has been fetched and decoded, the controller sets multiplexor M2 so the ALU receives the contents of register *A* and the *offset* field from the instruction. As with a *store*, the controller instructs the ALU to perform the addition, and the result is passed to the data memory as an address. The controller signals the data memory to perform a *fetch* operation, which means the output of the data memory is the value at the location given by the address input. The controller must set multiplexor M3 to ignore the output from the ALU and pass the output of the data memory along the *data in* path of the register unit. The controller signals the register unit to store its input value in the register specified by register *dst reg*. Once the register unit stores the value, execution of the instruction is complete.

Jump. After a *jump* instruction has been fetched and decoded, the controller sets multiplexor M2 to pass the *offset* field from the instruction, and instructs the ALU to perform the addition. The ALU adds the offset to the contents of register *A*. To use the result as an address, the controller sets multiplexor M3 to pass the output from the ALU and ignore the output from the data memory. Finally, the controller sets multiplexor M1 to pass the value from the ALU to the program counter. Thus, the result from the ALU becomes the input of the 32-bit program counter. The program counter receives and stores the value, and the instruction is complete. Recall that the program counter always specifies the address in memory from which to fetch the next instruction. Therefore, when the next instruction executes, the instruction will be extracted from the address that was computed in the previous instruction (i.e., the program will jump to the new location).

6.13 Summary

A computer system is programmable, which means that instead of having the entire sequence of operations hardwired into digital logic, the computer executes instructions from memory. Programmability provides substantial computational power and flexibility, allowing one to change the functionality of a computer by loading a new program into memory. Although the overall design of a computer that executes instructions is complex, the basic components are not difficult to understand.

A computer consists of multiple hardware components, such as a program counter, memories, register units, and an ALU. Connections among components form the computer's *data path*. We examined a set of components sufficient to execute basic instructions, and reviewed hardware for the steps of instruction fetch, decode, and execute, including register and data access. The encoding used for instructions is selected to make hardware design easier — fields from the instruction are extracted and passed to each of the hardware units.

In addition to the data path, a controller has connections to each of the hardware units. A *multiplexor* is an important mechanism that allows the controller to route data among the hardware units. In essence, each multiplexor acts as a switch that allows data from one of several sources to be sent to a given output. When an instruction executes, a controller uses fields of the instruction to determine how to set the multiplexors during the execution. Multiplexors permit a single ALU to compute address offsets as well as to compute arithmetic operations.

We reviewed execution of basic instructions and saw how multiplexors along the data path in a computer can control which values pass to a given hardware unit. We saw, for example, that a multiplexor selects whether the program counter is incremented by four to move to the next instruction or has the value replaced by the output of the ALU (to perform a *jump*).

EXERCISES

6.1 Does the example system follow the Von Neumann Architecture? Why or why not?

6.2 Consult Figure 6.3, and show each individual bit when the following instruction is stored in memory:

```
add  r1, r14, r9
```

6.3 Consult Figure 6.3, and show each individual bit when the following instruction is stored in memory:

```
load  r7, 43(r15)
```

6.4 Why is the following instruction invalid?

```
jump  40000(r15)
```

Hint: consider storing the instruction in memory.

6.5 The example presented in this chapter uses four instructions. Given the binary representation in Figure 6.2, how many possible instructions (opcodes) can be created?

6.6 Explain why the circuit in Figure 6.5 is not merely an infinite loop that runs wildly.

6.7 When a *jump* instruction is executed, what operation does the ALU perform?

6.8 A data path diagram, such as the diagram in Figure 6.9 hides many details. If the example is changed so that every instruction is sixty-four bits long, what trivial change must be made to the figure?

6.9 Make a table of all instructions and show how each of the multiplexors is set when the instruction is executed.

6.10 Modify the example system to include additional operations *right shift* and *subtract*.

6.11 In Figure 6.9, which input does multiplexor M1 forward during an *add* instruction?

6.12 In Figure 6.9, for what instructions does multiplexor M3 select the input from the ALU?

6.13 Redesign the computer system in Figure 6.9 to include a *relative branch* instruction. Assume the offset field contains a signed value, and add the value to the current program counter to produce the next value for the program counter.

6.14 Can the system in Figure 6.9 handle multiplication? Why or why not?

Chapter Contents

7

Operand Addressing And Instruction Representation

7.1 Introduction

The previous chapters discuss types of processors and consider processor instruction sets. This chapter focuses on two details related to instructions: the ways instructions are represented in memory and the ways that operands can be specified. We will see that the form of operands is especially relevant to programmers. We will also understand how the representation of instructions determines the possible operand forms.

The next chapter continues the discussion of processors by explaining how a *Central Processing Unit* (*CPU*) operates. We will see how a CPU combines many features we have discussed into a large, unified system.

7.2 Zero, One, Two, Or Three Address Designs

We said that an instruction is usually stored as an opcode followed by zero or more operands. How many operands are needed? The discussion in Chapter 5 assumes that the number of operands is determined by the operation being performed. Thus, an *add* instruction needs at least two operands because addition involves at least two quantities. Similarly, a Boolean *not* instruction needs one operand because logical inversion only involves one quantity. However, the example MIPS instruction set in Chapter 5 employs an additional operand on each instruction that specifies the location for the result.

Thus, in the example instruction set, an *add* instruction requires three operands: two that specify values to be added and a third that specifies a location for the result.

Despite the intuitive appeal of a processor in which each instruction can have an arbitrary number of operands, many processors do not permit such a scheme. To understand why, we must consider the underlying hardware. First, because an arbitrary number of operands implies variable-length instructions, fetching and decoding instructions is less efficient than using fixed-length instructions. Second, because fetching an arbitrary number of operands takes time, the processor will run slower than a processor with a fixed number of operands.

It may seem that parallel hardware can solve some of the inefficiency. Imagine, for example, parallel hardware units that each fetch one operand of an instruction. If an instruction has two operands, two units operate simultaneously; if an instruction has four operands, four units operate simultaneously. However, parallel hardware uses more space on a chip and requires additional power. In addition, the number of pins on a chip limits the amount of data from outside the chip that can be accessed in parallel. Thus, parallel hardware is not an attractive option in many cases (e.g., a processor in a portable phone that operates on battery power).

Can an instruction set be designed without allowing arbitrary operands? If so, what is the smallest number of operands that can be useful for general computation? Early computers answered the question by using a scheme in which each instruction only has one operand. Later computers introduced instruction sets that limited each instruction to two operands. Surprisingly, computers also exist in which instructions have no operands in the instruction itself. Finally, as we have seen in the previous chapter, some processors limit instructions to three operands.

7.3 Zero Operands Per Instruction

An architecture in which instructions have no operands is known as a *0-address* architecture. How can an architecture allow instructions that do not specify any operands? The answer is that operands must be *implicit*. That is, the location of the operands is already known. A 0-address architecture is also called a *stack architecture* because operands are kept on a run-time stack. For example, an *add* instruction takes two values from the top of the stack, adds them together, and places the result back on the stack. Of course, there are a few exceptions, and some of the instructions in a stack computer allow a programmer to specify an operand. For example, most zero-address architectures include a *push* instruction that inserts a new value on the top of the stack, and a *pop* instruction removes the top value from the stack and places the value in memory. Thus, on a stack machine, to add seven to variable X, one might use a sequence of instructions similar to the example in Figure 7.1.

The chief disadvantage of a stack architecture arises from the use of memory — it takes much longer to fetch operands from memory than from registers in the processor. A later section discusses the concept; for now, it is sufficient to understand why the computer industry has moved away from stack architectures.

```
push X
push 7
add
pop  X
```

Figure 7.1 An example of instructions used on a stack computer to add seven to a variable X. The architecture is known as a zero-address architecture because the operands for an instruction such as *add* are found on the stack.

7.4 One Operand Per Instruction

An architecture that limits each instruction to a single operand is classified as a *1-address* design. In essence, a 1-address design relies on an *implicit* operand for each instruction: a special register known as an *accumulator*†. One operand is in the instruction and the processor uses the value of the accumulator as a second operand. Once the operation has been performed, the processor places the result back in the accumulator. We think of an instruction as operating on the value in the accumulator. For example, consider arithmetic operations. Suppose an addition instruction has operand *X*:

```
add   X
```

When it encounters the instruction, the processor performs the following operation:

$$\text{accumulator} \leftarrow \text{accumulator} + \text{X}$$

Of course, the instruction set for a 1-address processor includes instructions that allow a programmer to load a constant or the value from a memory location into the accumulator or store the current value of the accumulator into a memory location.

7.5 Two Operands Per Instruction

Although it works well for arithmetic or logical operations, a 1-address design does not allow instructions to specify two values. For example, consider copying a value from one memory location to another. A 1-address design requires two instructions that load the value into the accumulator and then store the value in the new location. The design is especially inefficient for a system that moves graphics objects in display memory.

To overcome the limitations of 1-address systems, designers invented processors that allow each instruction to have two addresses. The approach is known as a *2-address* architecture. With a 2-address processor, an operation can be applied to a specified value instead of merely to the accumulator. Thus, in a 2-address processor,

†The general-purpose registers discussed in Chapter 5 can be considered an extension of the original accumulator concept.

```
add   X   Y
```

specifies that the value of X is to be added to the current value of Y:

$$Y \leftarrow Y + X$$

Because it allows an instruction to specify two operands, a 2-address processor can offer data movement instructions that treat the operands as a *source* and *destination*. For example, a 2-address instruction can copy data directly from location Q to location R†:

```
move   Q   R
```

7.6 Three Operands Per Instruction

Although a 2-address design handles data movement, further optimization is possible, especially for processors that have multiple general-purpose registers: allow each instruction to specify three operands. Unlike a 2-address design, the key motivation for a *3-address* architecture does not arise from operations that require three input values. Instead, the point is that the third operand can specify a destination. For example, an addition operation can specify two values to be added as well as a destination for the result:

```
add   X   Y   Z
```

specifies an assignment of:

$$Z \leftarrow X + Y$$

7.7 Operand Sources And Immediate Values

The discussion above focuses on the number of operands that each instruction can have without specifying the exact details of an operand. We know that an instruction has a bit field for each operand, but questions arise about how the bits are interpreted. How is each type of operand represented in an instruction? Do all operands use the same representation? What semantic meaning is given to a representation?

To understand the issue, observe that the data value used as an operand can be obtained in many ways. Figure 7.2 lists some of the possibilities for operands in a 3-address processor‡.

†Some architects reserve the term *2-address* for instructions in which both operands specify a memory location, and use the term *1 $^1/_2$-address* for situations where one operand is in memory and the other operand is in a register.

‡To increase performance, modern 3-address architectures often limit operands so that at most one of the operands in a given instruction refers to a location in memory; the other two operands must specify registers.

Operand used as a source (item used in the operation)

- **A signed constant in the instruction**
- **An unsigned constant in the instruction**
- **The contents of a general-purpose register**
- **The contents of a memory location**

Operand used as a destination (location to hold the result)

- **A general-purpose register**
- **A contiguous pair of general-purpose registers**
- **A memory location**

Figure 7.2 Examples of items an operand can reference in a 3-address processor. A source operand specifies a value and a destination operand specifies a location.

As the figure indicates, most architectures allow an operand to be a constant. Although the operand field is small, having an explicit constant is important because programs use small constants frequently (e.g., to increment a loop index by 1); encoding a constant in the instruction is faster and requires fewer registers.

We use the term *immediate value* to refer to an operand that is a constant. Some architectures interpret immediate values as signed, some interpret them as unsigned, and others allow a programmer to specify whether the value is signed or unsigned.

7.8 The Von Neumann Bottleneck

Recall that conventional computers that store both programs and data in memory are classified as following a *Von Neumann Architecture*. Operand addressing exposes the central weakness of a Von Neumann Architecture: memory access can become a bottleneck. That is, because instructions are stored in memory, a processor must make at least one memory reference per instruction. If one or more operands specify items in memory, the processor must make additional memory references to fetch or store values. To optimize performance and avoid the bottleneck, operands must be taken from registers instead of memory.

The point is:

> *On a computer that follows the Von Neumann Architecture, the time spent accessing memory can limit the overall performance. Architects use the term* Von Neumann bottleneck *to characterize the situation, and avoid the bottleneck by choosing designs in which operands are found in registers.*

7.9 Explicit And Implicit Operand Encoding

How should an operand be represented in an instruction? The instruction contains a bit field for each operand, but an architect must specify exactly what the bits mean (e.g., whether they contain an immediate value, the number of a register, or a memory address). Computer architects have used two interpretations of operands: *implicit* and *explicit*. The next sections describe each of the approaches.

7.9.1 Implicit Operand Encoding

An *implicit operand encoding* is easiest to understand: the opcode specifies the types of operands. That is, a processor that uses implicit encoding contains multiple opcodes for a given operation — each opcode corresponds to one possible combination of operands. For example, Figure 7.3 lists three instructions for addition that might be offered by a processor that uses implicit operand encoding.

Opcode	Operands	Meaning
Add register	R1 R2	R1 ← R1 + R2
Add immediate signed	R1 I	R1 ← R1 + I
Add immediate unsigned	R1 UI	R1 ← R1 + UI
Add memory	R1 M	R1 ← R1 + memory[M]

Figure 7.3 An example of addition instructions for a 2-address processor that uses implicit operand encoding. A separate opcode is used for each possible combination of operands.

As the figure illustrates, not all operands need to have the same interpretation. For example, consider the *add immediate signed* instruction. The instruction takes two operands: the first operand is interpreted to be a register number, and the second is interpreted to be a signed integer.

7.9.2 Explicit Operand Encoding

The chief disadvantage of implicit encoding is apparent from Figure 7.3: multiple opcodes are needed for a given operation. In fact, a separate opcode is needed for each combination of operands. If the processor uses many types of operands, the set of opcodes can be extremely large. As an alternative, an *explicit operand encoding* associates type information with each operand. Figure 7.4 illustrates the format of two *add* instructions for an architecture that uses explicit operand encoding.

As the figure shows, the operand field is divided into two subfields: one specifies the type of the operand and the other specifies a value. For example, an operand that references a register begins with a type field that specifies the remaining bits are to be interpreted as a register number.

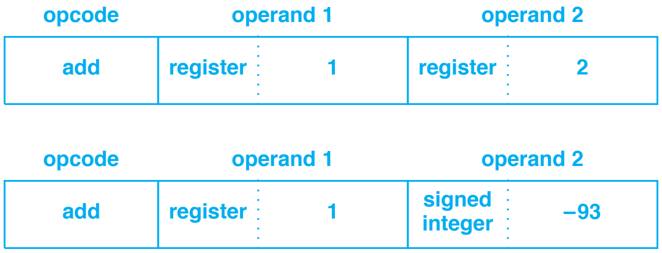


Figure 7.4 Examples of operands on an architecture that uses explicit encoding. Each operand specifies a type as well as a value.

7.10 Operands That Combine Multiple Values

The discussion above implies that each operand consists of a single value extracted from a register, memory, or the instruction itself. Some processors do indeed restrict each operand to a single value. However, other processors provide hardware that can compute an operand value by extracting and combining values from multiple sources. Typically, the hardware computes a sum of several values.

An example will help clarify how hardware handles operands composed of multiple values. One approach is known as a *register-offset* mechanism. The idea is straightforward: instead of two subfields that specify a type and value, each operand consists of three fields that specify a *register-offset type*, a *register*, and an *offset*. When it fetches an operand, the processor adds the contents of the offset field to the contents of the specified register to obtain a value that is then used as the operand. Figure 7.5 shows an example *add* instruction with register-offset operands.

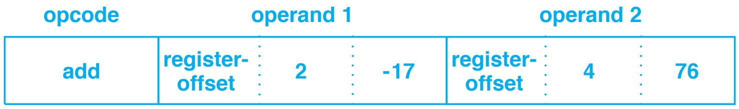


Figure 7.5 An example of an *add* instruction in which each operand consists of a register plus an offset. During operand fetch, the hardware adds the offset to the specified register to obtain the value of the operand.

In the figure, the first operand specifies the contents of register 2 minus the constant 17, and the second operand specifies the contents of register 4 plus the constant 76. When we discuss memory, we will see that allowing an operand to specify a register plus an offset is especially useful when referencing a data aggregate such as a C language *struct* because a pointer to the structure can be left in a register and offsets used to reference individual items.

7.11 Tradeoffs In The Choice Of Operands

The discussion above is unsatisfying — it seems that we have listed many design possibilities but have not focused on which approach has been adopted. In fact, there is no best choice, and each operand style we discussed has been used in practice. Why hasn't one particular style emerged as optimal? The answer is simple: each style represents a tradeoff between ease of programming, size of the code, speed of processing, and complexity of the hardware. The next paragraphs discuss several potential design goals, and explain how each relates to the choice of operands.

Ease Of Programming. Complex forms of operands make programming easier. For example, we said that allowing an operand to specify a register plus an offset makes data aggregate references straightforward. Similarly, a 3-address approach that provides an explicit target means a programmer does not need to code separate instructions to copy results into their final destination. Of course, to optimize ease of programming, an architect needs to trade off other aspects.

Fewer Instructions. Increasing the expressive power of operands reduces the number of instructions in a program. For example, allowing an operand to specify both a register and an offset means that the program does not need to use an extra instruction to add an offset to a register. Increasing the number of addresses per instruction also lowers the count of instructions (e.g., a 3-address processor requires fewer instructions than a 2-address processor). Unfortunately, fewer instructions produce a tradeoff in which each instruction is larger.

Smaller Instruction Size. Limiting the number of operands, the set of operands types, or the maximum size of an operand keeps each instruction small because fewer bits are needed to identify the operand type or represent an operand value. In particular, an operand that specifies only a register will be smaller than an operand that specifies a register and an offset. As a result, some of the smallest, least powerful processors limit operands to registers — except for *load* and *store* operations, each value used in a program must come from a register. Unfortunately, making each instruction smaller decreases the expressive power, and therefore increases the number of instructions needed.

Larger Range Of Immediate Values. Recall from Chapter 3 that a string of k bits can hold 2^k possible values. Thus, the number of bits allocated to an operand determines the numeric range of immediate values that can be specified. Increasing the range of immediate values results in larger instructions.

Faster Operand Fetch And Decode. Limiting the number of operands and the possible types of each operand allows hardware to operate faster. To maximize speed, for example, an architect avoids register-offset designs because hardware can fetch an operand from a register much faster than it can compute the value from a register plus an offset.

Decreased Hardware Size And Complexity. The amount of space on an integrated circuit is limited, and an architect must decide how to use the space. Decoding complex forms of operands requires more hardware than decoding simpler forms. Thus, limiting the types and complexity of operands reduces the size of the circuitry required. Of course, the choice represents a tradeoff: programs are larger.

The point is:

> *Processor architects have created a variety of operand styles. No single form is optimal for all processors because the choice represents a compromise among functionality, program size, complexity of the hardware required to fetch values, performance, and ease of programming.*

7.12 Values In Memory And Indirect Reference

A processor must provide a way to access values in memory. That is, at least one instruction must have an operand which the hardware interprets as a memory address†. Accessing a value in memory is significantly more expensive than accessing a value in a register. Although it may make programming easier, a design in which each instruction references memory usually results in lower performance. Thus, programmers usually structure code to keep values that will be used often in registers and only reference memory when needed.

Some processors extend memory references by permitting various forms of *indirection*. For example, an operand that specifies *indirection through register 6* causes a processor to perform two steps:

- Obtain A, the current value from register 6.
- Interpret A as a memory address, and fetch the operand from memory.

One extreme form of operand involves *double indirection*, or indirection through a memory location. That is, the processor interprets the operand as memory address M. However, instead of loading or storing a value to address M, the processor assumes M contains the memory address of the value. In such cases, a processor performs the following steps:

- Obtain M, the value in the operand itself.
- Interpret M as a memory address, and fetch the value A from memory.
- Interpret A as another memory address, and fetch the operand from memory.

Double indirection that goes through one memory location to another can be useful when a program has to follow a linked list in memory. However, the overhead is extremely high (execution of a single instruction entails multiple memory references).

† The third section of the text describes memory and memory addressing.

7.13 Illustration Of Operand Addressing Modes

A processor usually contains a special internal register, called an *instruction register*, that is used to hold an instruction while the instruction is being decoded. The possible types of operand addresses and the cost of each can be envisioned by considering the location of the operand and the references needed to fetch the value. An immediate value is the least expensive because the value is located in the instruction register (i.e., in the instruction itself). A general-purpose register reference is slightly more expensive than an immediate value. A reference to memory is more expensive than a reference to a register. Finally, double indirection, which requires two memory references, is the most expensive. Figure 7.6 lists the possibilities, and illustrates the hardware units involved in resolving each.

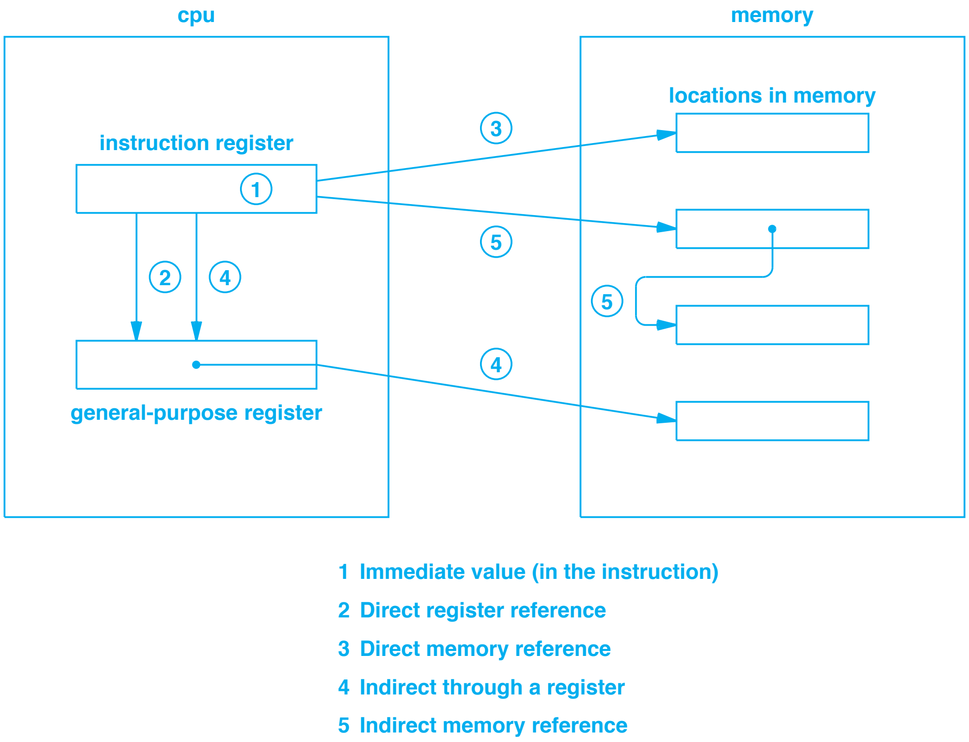

cpu
memory
locations in memory
instruction register
1
2
3
4
5
5
4
general-purpose register
1 Immediate value (in the instruction)
2 Direct register reference
3 Direct memory reference
4 Indirect through a register
5 Indirect memory reference

Figure 7.6 Illustration of the hardware units accessed when fetching an operand in various addressing modes. Indirect references take longer than direct references.

In the figure, modes 3 and 5 each require the instruction to contain a memory address. Although they were available on earlier computers, such modes have become unpopular because they require an instruction to be quite large.

7.14 Summary

When designing a processor, an architect chooses the number and possible types of operands for each instruction. To make operand handling efficient, many processors limit the number of operands for a given instruction to three or fewer.

An immediate operand specifies a constant value; other possibilities include an operand that specifies using the contents of a register or a value in memory. Indirection allows a register to contain the memory address of the operand. Double indirection means the operand specifies a memory address and the value at the address is a pointer to another memory location that holds the value. The type of the operand can be encoded implicitly (i.e., in the opcode) or explicitly.

Many variations exist because the choice of operand number and type represents a tradeoff among functionality, ease of programming, and engineering details such as the speed of processing.

EXERCISES

7.1 Suppose a computer architect is designing a processor for a computer that has an extremely slow memory. Would the architect choose a zero-address architecture? Why or why not?

7.2 Consider the size of instructions in memory. If an architecture allows immediate operands to have large numeric values, an instruction takes more space in memory. Why?

7.3 Assume a stack machine keeps the stack in memory. Also assume variable *p* is stored in memory. How many memory references will be needed to increment *p* by seven?

7.4 Assume two integers, *x* and *y* are stored in memory, and consider an instruction that sets *z* to the sum of *x*+*y*. How many memory references will be needed on a two-address architecture? Hint: remember to include *instruction fetch*.

7.5 How many memory operations are required to perform an *add* operation on a 3-address architecture if each operand specifies an indirect memory reference?

7.6 If a programmer increments a variable by a value that is greater than the maximum immediate operand, an optimizing compiler may generate two instructions. For example, on a computer that only allows immediate values of 127 or less, incrementing variable *x* by 140 results in the sequence:

```
load  r7, x
add_immediate  r7, 127
add_immediate  t7, 13
store  r7, x
```

Why doesn't the compiler store 140 in memory and add the value to register 7?

7.7 Assume a memory reference takes twelve times as long as a register reference, and assume a program executes N instructions on a 2-address architecture. Compare the running time of the program if all operands are in registers to the running time if all operands are in memory. Hint: *instruction fetch* requires a memory operation.

7.8 Consider each type of operand that Figure 7.6 illustrates, and make a table that contains an expression for the number of bits required to represent the operand. Hint: the number of bits required to represent values from zero through N is:

$$\left\lfloor \log_2 N \right\rfloor$$

7.9 Name one advantage of using a higher number of addresses per instruction.

7.10 Consider a two-address computer that uses implicit operands. Suppose one of the two operands can be any of the five operand types in Figure 7.6, and the other can be any except an immediate value. List all the *add* instructions the computer needs.

7.11 Most compilers contain optimization modules that choose to keep frequently used variables in registers rather than writing them back to memory. What term characterizes the problem that such an optimization module is attempting to overcome?

Chapter Contents

8

CPUs: Microcode, Protection, And Processor Modes

8.1 Introduction

Previous chapters consider two key aspects of processors: instruction sets and operands. The chapters explain possible approaches, and discuss the advantages and disadvantages of each approach. This chapter considers a broad class of general-purpose processors, and shows how many of the concepts from previous chapters are applied. The next chapter considers low-level programming languages used with processors.

8.2 A Central Processor

Early in the history of computers, centralization emerged as an important architectural approach — as much functionality as possible was collected into a single processor. The processor, which became known as a *Central Processing Unit* (*CPU*), controlled the entire computer, including both calculations and I/O.

In contrast to early designs, a modern computer system follows a decentralized approach. The system contains multiple processors, many of which are dedicated to a specific function or a hardware subsystem. For example, we will see that an I/O device, such as a disk, can include a processor that handles disk transfers.

Despite the shift in paradigm, the term CPU has survived because one chip contains the hardware used to perform most computations and coordinate and control other processors. In essence, the CPU manages the entire computer system by telling other processors when to start, when to stop, and what to do. When we discuss I/O, we will see how the CPU controls the operation of peripheral devices and processors.

8.3 CPU Complexity

Because it must handle a wide variety of control and processing tasks, a modern CPU is extremely complex. For example, Intel makes a CPU chip that contains 2.5 billion transistors. Why is a CPU so complex? Why are so many transistors needed?

Multiple Cores. In fact, modern CPU chips do not contain just one processor. Instead, they contain multiple processors called *cores*. The cores all function in parallel, permitting multiple computations to proceed at the same time. Multicore designs are required for high performance because a single core cannot be clocked at arbitrarily high speeds.

Multiple Roles. One aspect of CPU complexity arises because a CPU must fill several major roles: running application programs, running an operating system, handling external I/O devices, starting or stopping the computer, and managing memory. No single instruction set is optimal for all roles, so a CPU often includes many instructions.

Protection And Privilege. Most computer systems incorporate a system of protection that gives some subsystems higher privilege than others. For example, the hardware prevents an application program from directly interacting with I/O devices, and the operating system code is protected from inadvertent or deliberate change.

Hardware Priorities. A CPU uses a priority scheme in which some actions are assigned higher priority than others. For example, we will see that I/O devices operate at higher priority than application programs — if the CPU is running an application program when an I/O device needs service, the CPU must stop running the application and handle the device.

Generality. A CPU is designed to support a wide variety of applications. Consequently, the CPU instruction set often contains instructions that are used for each type of application (i.e., a CISC design).

Data Size. To speed processing, a CPU is designed to handle large data values. Recall from Chapter 2 that digital logic gates each operate on a single bit of data and that gates must be replicated to handle integers. Thus, to operate on values composed of sixty-four bits, each digital circuit in the CPU must have sixty-four copies of each gate.

High Speed. The final, and perhaps most significant, source of CPU complexity arises from the desire for speed. Recall the important concept discussed earlier:

> *Parallelism is a fundamental technique used to create high-speed hardware.*

That is, to achieve highest performance, the functional units in a CPU must be replicated, and the design must permit the replicated units to operate simultaneously. The large amount of parallel hardware needed to make a modern CPU operate at the highest rate also means that the CPU requires many transistors. We will see further explanations later in the chapter.

8.4 Modes Of Execution

The features listed above can be combined or implemented separately. For example, a given core can be granted access to other parts of memory with or without higher priority. How can a CPU accommodate all the features in a way that allows programmers to understand and use them without becoming confused?

In most CPUs, the hardware uses a set of parameters to handle the complexity and control operation. We say that the hardware has multiple *modes of execution*. At any given time, the current execution mode determines how the CPU operates. Figure 8.1 lists items usually associated with a CPU mode of execution.

- The subset of instructions that are valid
- The size of data items
- The region of memory that can be accessed
- The functional units that are available
- The amount of privilege

Figure 8.1 Items typically controlled by a CPU mode of execution. The characteristics of a CPU can change dramatically when the mode changes.

8.5 Backward Compatibility

How much variation can execution modes introduce? In principle, the modes available on a CPU do not need to share much in common. As one extreme case, some CPUs have a mode that provides *backward compatibility* with a previous model. Backward compatibility allows a vendor to sell a CPU with new features, but also permits customers to use the CPU to run old software.

Intel's line of processors (i.e., 8086, 186, 286,...) exemplifies how backward compatibility can be used. When Intel first introduced a CPU that operated on thirty-two-bit integers, the CPU included a *compatibility mode* that implemented the sixteen-bit instruction set from Intel's previous CPU. In addition to using different sizes of integers, the two architectures have different numbers of registers and different instructions. The two architectures differ so significantly that it is easiest to think of the design as two separate pieces of hardware with the execution mode determining which of the two is used at any time.

We can summarize:

> *A CPU uses an* execution mode *to determine the current operational characteristics. In some CPUs, the characteristics of modes differ so widely that we think of the CPU as having separate hardware subsystems and the mode as determining which piece of hardware is used at the current time.*

8.6 Changing Modes

How does a CPU change execution modes? There are two ways:

- Automatic (initiated by hardware)
- Manual (under program control)

Automatic Mode Change. External hardware can change the mode of a CPU. For example, when an I/O device requests service, the hardware informs the CPU. Hardware in the CPU changes mode (and jumps to the operating system code) automatically before servicing the device. We will learn more when we consider how I/O works.

Manual Mode Change. In essence, manual changes occur under control of a running program. Most often, the program is the operating system, which changes mode before it executes an application. However, some CPUs also provide multiple modes that applications can use, and allow an application to switch among the modes.

What mechanism is used to change mode? Three approaches have been used. In the simplest case, the CPU includes an instruction to set the current mode. In other cases, the CPU contains a special-purpose *mode register* to control the mode. To change modes, a program stores a value into the mode register. Note that a mode register is not a storage unit in the normal sense. Instead, it consists of a hardware circuit that responds to the *store* command by changing the operating mode. Finally, a mode change can occur as the side effect of another instruction. In most CPUs, for example, the instruction set includes an instruction that an application uses to make an operating system call. A mode change occurs automatically whenever the instruction is executed.

To accommodate major changes in mode, additional facilities may be needed to prepare for the new mode. For example, consider a case in which two modes of execution do not share general-purpose registers (e.g., in one mode the registers have sixteen bits and in another mode the registers contain thirty-two bits). It may be necessary to place values in alternate registers before changing mode and using the registers. In such cases, a CPU provides special instructions that allow software to create or modify values before changing the mode.

8.7 Privilege And Protection

The mode of execution is linked to CPU facilities for privilege and protection. That is, part of the current mode specifies the level of privilege for the CPU. For example, when it services an I/O device, a CPU must allow device driver software in the operating system to interact with the device and perform control functions. However, an arbitrary application program must be prevented from accidentally or maliciously issuing commands to the hardware or performing control functions. Thus, before it executes an application program, an operating system changes the mode to reduce privilege. When running in a less privileged mode, the CPU does not permit direct control of I/O devices (i.e., the CPU treats a privileged operation like an invalid instruction).

8.8 Multiple Levels Of Protection

How many levels of privilege are needed, and what operations should be allowed at each level? The subject has been discussed by hardware architects and operating system designers for many years. CPUs have been invented that offer no protection, and CPUs have been invented that offer eight levels, each with more privilege than the previous level. The idea of protection is to help prevent problems by using the minimum amount of privilege necessary at any time. We can summarize:

> *By using a protection scheme to limit the operations that are allowed, a CPU can detect attempts to perform unauthorized operations.*

Figure 8.2 illustrates the concept of two privilege levels.

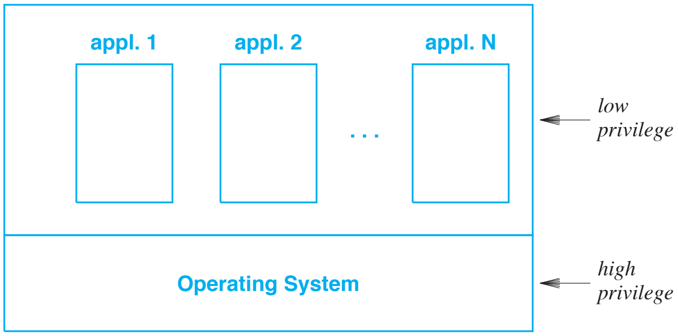


Figure 8.2 Illustration of a CPU that offers two levels of protection. The operating system executes with highest privilege, and application programs execute with less privilege.

Although no protection scheme suffices for all CPUs, designers generally agree on a minimum of two levels for a CPU that runs application programs:

> *A CPU that runs applications needs at least two levels of protection: the operating system must run with absolute privilege, but application programs can run with limited privilege.*

When we discuss memory, we will see that the issues of protection and memory access are intertwined. More important, we will see how memory access mechanisms, which are part of the CPU mode, provide additional forms of protection.

8.9 Microcoded Instructions

How should a complex CPU be implemented? Interestingly, one of the key abstractions used to build a complex instruction set comes from software: complex instructions are programmed! That is, instead of implementing the instruction set directly with digital circuits, a CPU is built in two pieces. First, a hardware architect builds a fast, but small processor known as a *microcontroller*†. Second, to implement the CPU instruction set (called a *macro instruction set*), the architect writes software for the microcontroller. The software that runs on the microcontroller is known as *microcode*. Figure 8.3 illustrates the two-level organization, and shows how each level is implemented.

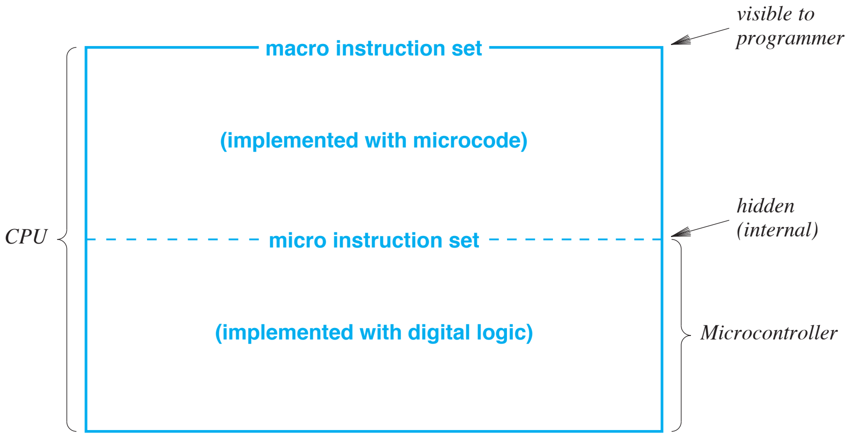


Figure 8.3 Illustration of a CPU implemented with a microcontroller. The macro instruction set that the CPU provides is implemented with microcode.

†The small processor is also called a *microprocessor*, but the term is somewhat misleading.

The easiest way to think about microcode is to imagine a set of functions that each implement one of the CPU macro instructions. The CPU invokes the microcode during the instruction execution. That is, once it has obtained and decoded a macro instruction, the CPU invokes the microcode procedure that corresponds to the instruction.

The macro- and micro architectures can differ. As an example, suppose that the CPU is designed to operate on data items that are thirty-two bits and that the macro instruction set includes an *add32* instruction for integer addition. Further suppose that the microcontroller only offers sixteen-bit arithmetic. To implement a thirty-two-bit addition, the microcode must add sixteen bits at a time, and must add the carry from the low-order bits into the high-order bits. Figure 8.4 lists the microcode steps that are required:

```
/* The steps below assume that two 32-bit operands are
   located in registers labeled R5 and R6, and that the
   microcode must use 16-bit registers labeled r0 through
   r3 to compute the results.
*/
add32:
    move low-order 16 bits from R5 into r2
    move low-order 16 bits from R6 into r3
    add r2 and r3, placing result in r1
    save value of the carry indicator
    move high-order 16 bits from R5 into r2
    move high-order 16 bits from R6 into r3
    add r2 and r3, placing result in r0
    copy the value in r0 to r2
    add r2 and the carry bit, placing the result in r0
    check for overflow and set the condition code
    move the thirty-two bit result from r0 and r1 to
        the desired destination
```

Figure 8.4 An example of the steps required to implement a thirty-two-bit macro addition with a microcontroller that only has sixteen-bit arithmetic. The macro- and micro architectures can differ.

The exact details are unimportant; the figure illustrates how the architecture of the microcontroller and the macro instruction set can differ dramatically. Also note that because each macro instruction is implemented by a microcode program, a macro instruction can perform arbitrary processing. For example, it is possible for a single macro instruction to implement a trigonometric function, such as *sine* or *cosine*, or to move large blocks of data in memory. Of course, to achieve higher performance, an architect can choose to limit the amount of microcode that corresponds to a given instruction.

8.10 Microcode Variations

Computer designers have invented many variations to the basic form of microcode. For example, we said that the CPU hardware implements the fetch-execute cycle and invokes a microcode procedure for each instruction. On some CPUs, microcode implements the entire fetch-execute cycle — the microcode interprets the opcode, fetches operands, and performs the specified operation. The advantage is greater flexibility: microcode defines all aspects of the macro system, including the format of macro instructions and the form and encoding of each operand. The chief disadvantage is lower performance: the CPU cannot have an instruction pipeline implemented in hardware.

As another variation, a CPU can be designed that only uses microcode for extensions. That is, the CPU has a complete macro instruction set implemented directly with digital circuits. In addition, the CPU has a small set of additional opcodes that are implemented with microcode. Thus, a vendor can manufacture minor variations of the basic CPU (e.g., a version with a special encryption instruction intended for customers who implement security software or a version with a special pattern matching instruction intended for customers who implement text processing software). If some or all of the extra instructions are not used in a particular version of the CPU, the vendor can insert microcode that makes them undefined (i.e., the microcode raises an error if an undefined instruction is executed).

8.11 The Advantage Of Microcode

Why is microcode used? There are three motivations. First, because microcode offers a higher level of abstraction, building microcode is less prone to errors than building hardware circuits. Second, building microcode takes less time than building circuits. Third, because changing microcode is easier than changing hardware circuits, new versions of a CPU can be created faster.

We can summarize:

> *A design that uses microcode is less prone to errors and can be updated faster than a design that does not use microcode.*

Of course, microcode does have some disadvantages that balance the advantages:

- Microcode has more overhead than a hardware implementation.
- Because it executes multiple micro instructions for each macro instruction, the microcontroller must run at much higher speed than the CPU.
- The cost of a macro instruction depends on the micro instruction set.

8.12 FPGAs And Changes To The Instruction Set

Because a microcontroller is an internal mechanism intended to help designers, the micro instruction set is usually hidden in the final design. The microcontroller and microcode typically reside on the integrated circuit along with the rest of the CPU, and are only used internally. Only the macro instruction set is available to programmers. Interestingly, some CPUs have been designed that make the microcode dynamic and accessible to customers who purchase the CPU. That is, the CPU contains facilities that allow the underlying hardware to be changed after the chip has been manufactured.

Why would customers want to change a CPU? The motivations are flexibility and performance: allowing a customer to make some changes to CPU instructions defers the decision about a macro instruction set, and allows a CPU's owner to tailor instructions to a specific use. For example, a company that sells video games might add macro instructions to manipulate graphics images, and a company that makes networking equipment might create macro instructions to process packet headers. Using the underlying hardware directly (e.g., with microcode) can result in higher performance.

One technology that allows modification has become especially popular. Known as *Field Programmable Gate Array* (*FPGA*), the technology permits gates to be altered after a chip has been manufactured. Reconfiguring an FPGA is a time-consuming process. Thus, the general idea is to reconfigure the FPGA once, and then use the resulting chip. An FPGA can be used to hold an entire CPU, or an FPGA can be used as a supplement that holds a few extra instructions.

We can summarize:

> *Technologies like dynamic microcode and FPGAs allow a CPU instruction set to be modified or extended after the CPU has been purchased. The motivations are flexibility and higher performance.*

8.13 Vertical Microcode

The question arises: what architecture should be used for a microcontroller? From the point of view of someone who writes microcode, the question becomes: what instructions should the microcontroller provide? We discussed the notion of microcode as if a microcontroller consists of a conventional processor (i.e., a processor that follows a conventional architecture). We will see shortly that other designs are possible.

In fact, a microcontroller cannot be exactly the same as a standard processor. Because it must interact with hardware units in the CPU, a microcontroller needs a few special hardware facilities. For example, a microcontroller must be able to access the ALU and store results in the general-purpose registers that the macro instruction set uses. Similarly, a microcontroller must be able to decode operand references and fetch values. Finally, the microcontroller must coordinate with the rest of the hardware, including memory.

Despite the requirements for special features, microcontrollers have been created that follow the same general approach used for conventional processors. That is, the microcontroller's instruction set contains conventional instructions such as *load*, *store*, *add*, *subtract*, *branch*, and so on. For example, the microcontroller used in a CISC processor can consist of a small, fast RISC processor. We say that such a microcontroller has a *vertical* architecture, and use the term *vertical microcode* to characterize the software that runs on the microcontroller.

Programmers are comfortable with vertical microcode because the programming interface is familiar. Most important, the semantics of vertical microcode are exactly what a programmer expects: one micro instruction is executed at a time. The next section discusses an alternative to vertical microcode.

8.14 Horizontal Microcode

From a hardware perspective, vertical microcode is somewhat unattractive. One of the primary disadvantages arises from the performance requirements. Most macro instructions require multiple micro instructions, which means that executing macro instructions at a rate of K per second requires a microcontroller to execute micro instructions at a rate of $N \times K$ per second, where N is the average number of micro instructions per macro instruction. Therefore, hardware associated with the microcontroller must operate at very high speed (e.g., the memory used to hold microcode must be able to deliver micro instructions at a high rate).

A second disadvantage of vertical microcode arises because a vertical technology cannot exploit the parallelism of the underlying hardware. Computer engineers have invented an alternative known as *horizontal microcode* that overcomes some limitations of vertical microcode. Horizontal microcode has the advantage of working well with the hardware, but not providing a familiar interface for programmers. That is:

> *Horizontal microcode allows the hardware to run faster, but is more difficult to program.*

To understand horizontal microcode, recall the data path description from Chapter 6: a CPU consists of multiple functional units, with data paths connecting them. Operation of the units must be controlled, and each unit is controlled independently. Furthermore, moving data from one functional unit to another requires explicit control of the two units: one unit must be instructed to send data across a data path, and the other unit must be instructed to receive data.

An example will clarify the concept. To make the example easy to understand, we will make a few simplifying assumptions and restrict the discussion to six functional units. Figure 8.5 shows how the six functional units are interconnected.

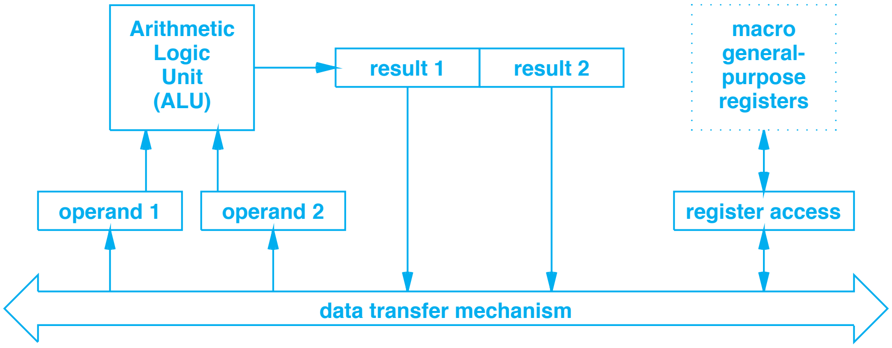


Figure 8.5 An illustration of the internal structure within a CPU. Solid arrows indicate a hardware path along which data can move.

The major item shown in the figure is an *Arithmetic Logic Unit* (*ALU*) that performs operations such as addition, subtraction, and bit shifting. The remaining functional units provide mechanisms that interface the ALU to the rest of the system. For example, the hardware units labeled *operand 1* and *operand 2* denote operand storage units (i.e., internal hardware registers). The ALU expects operands to be placed in the storage units before an operation is performed, and places the result of an operation in the two hardware units labeled *result 1* and *result 2*†. Finally, the *register access* unit provides a hardware interface to the general-purpose registers.

In the figure, arrows indicate paths along which data can pass as it moves from one functional unit to another; each arrow is a data path that handles multiple bits in parallel (e.g., 32 bits). Most of the arrows connect to the *data transfer mechanism*, which serves as a conduit between functional units (a later chapter explains that the data transfer mechanism depicted here is called a *bus*).

8.15 Example Horizontal Microcode

Each functional unit is controlled by a set of wires that carry commands (i.e., binary values that the hardware interprets as a command). Although Figure 8.5 does not show command wires, we can imagine that the number of command wires connected to a functional unit depends on the type of unit. For example, the unit labeled *result 1* only needs a single command wire because the unit can be controlled by a single binary value: zero causes the unit to stop interacting with other units, and one causes the unit to send the current contents of the result unit to the data transfer mechanism. Figure 8.6 summarizes the binary control values that can be passed to each functional unit in our example, and gives the meaning of each.

†Recall that an arithmetic operation, such as multiplication, can produce a result that is twice as large as an operand.

Unit	Command	Meaning
ALU	0 0 0	No operation
	0 0 1	Add
	0 1 0	Subtract
	0 1 1	Multiply
	1 0 0	Divide
	1 0 1	Left shift
	1 1 0	Right shift
	1 1 1	Continue previous operation
operand 1	0	No operation
	1	Load value from data transfer mechanism
operand 2	0	No operation
	1	Load value from data transfer mechanism
result 1	0	No operation
	1	Send value to data transfer mechanism
result 2	0	No operation
	1	Send value to data transfer mechanism
register interface	0 0 x x x x	No operation
	0 1 x x x x	Move register xxxx to data transfer
	1 0 x x x x	Move data transfer to register xxxx
	1 1 x x x x	No operation

Figure 8.6 Possible command values and the meaning of each for the example functional units in Figure 8.5. Commands are carried on parallel wires.

As Figure 8.6 shows, the register access unit is a special case because each command has two parts: the first two bits specify an operation, and the last four bits specify a register to be used in the operation. Thus, the command 0 1 0 0 1 1 means that value in register three should be moved to the data transfer mechanism.

Now that we understand how the hardware is organized, we can see how horizontal microcode works. Imagine that each microcode instruction consists of commands to functional units — when it executes an instruction, the hardware sends bits from the instruction to functional units. Figure 8.7 illustrates how bits of a microcode instruction correspond to commands in our example.

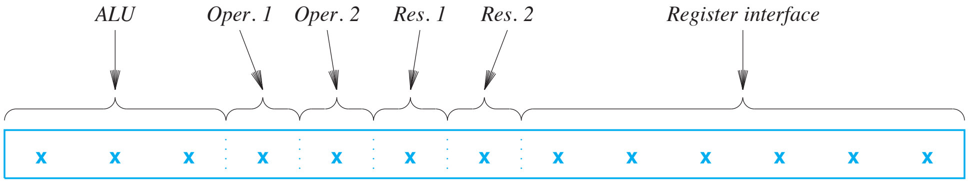


Figure 8.7 Illustration of thirteen bits in a horizontal microcode instruction that correspond to commands for the six functional units.

8.16 A Horizontal Microcode Example

How can horizontal microcode be used to perform a sequence of operations? In essence, a programmer chooses which functional units should be active at any time, and encodes the information in bits of the microcode. For example, suppose a programmer needs to write horizontal microcode that adds the value in general-purpose register 4 to the value in general-purpose register 13 and places the result in general-purpose register 4. Figure 8.8 lists the operations that must be performed.

- Move the value from register 4 to the hardware unit for operand 1
- Move the value from register 13 to the hardware unit for operand 2
- Arrange for the ALU to perform addition
- Move the value from the hardware unit for result 2 (the low-order bits of the result) to register 4

Figure 8.8 An example sequence of steps that the functional units must execute to add values from general-purpose registers 4 and 13, and place the result in general-purpose register 4.

Each of the steps can be expressed as a single micro instruction in our example system. The instruction has bits set to specify which functional unit(s) operate when the instruction is executed. For example, Figure 8.9 shows a microcode program that corresponds to the four steps.

In the figure, each row corresponds to one instruction, which is divided into fields that each correspond to a functional unit. A field contains a command to be sent to the functional unit when the instruction is executed. Thus, commands determine which functional units operate at each step.

Instr.	ALU			OP$_1$	OP$_2$	RES$_1$	RES$_2$	REG. INTERFACE					
1	0	0	0	1	0	0	0	0	1	0	1	0	0
2	0	0	0	0	1	0	0	0	1	1	1	0	1
3	0	0	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	1	0	0	1	0	0

Figure 8.9 An example horizontal microcode program that consists of four instructions with thirteen bits per instruction. Each instruction corresponds to a step listed in Figure 8.8.

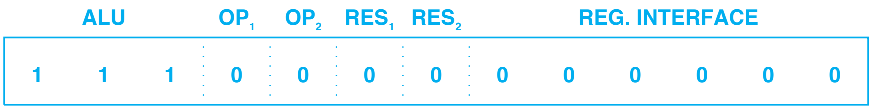
Consider the code in the figure carefully. The first instruction specifies that only two hardware units will operate: the unit for operand 1 and the register interface unit. The fields that correspond to the other four units contain zero, which means that those units will not operate when the first instruction is executed. The first instruction also uses the data transfer mechanism — data is sent across the transfer mechanism from the register interface unit to the unit for operand 1†. That is, fields in the instruction cause the register interface to send a value across the transfer mechanism, and cause the operand 1 unit to receive the value.

8.17 Operations That Require Multiple Cycles

Timing is among the most important aspects of horizontal microcode. Some hardware units take longer to operate than others. For example, multiplication can take longer than addition. That is, when a functional unit is given a command, the results do not appear immediately. Instead, the program must delay before accessing the output from the functional unit.

A programmer who writes horizontal microcode must ensure that each hardware unit is given the correct amount of time to complete its task. The code in Figure 8.9 assumes that each step can be accomplished in one micro instruction cycle. However, a micro cycle may be too short for some hardware units to complete a task. For example, an ALU may require two micro instruction cycles to complete an addition. To accommodate longer computation, an extra instruction can be inserted following the third instruction. The extra instruction merely specifies that the ALU should continue the previous operation; no other units are affected. Figure 8.10 illustrates an extra microcode instruction that can be inserted to create the necessary delay.

†For purposes of this simplified example, we assume the data transfer mechanism always operates and does not require any control.



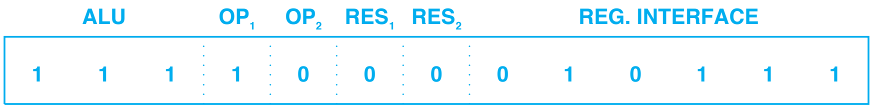
ALU			OP_1	OP_2	RES_1	RES_2	REG. INTERFACE					
1	1	1	0	0	0	0	0	0	0	0	0	0

Figure 8.10 An instruction that can be inserted to add delay processing to wait for the ALU to complete an operation. Timing and delay are crucial aspects of horizontal microcode.

8.18 Horizontal Microcode And Parallel Execution

Now that we have a basic understanding of how hardware operates and a general idea about horizontal microcode, we can appreciate an important property: the use of parallelism. Parallelism is possible because the underlying hardware units operate independently. A programmer can specify parallel operations because an instruction contains separate fields that each control one of the hardware units.

As an example, consider an architecture that has an ALU plus separate hardware units to hold operands. Assume the ALU requires multiple instruction cycles to complete an operation. Because the ALU accesses the operands during the first cycle, the hardware units used to hold operands remain unused during successive cycles. Thus, a programmer can insert an instruction that simultaneously moves a new value into an operand unit while an ALU operation continues. Figure 8.11 illustrates such an instruction.



ALU			OP_1	OP_2	RES_1	RES_2	REG. INTERFACE					
1	1	1	1	0	0	0	0	1	0	1	1	1

Figure 8.11 An example instruction that simultaneously continues an ALU operation and loads the value from register seven into operand hardware unit one. Horizontal microcode makes parallelism easy to specify.

The point is:

> *Because horizontal microcode instructions contain separate fields that each control one hardware unit, horizontal microcode makes it easy to specify simultaneous, parallel operation of the hardware units.*

8.19 Look-Ahead And High Performance Execution

In practice, the microcode used in CPUs is much more complex than the simplistic examples in this chapter. One of the most important sources of complexity arises from the desire to achieve high performance. Because silicon technology allows manufacturers to place billions of transistors on a single chip, it is possible for a CPU to include many functional units that all operate simultaneously.

A later chapter considers architectures that make parallel hardware visible to a programmer. For now, we will consider an architectural question: can multiple functional units be used to improve performance without changing the macro instruction set? In particular, can the internal organization of a CPU be arranged to detect and exploit situations in which parallel execution will produce higher performance?

We have already seen a trivial example of an optimization: Figure 8.11 shows that horizontal microcode can allow an ALU operation to continue at the same time a data value is transferred to a hardware unit that holds an operand. However, our example requires a programmer to explicitly code the parallel behavior when creating the microcode.

To understand how a CPU exploits parallelism automatically, imagine a system that includes an intelligent microcontroller and multiple functional units. Instead of working on one macro instruction at a time, the intelligent controller is given access to many macro instructions. The controller looks ahead at the instructions, finds values that will be needed, and directs functional units to start fetching or computing the values. For example, suppose the intelligent controller finds the following four instructions on a 3-address architecture:

```
add     R1, R3, R7
sub     R4, R4, R6
add     R9, R5, R2
shift   R8, 5
```

We say that an intelligent controller *schedules* the instructions by assigning the necessary work to functional units. For example, the controller can assign each operand to a functional unit that fetches and prepares operand values. Once the operand values are available for an instruction, the controller assigns the instruction to a functional unit that performs the operation. The instructions listed above can each be assigned to an ALU. Finally, when the operation completes, the controller can assign a functional unit the task of moving the result to the appropriate destination register. The point is: if the CPU contains enough functional units, an intelligent controller can schedule all four macro instructions to be executed at the same time.

8.20 Parallelism And Execution Order

Our above description of an intelligent microcontroller overlooks an important detail: the semantics of the macro instruction set. In essence, the controller must ensure that computing values in parallel does not change the meaning of the program. For example, consider the following sequence of instructions:

```
add      R1, R3, R7
sub      R4, R4, R6
add      R9, R1, R2
shift    R8, 5
```

Unlike the previous example, the operands overlap. In particular, the first instruction specifies register one as a destination, and the third instruction specifies register one as an operand. The macro instruction set semantics dictate sequential processing of instructions, which means that the first instruction will place a value in register one *before* the third instruction references the value. To preserve sequential semantics, an intelligent controller must understand and accommodate such overlap. In essence, the controller must balance between two goals: maximize the amount of parallel execution, while preserving the original (i.e., sequential) semantics.

8.21 Out-Of-Order Instruction Execution

How can a controller that schedules parallel activities handle the case where an operand in one instruction depends on the results of a previous instruction? The controller uses a mechanism known as a *scoreboard* that tracks the status of each instruction being executed. In particular, a scoreboard maintains information about dependencies among instructions and the original macro instruction sequence execution. Thus, the controller can use the scoreboard to decide when to fetch operands, when execution can proceed, and when an instruction is finished. In short, the scoreboard approach allows the controller to execute instructions out of order, but then reorders the results to reflect the order specified by the code.

> *To achieve highest speed, a modern CPU contains multiple copies of functional units that permit multiple instructions to be executed simultaneously. An intelligent controller uses a* scoreboard *mechanism to schedule execution in an order that preserves the appearance of sequential processing.*

8.22 Conditional Branches And Branch Prediction

Conditional branches pose another problem for parallel execution. For example, consider the following computation:

```
Y  ←  f(X)
if ( Y > Z ) {
        Q
} else {
        R
}
```

When translated into machine instructions, the computation contains a conditional branch that directs execution either to the code for Q or the code for R. The condition depends on the value of Y, which is computed in the first step. Now consider running the code on a CPU that uses parallel execution of instructions. In theory, once it reaches the conditional branch, the CPU must wait for the results of the comparison — the CPU cannot start to schedule code for R or Q until it knows which one will be selected.

In practice, there are two approaches used to handle conditional branches. The first, which is known as *branch prediction*, is based on measurements which show that in most code, the branch is taken approximately sixty percent of the time. Thus, building hardware that schedules instructions along the branch path provides more optimization than hardware that schedules instructions along the non-branch path. Of course, assuming the branch will occur may be incorrect — if the CPU eventually determines that the branch should not be taken, the results from the branch path must be discarded and the CPU must follow the other path. The second approach simply follows both paths in parallel. That is, the CPU schedules instructions for both outcomes of the conditional branch. As with branch prediction, the CPU must eventually decide which result is valid. That is, the CPU continues to execute instructions, but holds the results internally. Once the value of the condition is known, the CPU discards the results from the path that is not valid, and proceeds to move the correct results into the appropriate destinations. Of course, a second conditional branch can occur in either Q or R; the scoreboard mechanism handles all the details.

The point is:

> *A CPU that offers parallel instruction execution can handle conditional branches by proceeding to precompute values on one or both branches, and choosing which values to use at a later time when the computation of the branch condition completes.*

It may seem wasteful for a CPU to compute values that will be discarded later. However, the goal is higher performance, not elegance. We can also observe that if a CPU is designed to wait until a conditional branch value is known, the hardware will merely sit idle. Therefore, high-speed CPUs, such as those manufactured by Intel and AMD, are designed with parallel functional units and sophisticated scoreboard mechanisms.

8.23 Consequences For Programmers

Can understanding how a CPU is structured help programmers write faster code? In some cases, yes. Suppose a CPU is designed to use branch prediction and that the CPU assumes the branch is taken. A programmer can optimize performance by arranging code so that the most common cases take the branch. For example, if a programmer knows that it will be more common for Y to be less than Z, instead of testing $Y > Z$, a programmer can rewrite the code to test whether $Y \leq Z$.

8.24 Summary

A modern CPU is a complex processor that uses multiple modes of execution to handle some of the complexity. An execution mode determines operational parameters such as the operations that are allowed and the current privilege level. Most CPUs offer at least two levels of privilege and protection: one for the operating system and one for application programs.

To reduce the internal complexity, a CPU is often built with two levels of abstraction: a microcontroller is implemented with digital circuits, and a macro instruction set is created by adding microcode.

There are two broad classes of microcode. A microcontroller that uses vertical microcode resembles a conventional RISC processor. Typically, vertical microcode consists of a set of procedures that each correspond to one macro instruction; the CPU runs the appropriate microcode during the fetch-execute cycle. Horizontal microcode, which allows a programmer to schedule functional units to operate on each cycle, consists of instructions in which each bit field corresponds to a functional unit. A third alternative uses *Field Programmable Gate Array* (*FPGA*) technology to create the underlying system.

Advanced CPUs extend parallel execution by scheduling a set of instructions across multiple functional units. The CPU uses a scoreboard mechanism to handle cases where the results of one instruction are used by a successive instruction. The idea can be extended to conditional branches by allowing parallel evaluation of each path to proceed, and then, once the condition is known, discarding the values along the path that is not taken.

EXERCISES

8.1 If a quad-core CPU chip contains 2 billion transistors, approximately how many transistors are needed for a single core?

8.2 List seven reasons a modern CPU is complex.

8.3 The text says that some CPU chips include a *backward compatibility* mode. Does such a mode offer any advantage to a user?

8.4 Suppose that in addition to other hardware, the CPU used in a smart phone contains additional hardware for three previous versions of the chip (i.e., three backward compatibility modes). What is the disadvantage from a user's point of view?

8.5 Virtualized software systems used in cloud data centers often include a *hypervisor* that runs and controls multiple *operating systems*, and *applications* that each run on one of the operating systems. How do the levels protection used with such systems differ from conventional levels of protection?

8.6 Some manufacturers offer a chip that contains a processor with a basic set of instructions plus an attached FPGA. An owner can configure the FPGA with additional instructions. What does such a chip provide that conventional software cannot?

8.7 Read about FPGAs, and find out how they are "programmed." What languages are used to program an FPGA?

8.8 Create a microcode algorithm that performs 32-bit multiplication on a microcontroller that only offers 16-bit arithmetic, and implement your algorithm in C using *short* variables.

8.9 You are offered two jobs for the same salary, one programming vertical microcode and the other programming horizontal microcode. Which do you choose? Why?

8.10 Find an example of a commercial processor that uses horizontal microcode, and document the meaning of bits for an instruction similar to the diagram in Figure 8.7.

8.11 What is the motivation for a *scoreboard* mechanism in a CPU chip, and what functionality does it provide?

8.12 If Las Vegas casinos computed the odds on program execution, what odds would they give that a branch is taken? Explain your answer.

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

9

Assembly Languages And Programming Paradigm

9.1 Introduction

Previous chapters describe processor instruction sets and operand addressing. This chapter discusses programming languages that allow programmers to specify all the details of instructions and operand addresses. The chapter is not a tutorial about a language for a particular processor. Instead, it provides a general assessment of features commonly found in low-level languages. The chapter examines programming paradigms, and explains how programming in a low-level language differs from programming in a conventional language. Finally, the chapter describes software that translates a low-level language into binary instructions.

Low-level programming and low-level programming languages are not strictly part of computer architecture. We consider them here, however, because such languages are so closely related to the underlying hardware that the two cannot be separated easily. Subsequent chapters return to the focus on hardware by examining memory and I/O facilities.

9.2 Characteristics Of A High-level Programming Language

Programming languages can be divided into two broad categories:

- High-level languages
- Low-level languages

A conventional programming language, such as Java or C, is classified as a *high-level-language* because the language exhibits the following characteristics:

- One-to-many translation
- Hardware independence
- Application orientation
- General-purpose
- Powerful abstractions

One-To-Many Translation. Each statement in a high-level language corresponds to multiple machine instructions. That is, when a compiler translates the language into equivalent machine instructions, a statement usually translates into several instructions.

Hardware Independence. High-level languages allow programmers to create a program without knowing details about the underlying hardware. For example, a high-level language allows a programmer to specify floating point operations, such as addition and subtraction, without knowing whether the ALU implements floating point arithmetic directly or uses a separate floating point coprocessor.

Application Orientation. A high-level language, such as C or Java, is designed to allow a programmer to create application programs. Thus, a high-level language usually includes I/O facilities as well as facilities that permit a programmer to define arbitrarily complex data objects.

General-Purpose. A high-level language, like C or Java, is not restricted to a specific task or a specific problem domain. Instead, the language contains features that allow a programmer to create a program for an arbitrary task.

Powerful Abstractions. A high-level language provides abstractions, such as procedures, that allow a programmer to express complex tasks succinctly.

9.3 Characteristics Of A Low-level Programming Language

The alternative to a high-level language is known as a *low-level language* and has the following characteristics:

- One-to-one translation
- Hardware dependence
- Systems programming orientation
- Special-purpose
- Few abstractions

One-To-One Translation. In general, each statement in a low-level programming language corresponds to a single instruction on the underlying processor. Thus, the translation to machine code is one-to-one.

Hardware Dependence. Because each statement corresponds to a machine instruction, a low-level language created for one type of processor cannot be used with another type of processor.

Systems Programming Orientation. Unlike a high-level language, a low-level language is optimized for systems programming — the language has facilities that allow a programmer to create an operating system or other software that directly controls the hardware.

Special-Purpose. Because they focus on the underlying hardware, low-level languages are only used in cases where extreme control or efficiency is needed. For example, communication with a coprocessor usually requires a low-level language.

Few Abstractions. Unlike high-level languages, low-level languages do not provide complex data structures (e.g., strings or objects) or control statements (e.g., *if-then-else* or *while*). Instead, the language forces a programmer to construct abstractions from low-level hardware mechanisms†.

9.4 Assembly Language

The most widely used form of low-level programming language is known as *assembly language*, and the software that translates an assembly language program into a binary image that the hardware understands is known as an *assembler*.

It is important to understand that the phrase *assembly language* differs from phrases such as *Java language* or *C language* because *assembly* does not refer to a single language. Instead, a given assembly language uses the instruction set and operands from a single processor. Thus, many assembly languages exist, one for each processor. Programmers might talk about *MIPS assembly language* or *Intel x86 assembly language*. To summarize:

> *Because an assembly language is a low-level language that incorporates specific characteristics of a processor, such as the instruction set, operand addressing, and registers, many assembly languages exist.*

The consequence for programmers should be obvious: when moving from one processor to another, an assembly language programmer must learn a language. On the down side, the instruction set, operand types, and register names often differ among assembly languages. On the positive side, most assembly languages tend to follow the same basic pattern. Therefore, once a programmer learns one assembly language, the programmer can learn others quickly. More important, if a programmer understands the basic assembly language paradigm, moving to a new architecture usually involves learning new details, not learning a new programming style. The point is:

†Computer scientist Alan Perlis once quipped that a programming language is low-level if programming requires attention to irrelevant details. His point is that because most applications do not need direct control, using a low-level language creates overhead for an application programmer without any real benefit.

Despite differences, many assembly languages share the same fundamental structure. Consequently, a programmer who understands the assembly programming paradigm can learn a new assembly language quickly.

To help programmers understand the concept of assembly language, the next sections focus on general features and programming paradigms that apply to most assembly languages. In addition to specific language details, we will discuss concepts such as macros.

9.5 Assembly Language Syntax And Opcodes

9.5.1 Statement Format

Because assembly language is low-level, a single assembly language statement corresponds to a single machine instruction. To make the correspondence between language statements and machine instructions clear, most assemblers require a program to contain a single statement per line of input. The general format is:

label: opcode operand_1, operand_2, ...

where *label* gives an optional label for the statement (used for branching), *opcode* specifies one of the possible instructions, each *operand* specifies an operand for the instruction, and whitespace separates the opcode from other items.

9.5.2 Opcode Names

The assembly language for a given processor defines a symbolic name for each instruction that the processor provides. Although the symbolic names are intended to help a programmer remember the purpose of the instruction, most assembly languages use extremely short abbreviations instead of long names. Thus, if a processor has an instruction for addition, the assembly language might use the opcode *add*. However, if the processor has an instruction that branches to a new location, the opcode for the instruction typically consists of a single letter, *b*, or the two-letter opcode *br*. Similarly, if the processor has an instruction that jumps to a subroutine, the opcode is often *jsr*.

Unfortunately, there is no global agreement on opcode names even for basic operations. For example, most architectures include an instruction that copies the contents of one register to another. To denote such an operation, some assembly languages use the opcode *mov* (an abbreviation for *move*), and others use the opcode *ld* (an abbreviation for *load*).

9.5.3 Commenting Conventions

Short opcodes tend to make assembly language easy to write but difficult to read. Furthermore, because it is low-level, assembly language tends to require many instructions to achieve a straightforward task. Thus, to ensure that assembly language programs remain readable, programmers add two types of comments: block comments that explain the purpose of each major section of code, and a detailed comment on each individual line to explain the purpose of the line.

To make it easy for programmers to add comments, assembly languages often allow comments to extend until the end of a line. That is, the language only defines a character (or sequence of characters) that starts a comment. One commercial assembly language defines the pound sign character (#) as the start of a comment, a second uses a semicolon to denote the start of a comment, and a third has adopted the C++ comment style and uses two adjacent slash characters. A block comment can be created in which each line begins with the comment character, and a detailed comment can be added to each line of the program. Programmers often add additional characters to surround a block comment. For example, if the pound sign signals the start of a comment, the block comment below explains that a section of code searches a list to find a memory block of a given size:

```
    #################################################################
    #                                                               #
    #   Search linked list of free memory blocks to find a block    #
    #   of size N bytes or greater.  Pointer to list must be in     #
    #   register 3, and N must be in register 4.  The code also     #
    #   destroys the contents of register 5, which is used to       #
    #   walk the list.                                              #
    #                                                               #
    #################################################################
```

Most programmers place a comment on each line of assembly code to explain how the instruction fits into the algorithm. For example, the code to search for a memory block might begin:

```
            ld      r5,r3     # load the address of list into r5
    loop_1: cmp     r5,r0     # test to see if at end of list
            bz      notfnd    # if reached end of list go to notfnd
            . . .
```

Although details in the example above may seem obscure, the point is relatively straightforward: a block comment before a section of code explains *what* the code accomplishes, and a comment on each line of code explains *how* that particular instruction contributes to the result.

9.6 Operand Order

One frustrating difference among assembly languages causes subtle problems for programmers who move from one assembly language to another: the order of operands. A given assembly language usually chooses a consistent operand order. For example, consider a load instruction that copies the contents of one register to another register. In the example code above, the first operand represents the *target* register (i.e., the register into which the value will be placed), and the second operand represents the *source* register (i.e., the register from which the value will be copied). Under such an interpretation, the statement:

```
ld      r5,r3     # load the address of list into r5
```

copies the contents of register 3 into register 5. As a mnemonic aid to help them remember the right-to-left interpretation, programmers are told to think of an assignment statement in which the expression is on the right and the target of the assignment is on the left.

As an alternative to the example code, some assembly languages specify the opposite order — the source register is on the left and the target register is on the right. In such assembly languages, the code above is written with operands in the opposite order:

```
ld      r3,r5     # load the address of list into r5
```

As a mnemonic aid to help them remember the left-to-right interpretation, programmers are told to think of a computer reading the instruction. Because text is read left to right, we can imagine the computer reading the opcode, picking up the first operand, and depositing the value in the second operand. Of course, the underlying hardware does not process the instruction left-to-right or right-to-left — the operand order is only assembly language syntax.

Operand ordering is further complicated by several factors. First, unlike our examples above, many assembly language instructions do not have two operands. For example, an instruction that performs bitwise complement only needs one operand. Furthermore, even if an instruction has two operands, the notions of source and destination may not apply (e.g., a comparison). Therefore, a programmer who is unfamiliar with a given assembly language may need to consult a manual to find the order of operands for a given opcode.

Of course, there can be a significant difference between what a programmer writes and the resulting binary value of the instruction because the assembly language merely uses notation that is convenient for the programmer. The assembler can reorder operands during translation. For example, the author once worked on a computer that had two assembly languages, one produced by the computer's vendor and another produced by researchers at Bell Labs. Although both languages were used to produce code for the same underlying computer, one language used a left-to-right interpretation of the operands, and the other used a right-to-left interpretation.

9.7 Register Names

Because a typical instruction includes a reference to at least one register, most assembly languages include a special way to denote registers. For example, in many assembly languages, names that consist of the letter *r* followed by one or more digits are reserved to refer to registers. Thus, a reference to *r10* refers to register 10.

However, there is no universal standard for register references. In one assembly language, all register references begin with a dollar sign followed by digits; thus, $10 refers to register 10. Other assemblers are more flexible: the assembler allows a programmer to choose register names. That is, a programmer can insert a series of declarations that define a specific name to refer to a register. Thus, one might find declarations such as:

```
#
#  Define register names used in the program
#
r1      register 1          # define name r1 to be register 1
r2      register 2          #   and so on for r2, r3, and r4
r3      register 3
r4      register 4
```

The chief advantage of allowing programmers to define register names arises from increased readability: a programmer can choose meaningful names. For example, suppose a program manages a linked list. Instead of using numbers or names like *r6*, a programmer can give meaningful names to the registers:

```
#
#  Define register names for a linked list program
#
listhd   register 6         # holds starting address of list
listptr  register 7         # moves along the list
```

Of course, allowing programmers to choose names for registers can also lead to unexpected results that make the code difficult to understand. For example, consider reading a program in which a programmer has used the following declaration:

```
r3      register 8          # define name r3 to be register 8!
```

The points can be summarized:

> *Because registers are fundamental to assembly language programming, each assembly language provides a way to identify registers. In some languages, special names are reserved; in others, a programmer can assign a name to a register.*

9.8 Operand Types

As Chapter 7 explains, a processor often provides multiple types of operands. The assembly language for each processor must accommodate all operand types that the hardware offers. As an example, suppose a processor allows each operand to specify a register, an immediate value (i.e., a constant), a memory location, or a memory location specified by adding an offset in the instruction to the contents of a register. The assembly language for the processor needs a syntactic form for each possible operand type.

We said that assembly languages often use special characters or names to distinguish registers from other values. In many assembly languages, for example, *10* refers to the constant ten, and *r10* refers to register ten. However, some assembly languages require a special symbol before a constant (e.g., *#10* to refer to the constant ten).

Each assembly language must provide syntactic forms for each possible operand type. Consider, for example, copying a value from a source to a target. If the processor allows the instruction to specify either a register (direct) or a memory location (indirect) as the source, the assembly language must provide a way for a programmer to distinguish the two. One assembly language uses parentheses to distinguish the two possibilities:

```
mov     r2,r1       # copy contents of reg. 1 into reg. 2
mov     r2,(r1)     # treat r1 as a pointer to memory and
                    # copy from the mem. location to reg. 2
```

The point is:

> *An assembly language provides a syntactic form for each possible operand type that the processor supports, including a reference to a register, an immediate value, and an indirect reference to memory.*

9.9 Assembly Language Programming Paradigm And Idioms

Because a programming language provides facilities that programmers use to structure data and code, a language can impact the programming process and the resulting code. Assembly language is especially significant because the language does not provide high-level constructs nor does the language enforce a particular style. Instead, assembly language gives a programmer complete freedom to code arbitrary sequences of instructions and store data in arbitrary memory locations.

Experienced programmers understand that consistency and clarity are usually more important than clever tricks or optimizations. Thus, experienced programmers develop idioms: patterns that they use consistently. The next sections use basic control structures to illustrate the concept of assembly language idioms.

9.10 Coding An IF Statement In Assembly

We use the term *conditional execution* to refer to code that may or may not be executed, depending on a certain condition. Because conditional execution is a fundamental part of programming, high-level languages usually include one or more statements that allow a programmer to express conditional execution. The most basic form of conditional execution is known as an *if* statement.

In assembly language, a programmer must code a sequence of statements to perform conditional execution. Figure 9.1 illustrates the form used for conditional execution in a typical high-level language and the equivalent form used in a typical assembly language.

```
if (condition) {                  code to test the condition and
        body                         set the condition code
}                                 branch to label if condition false
next statement;                   code to perform body
                          label:  code for next statement

        (a)                                (b)
```

Figure 9.1 (a) Conditional execution as specified in a high-level language, and (b) the equivalent assembly language code.

As the figure indicates, some processors use a *condition code* as the fundamental mechanism for conditional execution. Whenever it performs an arithmetic operation or a comparison, the ALU sets the condition code. A conditional branch instruction can be used to test the condition code and execute the branch if the condition code matches the instruction. Note that in the case of emulating an *if* statement, the branch instruction must test the opposite of the condition (i.e., the branch is taken if the condition is *not* met). For example, consider the statement:

```
if (a == b) { x }
```

If we assume *a* and *b* are stored in registers five and six, the equivalent assembly language is:

```
        cmp     r5, r6      # compare the values of a and b and set cc
        bne     lab1        # branch if previous comparison not equal
        code for x
          . . .
lab1:   code for next statement
```

9.11 Coding An IF-THEN-ELSE In Assembly

The *if-then-else* statement found in high-level languages specifies code to be executed for both the case when a condition is true and when the condition is false. Figure 9.2 shows the assembly language equivalent of an *if-then-else* statement.

```
if (condition) {
        then_part
} else {
        else_part
}
next statement;
```

(a)

```
          code to test the condition and
              set the condition code
          branch to label1 if condition false
          code to perform then_part
          branch to label2
label1:   code for else_part
label2:   code for next statement
```

(b)

Figure 9.2 (a) An *if-then-else* statement used in a high-level language, and (b) the equivalent assembly language code.

9.12 Coding A FOR-LOOP In Assembly

The term *definite iteration* refers to a programming language construct that causes a piece of code to be executed a fixed number of times. A typical high-level language uses a *for* statement to implement definite iteration. Figure 9.3 shows the assembly language equivalent of a *for* statement.

Definite iteration illustrates an interesting difference between a high-level language and assembly language: location of code. In assembly language, the code to implement a control structure can be divided into separate locations. In particular, although a programmer thinks of the initialization, continuation test, and increment as being specified in the header of a *for* statement, the equivalent assembly code places the increment after the code for the body.

9.13 Coding A WHILE Statement In Assembly

In programming language terminology, *indefinite iteration* refers to a loop that executes zero or more times. Typically, a high-level language uses the keyword *while* to indicate indefinite iteration. Figure 9.4 shows the assembly language equivalent of a *while* statement.

```
for (i=0; i<10; i++) {                    set r4 to zero
        body                      label1: compare r4 to 10
}                                         branch to label2 if >=
next statement;                           code to perform body
                                          increment r4
                                          branch to label1
                                  label2: code for next statement

         (a)                                      (b)
```

Figure 9.3 (a) A *for* statement used in a high-level language, and (b) the equivalent assembly language code using register 4 as an index.

```
while (condition) {               label1: code to compute condition
        body                              branch to label2 if false
}                                         code to perform body
next statement;                           branch to label1
                                  label2: code for next statement

         (a)                                      (b)
```

Figure 9.4 (a) A *while* statement used in a high-level language, and (b) the equivalent assembly language code.

9.14 Coding A Subroutine Call In Assembly

We use the term *procedure* or *subroutine* to refer to a piece of code that can be invoked, perform a computation, and return control to the invoker. The terms *procedure call* or *subroutine call* to refer to the invocation. The key idea is that when a subroutine is invoked, the processor records the location from which the call occurred, and resumes execution at that point once the subroutine completes. Thus, a given subroutine can be invoked from multiple points in a program because control always passes back to the location from which the invocation occurred.

Many processors provide two basic assembly instructions for procedure invocation. A *jump to subroutine* (*jsr*) instruction saves the current location and branches to a subroutine at a specified location, and a *return from subroutine* (*ret*) instruction causes the processor to return to the previously saved location. Figure 9.5 shows how the two assembly instructions can be used to code a procedure declaration and two invocations.

```
x() {                          x:  code for body of x
        body of function x         ret
}

x();                               jsr  x
other statement;                   code for other statement
x();                               jsr  x
next statement;                    code for next statement

          (a)                                (b)
```

Figure 9.5 (a) A declaration for procedure *x* and two invocations in a high-level language, and (b) the assembly language equivalent.

9.15 Coding A Subroutine Call With Arguments In Assembly

In a high-level language, procedure calls are *parameterized*. The procedure body is written with references to parameters, and the caller passes a set of values to the procedure that are known as *arguments*. When the procedure refers to a parameter, the value is obtained from the corresponding argument. The question arises: how are arguments passed to a procedure in assembly code?

Unfortunately, the details of argument passing vary widely among processors. For example, each of following three schemes has been used in at least one processor†:

- The processor uses a stack in memory for arguments
- The processor uses register windows to pass arguments
- The processor uses special-purpose argument registers

As an example, consider a processor in which registers *r1* through *r8* are used to pass arguments during a procedure call. Figure 9.6 shows the assembly language code for a procedure call on such an architecture.

9.16 Consequence For Programmers

The consequence of a variety of argument passing schemes should be clear: the assembly language code needed to pass and reference arguments varies significantly from one processor to another. More important, programmers are free to invent new mechanisms for argument passing that optimize performance. For example, memory references are slower than register references. Thus, even if the hardware is designed to use a stack in memory, a programmer might choose to increase performance by passing some arguments in general-purpose registers rather than memory.

†The storage used for a *return address* (i.e., the location to which a *ret* instruction should branch) is often related to the storage used for arguments.

```
x ( a, b )  {                        x:  code for body of x that assumes
        body of function x                   register 1 contains parameter a
}                                            and register 2 contains b
                                         ret


x ( -4, 17 );                            load -4 into register 1
other statement;                         load 17 into register 2
x ( 71, 27  );                           jsr  x
next statement                           code for other statement
                                         load 71 into register 1
                                         load 27 into register 2
                                         jsr  x
                                         code for next statement

           (a)                                      (b)
```

Figure 9.6 (a) A declaration for parameterized procedure *x* and two invocations in a high-level language, and (b) the assembly language equivalent for a processor that passes arguments in registers.

The point is:

> *No single argument passing paradigm is used in assembly languages because a variety of hardware mechanisms exist for argument passing. In addition, programmers sometimes use alternatives to the basic mechanism to optimize performance (e.g., passing values in registers).*

9.17 Assembly Code For Function Invocation

The term *function* refers to a procedure that returns a single-value result. For example, an arithmetic function can be created to compute *sine(x)* — the argument specifies an angle, and the function returns the sine of the angle. Like a procedure, a function can have arguments, and a function can be invoked from an arbitrary point in the program. Thus, for a given processor, function invocation uses the same basic mechanisms as procedure invocation.

Despite the similarities between functions and procedures, function invocation requires one additional detail: an agreement that specifies exactly how the function result is returned. As with argument passing, many alternative implementations exist. Processors have been built that provide a separate, special-purpose hardware register for a function return value. Other processors assume that the program will use one of the general-purpose registers. In any case, before executing a *ret* instruction, a function must load the return value into the location that the processor uses. After the return occurs, the calling program extracts and uses the return value.

9.18 Interaction Between Assembly And High-level Languages

Interaction is possible in either direction between code written in an assembly language and code written in a high-level language. That is, a program written in a high-level language can call a procedure or function that has been written in assembly language, and a program written in assembly language can call a procedure or function that has been written in a high-level language. Of course, because a programmer can only control the assembly language code and not the high-level language code, the assembly program must follow the *calling conventions* that the high-level language uses. That is, the assembly code must use exactly the same mechanisms as the high-level language uses to store a return address, invoke a procedure, pass arguments, and return a function value.

Why would a programmer mix code written in assembly language with code written in a high-level language? In some cases, assembly code is needed because a high-level language does not allow direct interaction with the underlying hardware. For example, a computer that has special graphics hardware may need assembly code to use the graphics functions. In most cases, however, assembly language is only used to optimize performance — once a programmer identifies a particular piece of code as a bottleneck, the programmer writes an optimized version of the code in assembly language. Typically, optimized assembly language code is placed into a procedure or function; the rest of the program remains written in a high-level language. As a result, the most common case of interaction between code written in a high-level language and code written in assembly language consists of a program written in a high-level language calling a procedure or function that is written in an assembly language.

The point is:

> *Because writing application programs in assembly language is difficult, assembly language is reserved for situations where a high-level language has insufficient functionality or results in poor performance.*

9.19 Assembly Code For Variables And Storage

In addition to statements that generate instructions, assembly languages permit a programmer to define data items. Both initialized and uninitialized variables can be declared. For example, some assembly languages use the directive *.word* to declare storage for a sixteen-bit item, and the directive *.long* to declare storage for a thirty-two-bit item. Figure 9.7 shows declarations in a high-level language and equivalent assembly code.

```
int     x, y, z;             x:   .long
                             y:   .long
                             z:   .long
short   w, q;                w:   .word
                             q:   .word
statement(s)                      statement(s)

          (a)                        (b)
```

Figure 9.7 (a) Declaration of variables in a high-level language, and (b) equivalent variable declarations in assembly language.

The keywords *.word* and *.long* are known as assembly language *directives*. Although it appears in the same location that an opcode appears, a directive does not correspond to an instruction. Instead, a directive controls the translation. The directives in the figure specify that storage locations should be reserved to hold variables. In most assembly languages, a directive that reserves storage also allows a programmer to specify an initial value. Thus, the directive:

```
x:  .word  949
```

reserves a sixteen bit memory location, assigns the location the integer value 949, and defines *x* to be a *label* (i.e., a name) that the programmer can use to refer to the location.

9.20 Example Assembly Language Code

An example will help clarify the concepts and show how assembly language idioms apply in practice. To help compare x86 and ARM architectures, we will use the same example for each architecture. To make the example clear, we begin with a C program and then show how the same algorithm can be implemented in assembly language.

Instead of using a long, complex program to show all the idioms, we will use a trivial example that demonstrates a few basics. In particular, it will show indefinite iteration and conditional execution. The example consists of a piece of code that prints an initial list of the *Fibonacci sequence*. The first two values in the sequence are each 1. Each successive value is computed as the sum of the preceding two values. Thus, the sequence is 1, 1, 2, 3, 5, 8, 13, 21, and so on.

To ensure our example relates to concepts from computer architecture, we will arrange the code to print all values in the Fibonacci sequence that fit into a two's complement thirty-two-bit signed integer. As the sequence is generated, the code will count the number of values greater than 1000, and will print a summary.

9.20.1 The Fibonacci Example In C

Figure 9.8 shows a C program that computes each value in the Fibonacci sequence that fits in a thirty-two-bit signed integer. The program uses printf to print each value. It also counts the number of values greater than 1000, and uses printf to print the total as well as a summary of the final values of variables that are used in the computation.

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

int     a = 1, b = 1, n, tmp;

void    main(void) {

        n = 0;
        printf(" %10d\n", b);
        printf(" %10d\n", a);
        while ( (tmp = a + b) > 0 ) {
                b = a;
                a = tmp;
                if (a > 1000) {
                        n++;
                }
                printf(" %10d\n", a);
        }

        printf("\nThe number of values greater than 1000 is %d\n", n);
        printf("Final values are: a=0x%08X b=0x%08X tmp=0x%08X\n",a,b,tmp);
        exit(0);
}
```

Figure 9.8 An example C program that computes and prints values in the Fibonacci sequence that fit into a thirty-two-bit signed integer.

Figure 9.9 shows the output that results when the program runs. The last line of the output gives the value of variables *a*, *b*, and *tmp* after the *while* loop finishes. Variable *a* (1,836,311,903 in decimal) is 6D73E55F in hex. Notice that variable *tmp* has value B11924E1, which has the high-order bit set. As Chapter 3 explains, when *tmp* is interpreted as a signed integer, the value will be negative, which is why the loop terminated. Also note that variable *n*, which counts the number of Fibonacci values has the final value 30; the value can be verified by counting lines of output with values greater than 1000.

```
          1
          1
          2
          3
          5
          8
         13
         21
         34
         55
         89
        144
        233
        377
        610
        987
       1597
       2584
       4181
       6765
      10946
      17711
      28657
      46368
      75025
     121393
     196418
     317811
     514229
     832040
    1346269
    2178309
    3524578
    5702887
    9227465
   14930352
   24157817
   39088169
   63245986
  102334155
  165580141
  267914296
  433494437
  701408733
 1134903170
 1836311903

The number of values greater than 1000d is 30
Final values are: a=0x6D73E55F b=0x43A53F82 tmp=0xB11924E1
```

Figure 9.9 The output that results from running the program in Figure 9.8.

9.20.2 The Fibonacci Example In x86 Assembly Language

Figure 9.10 shows x86 assembly code that generates the same output as the program in Figure 9.8. The code uses the gcc calling conventions to call *printf*.

```
        .data

a:      .long   1               # initialized data (a and b)
b:      .long   1
        .comm   n,4,4           # uninitialized data (n and tmp)
        .comm   tmp,4,4

fmt1:   .string " %10d\n"
fmt2:   .string "\nThe number of values greater than 1000 is %d\n"
fmt3:   .string "Final values are: a=0x%08X b=0x%08X tmp=0x%08X\n"

        .text
        .globl  main
main:
        movl    $0, n           # n = 0

        movl    b, %esi         # set up args to print a
        movl    $fmt1, %edi
        movl    $0, %eax
        call    printf

        movl    a, %esi         # set up args to print b
        movl    $fmt1, %edi
        movl    $0, %eax
        call    printf

while:
        movl    a,%eax          # eax <- a
        addl    b,%eax          # eax <- eax + b
        movl    %eax,tmp        # tmp <- eax
        testl   %eax, %eax      # test eax
        jle     endwhile        # if <= 0 jump to endwhile

        movl    a, %eax         # eax <- a
        movl    %eax, b         # b <- eax
        movl    tmp, %eax       # eax <- tmp
        movl    %eax, a         # a <- eax

        cmpl    $1000, %eax     # compare 1000 to eax
        jle     endif           # if <= jump to endif
        movl    n, %ebx         # ebx <- n
        addl    $1, %ebx        # ebx <- ebx + 1
        movl    %ebx, n         # n <- ebx
```

```
endif:

        movl    a, %esi         # set up args to print a
        movl    $fmt1, %edi
        movl    $0, %eax
        call    printf
        jmp     while

endwhile:
        movl    n, %esi         # set up args to print n
        movl    $fmt2, %edi
        movl    $0, %eax
        call    printf

        movl    tmp, %ecx       # set up args to print a, b, and tmp
        movl    b, %edx
        movl    a, %esi
        movl    $fmt3, %edi
        movl    $0, %eax
        call    printf

        movl    $0, %edi        # exit with argument 0
        call    exit
```

Figure 9.10 An x86 assembly language program that follows the C program shown in Figure 9.8.

9.20.3 The Fibonacci Example In ARM Assembly Language

Figure 9.11 shows ARM assembly code that generates the same output as the program in Figure 9.8. Neither the x86 nor the ARM code has been optimized. In each case, instructions can be eliminated by keeping variables in registers. As an example, a small amount of optimization has been done for the ARM code: registers *r4* through *r8* are initialized to contain the addresses of variables *a*, *b*, *n*, *tmp*, and the format string *fmt1*. The registers remain unchanged while the program runs because called subprograms are required to save and restore values. Thus, when calling *printf* to print variable *a*, the code can use a single instruction to move the address of the format into the first argument register (*r0*):

```
mov    r0, r8
```

The code can also use a single instruction to load the value of *a* into the second argument register (*r1*):

```
ldr    r1, [r4]
```

Exercises suggest ways to improve the code.

```
        .text
        .align  4
        .global main
main:
        movw    r4, #:lower16:a         @ r4 <- &a
        movt    r4, #:upper16:a
        movw    r5, #:lower16:b         @ r5 <- &b
        movt    r5, #:upper16:b
        movw    r6, #:lower16:n         @ r6 <- &n
        movt    r6, #:upper16:n
        movw    r7, #:lower16:tmp       @ r7 <- &tmp
        movt    r7, #:upper16:tmp
        movw    r8, #:lower16:fmt1      @ r8 <- &fmt1
        movt    r8, #:upper16:fmt1

        mov     r0, #0
        str     r0, [r6]                @ n = 0

        ldr     r1, [r5]                @ r1 <- b
        mov     r0, r8                  @ r0 <- &fmt1
        bl      printf

        ldr     r1, [r4]                @ r1 <- a
        mov     r0, r8                  @ r0 <- &fmt1
        bl      printf

while:
        ldr     r3, [r4]                @ r3 <- a
        ldr     r2, [r5]                @ r2 <- b
        add     r1, r3, r2              @ r1 <- a + b
        str     r1, [r7]                @ tmp <- r1  (i.e., tmp <- a + b)
        cmp     r1, #0                  @ test tmp
        ble     endwhile                @ if tmp <= 0 go to endwhile

        str     r3, [r5]                @ b <- a
        str     r1, [r4]                @ a <- tmp

        cmp     r1, #1000               @ compare a and 1000
        ldrgt   r3, [r6]                @ if a>1000 r3 <- n
        addgt   r3, r3, #1              @ if a>1000 r3 <- r3 + 1
        strgt   r3, [r6]                @ if a>1000 n <- r3
        mov     r0, r8                  @ r0 <- &fmt1
        bl      printf                  @ r1 is still a

        b       while

endwhile:

        movw    r0, #:lower16:fmt2
        movt    r0, #:upper16:fmt2      @ r0 <- &fmt2
        ldr     r1, [r6]                @ r1 <- n
        bl      printf

        ldr     r3, [r7]                @ r3 <- tmp
        ldr     r2, [r5]                @ r2 <- b
        ldr     r1, [r4]                @ r1 <- a
        movw    r0, #:lower16:fmt3
        movt    r0, #:upper16:fmt3      @ r0 <- &fmt3
        bl      printf

        mov     r0, #0
        bl      exit                    @ exit with argument 0

        .align 4

        .comm   tmp,4,4                 @ uninitialized data
        .comm   n,4,4
        .data
        .align  4

b:      .word   1                       @ initialized data
a:      .word   1

fmt1:   .ascii  " %10d\012\000"
fmt2:   .ascii  "\012The number of values greater than 1000 is %d\012\000"
fmt3:   .ascii  "Final values are: a=0x%08X b=0x%08X tmp=0x%08X\012\000"
```

Figure 9.11 An ARM assembly language program that follows the algorithm shown in Figure 9.8.

9.21 Two-Pass Assembler

We use the term *assembler* to refer to a piece of software that translates assembly language programs into binary code for the processor to execute. Conceptually, an assembler is similar to a compiler because each takes a source program as input and produces equivalent binary code as output. An assembler differs from a compiler, however, because a compiler has significantly more responsibility. For example, a compiler can choose how to allocate variables to memory, which sequence of instructions to use for each statement, and which values to keep in general-purpose registers. An assembler cannot make such choices because the source program specifies the exact details. The difference between an assembler and compiler can be summarized:

> *Although both a compiler and an assembler translate a source program into equivalent binary code, a compiler has more freedom to choose which values are kept in registers, the instructions used to implement each statement, and the allocation of variables to memory. An assembler merely provides a one-to-one translation of each statement in the source program to the equivalent binary form.*

Conceptually, an assembler follows a *two-pass algorithm*, which means the assembler scans through the source program two times. To understand why two passes are needed, observe that many branch instructions contain *forward references* (i.e., the label referenced in the branch is defined later in the program). When the assembler first reaches a branch statement, the assembler cannot know which address will be associated with the label. Thus, the assembler makes an initial pass, computes the address that each label will have in the final program, and stores the information in a table known as a *symbol table*. The assembler then makes a second pass to generate code. Figure 9.12 illustrates the idea by showing a snippet of assembly language code and the relative location of statements.

locations			assembly code		
0x00	–	0x03	x:	.long	
0x04	–	0x07	label1:	cmp	r1, r2
0x08	–	0x0B		bne	label2
0x0C	–	0x0F		jsr	label3
0x10	–	0x13	label2:	load	r3, 0
0x14	–	0x17		br	label4
0x18	–	0x1B	label3:	add	r5, 1
0x1C	–	0x1F		ret	
0x20	–	0x23	label4:	ld	r1, 1
0x24	–	0x27		ret	

Figure 9.12 A snippet of assembly language code and the locations assigned to each statement for a hypothetical processor. Locations are determined in the assembler's first pass.

During the first pass, the assembler computes the size of instructions without actually filling in details. Once the assembler has completed its first pass, the assembler will have recorded the location for each statement. Consequently, the assembler knows the value for each label in the program. In the figure, for example, the assembler knows that *label4* starts at location 0x20 (32 in decimal). Thus, when the second pass of the assembler encounters the statement:

```
br  label4
```

the assembler can generate a branch instruction with 32 as an immediate operand. Similarly, code can be generated for each of the other branch instructions during the second pass because the location of each label is known.

It is not important to understand the details of an assembler, but merely to know that:

> *Conceptually, an assembler makes two passes over an assembly language program. During the first pass, the assembler assigns a location to each statement. During the second pass, the assembler uses the assigned locations to generate code.*

Now that we understand how an assembler works, we can discuss one of the chief advantages of using an assembler: automatic recalculation of branch addresses. To see how automatic recalculation helps, consider a programmer working on a program. If the programmer inserts a statement in the program, the location of each successive statement changes. As a result, every branch instruction that refers to a label beyond the insertion point must be changed.

Without an assembler, changing branch labels can be tedious and prone to errors. Furthermore, programmers often make a series of changes while debugging a program. An assembler allows a programmer to make a change easily — the programmer merely reruns the assembler to produce a binary image with all branch addresses updated.

9.22 Assembly Language Macros

Because assembly language is low-level, even trivial operations can require many instructions. More important, an assembly language programmer often finds that sequences of code are repeated with only minor changes between instances. Repeated sequences of code make programming tedious, and can lead to errors if a programmer uses a cut-and-paste approach.

To help programmers avoid repetitious coding, many assembly languages include a *parameterized macro* facility. To use a macro facility, a programmer adds two types of items to the source program: one or more macro *definitions* and one or more macro *expansions*. Note: C programmers will recognize assembly language macros because they operate like C preprocessor macros.

In essence, a macro facility adds an extra pass to the assembler. The assembler makes an initial pass in which macros are expanded. The important concept is that the macro expansion pass does not parse assembly language statements and does not handle translation of the instructions. Instead, the macro processing pass takes as input an assembly language source program that contains macros, and produces as output an assembly language source program in which macros are expanded. That is, the output of the macro preprocessing pass becomes the input to the normal two-pass assembler. Many assemblers have an option that allows a programmer to obtain a copy of the expanded source code for use in debugging (i.e., to see if macro expansion is proceeding as the programmer planned).

Although the details of assembly language macros vary across assembly languages, the concept is straightforward. A macro definition is usually bracketed by keywords (e.g., *macro* and *endmacro*), and contains a sequence of code. For example, Figure 9.13 illustrates a definition for a macro named *addmem* that adds the contents of two memory locations and places the result in a third location.

```
macro  addmem(a, b, c)
load   r1, a    # load 1st arg into register 1
load   r2, b    # load 2nd arg into register 2
add    r1, r2   # add register 2 to register 1
store  r3, c    # store the result in 3rd arg
endmacro
```

Figure 9.13 An example macro definition using the keywords *macro* and *endmacro*. Items in the macro refer to parameters *a*, *b*, and *c*.

Once a macro has been defined, the macro can be expanded. A programmer invokes the macro and supplies a set of arguments. The assembler replaces the macro call with a copy of the body of the macro, substituting actual arguments in place of formal parameters. For example, Figure 9.14 shows the assembly code generated by an expansion of the *addmem* macro defined in Figure 9.13.

```
#
# note: code below results from addmem(xxx, YY, zqz)
#
load   r1, xxx  # load 1st arg into register 1
load   r2, YY   # load 2nd arg into register 2
add    r1, r2   # add register 2 to register 1
store  r3, zqz  # store the result in 3rd arg
```

Figure 9.14 An example of the assembly code that results from an expansion of macro *addmem*.

It is important to understand that although the macro definition in Figure 9.13 resembles a procedure declaration, a macro does not operate like a procedure. First, the declaration of a macro does not generate any machine instructions. Second, a macro is expanded, not called. That is, a complete copy of the macro body is copied into the assembly program. Third, macro arguments are treated as strings that replace the corresponding parameter. The literal substitution of arguments is especially important to understand because it can yield unexpected results. For example, consider Figure

9.15 which illustrates how an illegal assembly program can result from a macro expansion.

```
#
# note: code below results from addmem(1+, %*J , +)
#
load   r1, 1+   # load 1st arg into register 1
load   r2, %*J  # load 2nd arg into register 2
add    r1, r2   # add register 2 to register 1
store  r3, +    # store the result in 3rd arg
```

Figure 9.15 An example of an illegal program that can result from an expansion of macro *addmem*. The assembler substitutes arguments without checking their validity.

As the figure shows, an arbitrary string can be used as an argument to the macro, which means a programmer can inadvertently make a mistake. No warning is issued until the assembler processes the expanded source program. For example, the first argument in the example consists of the string *1+*, which is a syntax error. When it expands the macro, the assembler substitutes the specified string which results in:

```
load  r1, 1+
```

Similarly, substitution of the second argument, *%*J*, results in:

```
load  r2, %*J
```

which makes no sense. However, the errors will not be detected until after the macro expander has run and the assembler attempts to assemble the program. More important, because macro expansion produces a source program, error messages that refer to line numbers will reference lines in the expanded program, not in the original source code that a programmer submits.

The point is:

> *A macro expansion facility preprocesses an assembly language source program to produce another source program in which each macro invocation is replaced by the text of the macro. Because a macro processor uses textual substitution, incorrect arguments are not detected by the macro processor; errors are only detected by the assembler after the macro processor completes.*

9.23 Summary

Assembly languages are low-level languages that incorporate characteristics of a processor, such as the instruction set, operand addressing modes, and registers. Many assembly languages exist, one or more for each type of processor. Despite differences, most assembly languages follow the same basic structure.

Each statement in an assembly language corresponds to a single instruction on the underlying hardware; the statement consists of an optional label, opcode, and operands. The assembly language for a processor defines a syntactic form for each type of operand the processor accepts.

Although assembly languages differ, most follow the same basic paradigm. Therefore, we can specify typical assembly language sequences for conditional execution, conditional execution with alternate paths, definite iteration, and indefinite iteration. Most processors include instructions used to invoke a subroutine or function and return to the caller. The details of argument passing, return address storage, and return of values to a caller differ. Some processors place arguments in memory, and others pass arguments in registers.

An assembler is a piece of software that translates an assembly language source program into binary code that the processor can execute. Conceptually, an assembler makes two passes over the source program: one to assign addresses and one to generate code. Many assemblers include a macro facility to help programmers avoid tedious coding repetition; the macro expander generates a source program which is then assembled. Because it uses textual substitution, macro expansion can result in illegal code that is only detected and reported by the two main passes of the assembler.

EXERCISES

9.1 State and explain the characteristics of a low-level language.

9.2 Where might a programmer expect to find comments in an assembly language program?

9.3 If a program contains an *if-then-else* statement, how many branch instructions will be performed if the condition is true? If the condition is false?

9.4 What is the assembly language used to implement a *repeat* statement?

9.5 Name three argument passing mechanisms that have been used in commercial processors.

9.6 Write an assembly language function that takes two integer arguments, adds them, and returns the result. Test your function by calling it from C.

9.7 Write an assembly language program that declares three integer variables, assigns them 1, 2, and 3, and then calls printf to format and print the values.

9.8 Programmers sometimes mistakenly say *assembler language*. What have they confused, and what term should they use?

9.9 In Figure 9.12, if an instruction is inserted following label4 that jumps to label2, to what address will it jump? Will the address change if the new instruction is inserted before label1?

9.10 Look at Figure 9.8 to see the example Fibonacci program written in C. Can the program be redesigned to be faster? How?

9.11 Optimize the Fibonacci programs in Figures 9.10 and 9.11 by choosing to keep values in registers rather than writing them to memory. Explain your choices.

9.12 Compare the x86 and ARM versions of the Fibonacci program in Figures 9.10 and 9.11. Which version do you expect to require more code? Why?

9.13 Use the -S option on gcc to generate assembly code for a C program. For example, try the program in Figure 9.8. Explain all the extra code that is generated.

9.14 What is the chief disadvantage of using an assembly language macro instead of a function?

Part III

Memories

Program And Data
Storage Technologies

Chapter Contents

10

Memory And Storage

10.1 Introduction

Previous chapters examine one of the major components used in computer systems: processors. The chapters review processor architectures, including instruction sets, operands, and the structure of complex CPUs.

This chapter introduces the second major component used in computer systems: memories. Successive chapters explore the basic forms of memory: physical memory, virtual memory, and caches. Later chapters examine I/O, and show how I/O devices use memory.

10.2 Definition

When programmers think of memory, they usually focus on the main memory found in a conventional computer. From the programmer's point of view, the main memory holds running programs as well as the data the programs use. In a broader sense, computer systems use a *storage hierarchy* that includes general-purpose registers, main memory, and secondary storage (e.g., a disk or flash storage). Throughout this text, we will use the term *memory* to refer specifically to main memory, and generally use the term *storage* for the broader hierarchy and the abstractions programmers use with the hierarchy.

An architect views a *memory* as a solid-state digital device that provides storage for data values. The next sections clarify the concept by examining the variety of possibilities.

10.3 The Key Aspects Of Memory

When an architect begins to design a memory system, two key choices arise:

- Technology
- Organization

Technology refers to the properties of the underlying hardware mechanisms used to construct the memory system. We will learn that many technologies are available, and see examples of their properties. We will also learn how basic technologies operate, and understand when each technology is appropriate.

Organization refers to the way the underlying technology is used to form a working system. We will see that there are many choices about how to combine a one-bit memory cell into multibit memory cells, and we will learn that there are multiple ways to map a memory address into the underlying units.

In essence, *memory technology* refers to the lowest-level hardware pieces (i.e., individual chips), and *memory organization* refers to how those pieces are combined to create meaningful storage systems. We will see that both aspects contribute to the cost and performance of a memory system.

10.4 Characteristics Of Memory Technologies

Memory technology is not easy to define because a wide range of technologies has been invented. To help clarify the broad purpose and intent of a given type of memory, engineers use several characteristics:

- Volatile or nonvolatile
- Random or sequential access
- Read-write or read-only
- Primary or secondary

10.4.1 Memory Volatility

A memory is classified as *volatile* if the contents of the memory disappear when power is removed. The main memory used in most computers (RAM) is volatile — when the computer is shut down, the running applications and data stored in the main memory vanish.

In contrast, memory is known as *nonvolatile* if the contents survive even after power is removed†. For example, the flash memory used in digital cameras and *Solid State Disks* (*SSDs*) is nonvolatile — the data stored in the camera or on the disk

†An emerging technology known as *NonVolatile RAM* (*NVRAM*) operates like a traditional main memory, but retains values when the power is removed.

remains intact when the power is turned off. In fact, data remains even if the storage device is removed from the camera or computer.

10.4.2 Memory Access Paradigm

The most common forms of memory are classified as *random access*, which means that any value in the memory can be accessed in a fixed amount of time independent of its location or of the sequence of locations accessed. The term *Random Access Memory* (*RAM*) is so common that consumers look for RAM when they purchase a computer. The alternative to random access is *sequential access* in which the time to access a given value depends on the location of that value in the memory and the location of the previously accessed value (typically, accessing the next sequential location in memory is much faster than accessing any other location). For example, one type of sequential access memory consists of a *FIFO*† queue implemented in hardware.

10.4.3 Permanence Of Values

Memory is characterized by whether values can be extracted, updated, or both. The primary form of memory used in a conventional computer system permits an arbitrary value in memory to be accessed (read) or updated (written) at any time. Other forms of memory provide more permanence. For example, some memory is characterized as *Read Only Memory* (*ROM*) because the memory contains data values that can be accessed, but cannot be changed.

A form of ROM, *Programmable Read Only Memory* (*PROM*), is designed to allow data values to be stored in the memory and then accessed many times. In the extreme case, a PROM can only be written once — high voltage is used to alter the chip permanently.

Intermediate forms of permanence also exist. For example, the *flash memory* commonly used in smart phones and solid state disks represents a compromise between permanent ROM and technologies with little permanence — although it retains data when power is removed, the items in flash memory do not last forever. An exercise asks the reader to research flash technologies to discover how long data will last if a flash device sits idle.

10.4.4 Primary And Secondary Memory

The terms *primary memory* and *secondary memory* are qualitative. Originally, the terms were used to distinguish between the fast, volatile, internal main memory of a computer and the slower, nonvolatile storage provided by an external electromechanical device such as a hard disk. However, many computer systems now use solid-state memory technologies for both primary and secondary storage. In particular, *Solid State Disks* (*SSDs*) are used for secondary storage.

†FIFO abbreviates *First-In-First-Out*.

10.5 The Important Concept Of A Memory Hierarchy

The notions of primary and secondary memory arise as part of the *memory hierarchy* in a computer system. To understand the hierarchy, we must consider both performance and cost: memory that has the highest performance characteristics is also the most expensive. Thus, an architect must choose memory that satisfies cost constraints.

Research on memory use has led to an interesting principle: for a given cost, optimal performance is not achieved by using one type of memory throughout a computer. Instead, a set of technologies should be arranged in a conceptual *memory hierarchy*. The hierarchy has a small amount of the highest performance memory, a slightly larger amount of slightly slower memory, and so on. For example, an architect selects a small number of general-purpose registers, a larger amount of primary memory, and an even larger amount of secondary memory. We can summarize the principle:

> *To optimize memory performance for a given cost, a set of technologies are arranged in a hierarchy that contains a relatively small amount of fast memory and larger amounts of less expensive, but slower memory.*

Chapter 12 further examines the concept of a memory hierarchy. The chapter presents the scientific principle behind a hierarchical structure, and explains how a memory mechanism known as a *cache* uses the principle to achieve higher performance without high cost.

10.6 Instruction And Data Store

Recall that some of the earliest computer systems used a *Harvard Architecture* with separate memories for programs and data. Later, most architects adopted a *Von Neumann Architecture* in which a single memory holds both programs and data.

Interestingly, the advent of specialized solid state memory technologies has reintroduced the separation of program and data memory — special-purpose systems sometimes use separate memories. Memory used to hold a program is known as *instruction store*, and memory used to hold data is known as *data store*.

One of the motivations for a separate instruction store comes from the notion of memory hierarchy: on many systems, overall performance can be increased by increasing the speed of the instruction store. To understand why, observe that high-speed instructions are designed to operate on values in general-purpose registers rather than values in memory. Thus, to optimize speed, data is kept in registers whenever possible. However, an instruction must be accessed on each iteration of the fetch-execute cycle. Thus, the instruction store experiences more activity than the data store. More important, although data accesses tend to follow a random pattern of accessing a variable and then another variable, a processor typically accesses an instruction store sequentially.

That is, instructions are placed one after another in memory, and the processor moves from one to the next unless a branch occurs. Separating the two memories allows a designer to optimize the instruction store for sequential access.

We can summarize:

> *Although most modern computer systems place programs and data in a single memory, it is possible to separate the instruction store from the data store. Doing so allows an architect to select memory performance appropriate for each activity.*

10.7 The Fetch-Store Paradigm

As we will see, all memory technologies use a single paradigm that is known as *fetch-store*. For now, it is only important to understand that there are two basic operations associated with the paradigm: fetching a value from the memory or storing a value into the memory. The fetch operation is sometimes called *read* or *load*. If we think of memory as an array of locations, we can view reading a value from memory as an array index operation in which a memory address is used:

$$\text{value} \leftarrow \text{memory}[\,\text{address}\,]$$

The analogy also holds for store operations, which are sometimes called *write* operations. That is, we can view storing a value into memory as storing a value into an array:

$$\text{memory}[\,\text{address}\,] \leftarrow \text{value}$$

The next chapter explains the idea in more detail. Later chapters on I/O explain how the fetch-store paradigm is used for input and output devices, and how the underlying memory access relates to I/O.

10.8 Summary

The two key aspects of memory are the underlying technology and the organization. A variety of technologies exist; they can be characterized as volatile or nonvolatile, random or sequential access, permanent or nonpermanent (read-only or read-write), and primary or secondary.

To achieve maximal performance at a given cost, an architect organizes memory into a conceptual hierarchy. The hierarchy contains a small amount of high performance memory and a large amount of lower performance memory.

Memory systems use a fetch-store paradigm. The memory hardware only supports two operations: one that retrieves a value from memory and another that stores a value into memory.

EXERCISES

10.1 Define *storage hierarchy* and give an example.

10.2 What are the two key choices an architect makes when designing a memory system?

10.3 Read about RAM and SSD technologies for a typical computer. What is the approximate financial cost per byte of each type of memory?

10.4 Extend the previous exercise by finding the speed (access times) of the memories and comparing the financial cost of the performance.

10.5 Which type of memory is more secure (i.e., less susceptible to someone trying to change the contents), flash memory or ROM?

10.6 If data is stored in a *volatile* memory, what happens to the data when power is removed?

10.7 Suppose NVRAM were to replace DRAM. What characteristic of memory technologies becomes less important (or even disappears)?

10.8 Compare the performance of an NVRAM to traditional RAM. How much slower is NVRAM?

10.9 Research the flash technology used in typical USB flash drives which are also called *jump drives* or *thumb drives*. If a flash drive is left unused, how long will the data persist? Are you surprised by the answer?

10.10 Registers are much faster than main memory, which means a program could run much faster if all the data were kept in registers instead of main memory. Why do designers create processors with so few registers?

10.11 If a computer follows a Harvard Architecture, do you expect to find two identical memories, one for instructions and one for data? Why or why not?

10.12 Do the terms *fetch-execute* and *fetch-store* refer to the same concept? Explain.

Chapter Contents

11

Physical Memory And Physical Addressing

11.1 Introduction

The previous chapter introduces the topic of memory, lists characteristics of memory systems, and explains the concept of a memory hierarchy. This chapter explains how a basic memory system operates. The chapter considers both the underlying technologies used to construct a typical computer memory and the organization of the memory into bytes and words. The next chapter expands the discussion to consider virtual memory.

11.2 Characteristics Of Computer Memory

Engineers use the term *Random Access Memory* (*RAM*) to denote the type of memory used as the primary memory system in most computers. As the name implies, RAM is optimized for random (as opposed to sequential) access. In addition, RAM offers read-write capability that makes access and update equally inexpensive. Finally, we will see that most RAM is volatile — values do not persist after the computer is powered down.

11.3 Static And Dynamic RAM Technologies

The technologies used to implement Random Access Memory can be divided into two broad categories. *Static RAM* (*SRAM*†) is the easiest type for programmers to understand because it is a straightforward extension of digital logic. Conceptually, SRAM stores each data bit in a latch, a miniature digital circuit composed of multiple transistors similar to the latch discussed in Chapter 2. Although the internal implementation is beyond the scope of this text, Figure 11.1 illustrates the three external connections used for a single-bit of RAM.

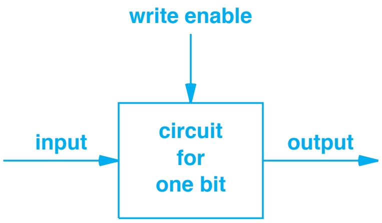


Figure 11.1 Illustration of a miniature static RAM circuit that stores one data bit. The circuit contains multiple transistors.

In the figure, the circuit has two inputs and one output. When the *write enable* input is on (i.e., logical 1), the circuit sets the output value equal to the input (0 or 1); when the *write enable* input is off (i.e., logical 0), the circuit ignores the input and keeps the output at the last setting. Thus, to store a value, the hardware places the value on the input, turns on the write enable line, and then turns the enable line off again.

Although it performs at high speed, SRAM has a significant disadvantage: high power consumption (which generates heat). The miniature SRAM circuit contains multiple transistors that operate continuously. Each transistor consumes a small amount of power, and therefore, generates a small amount of heat.

The alternative to static RAM, which is known as *Dynamic RAM* (*DRAM*‡), consumes less power. The internal working of dynamic RAM is surprising and can be confusing. At the lowest level, to store information, DRAM uses a circuit that acts like a *capacitor*, a device that stores electrical charge. When a value is written to DRAM, the hardware charges or discharges the capacitor to store a 1 or 0. Later, when a value is read from DRAM, the hardware examines the charge on the capacitor and generates the appropriate digital value.

The conceptual difficulty surrounding DRAM arises from the way a capacitor works: because physical systems are imperfect, a capacitor gradually loses its charge. In essence, a DRAM chip is an imperfect memory device — as time passes, the charge dissipates and a one becomes zero. More important, DRAM loses its charge in a short time (e.g., in some cases, under a second).

†SRAM is pronounced "ess-ram."
‡DRAM is pronounced "dee-ram."

How can DRAM be used as a computer memory if values can quickly become zero? The answer lies in a simple technique: devise a way to read a bit from memory before the charge has time to dissipate, and then write the same value back again. Writing a value causes the capacitor to start again with the appropriate charge. So, reading and then writing a bit will reset the capacitor without changing the value of the bit.

In practice, computers that use DRAM contain an extra hardware circuit, known as a *refresh circuit*, that performs the task of reading and then writing a bit. Figure 11.2 illustrates the concept.

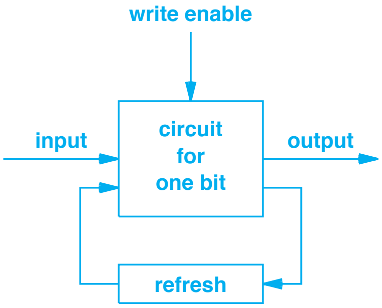


Figure 11.2 Illustration of a bit in dynamic RAM. An external refresh circuit must periodically read the data value and write it back again, or the charge will dissipate and the value will be lost.

The refresh circuit is more complex than the figure implies. To keep the refresh circuit small, architects do not build one refresh circuit for each bit. Instead, a single, small refresh mechanism is designed that can cycle through the entire memory. As it reaches a bit, the refresh circuit reads the bit, writes the value back, and then moves on. Complexity also arises because a refresh circuit must coordinate with normal memory operations. First, the refresh circuit must not interfere or delay normal memory operations. Second, the refresh circuit must ensure that a normal *write* operation does not change the bit between the time the refresh circuit reads the bit and the time the refresh circuit writes the same value back. Despite the need for a refresh circuit, the cost and power consumption advantages of DRAM are so beneficial that most computer memory is composed of DRAM rather than SRAM.

11.4 The Two Primary Measures Of Memory Technology

Architects use several quantitative measures to assess memory technology; two stand out:

- Density
- Latency and cycle times

11.5 Density

In a strict sense, the term *density* refers to the number of memory cells per square area of silicon. In practice, however, density often refers to the number of bits that can be represented on a standard size chip or plug-in module. For example, a *Dual In-line Memory Module* (*DIMM*) might contain a set of chips that offer 128 million locations of 64 bits per location, which equals 8.192 billion bits or one Gigabyte. Informally, it is known as a *1 gig module*. Higher density is usually desirable because it means more memory can be held in the same physical space. However, higher density has the disadvantages of increased power utilization and increased heat generation.

The density of memory chips is related to the size of transistors in the underlying silicon technology, which has followed Moore's Law. Thus, memory density tends to double approximately every eighteen months.

11.6 Separation Of Read And Write Performance

A second measure of a memory technology focuses on speed: how fast can the memory respond to requests? It may seem that speed should be easy to measure, but it is not. For example, as the previous chapter discusses, some memory technologies take much longer to write values than to read them. To choose an appropriate memory technology, an architect needs to understand both the cost of access and the cost of update. Thus, a principle arises:

> *In many memory technologies, the time required to fetch information from memory differs from the time required to store information in memory, and the difference can be dramatic. Therefore, any measure of memory performance must give two values: the performance of* read *operations and the performance of* write *operations.*

11.7 Latency And Memory Controllers

In addition to separating *read* and *write* operations, we must decide exactly what to measure. It may seem that the most important measure is *latency* (i.e., the time that elapses between the start of an operation and the completion of the operation). However, latency is a simplistic measure that does not provide complete information.

To see why latency does not suffice as a measure of memory performance, we need to understand how the hardware works. In addition to the memory chips themselves, additional hardware known as a *memory controller*† provides an interface between the processor and memory. Figure 11.3 illustrates the organization.

†We will learn more about the memory controller later in the chapter.

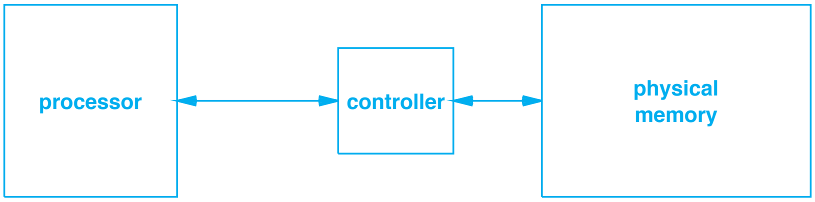


Figure 11.3 Illustration of the hardware used for memory access. A controller sits between the processor and physical memory.

To access memory, a device (typically a processor) presents a *read* or *write* request to the controller. The controller translates the request into signals appropriate for the underlying memory, and passes the signals to the memory chips. To minimize latency, the controller returns an answer as quickly as possible (i.e., as soon as the memory responds). However, after it responds to a device, a controller may need additional clock cycle(s) to reset hardware circuits and prepare for the next operation.

A second principle of memory performance arises:

> *Because a memory system may need extra time between operations, latency is an insufficient measure of performance; a performance measure needs to measure the time required for successive operations.*

That is, to assess the performance of a memory system, we need to measure how fast the system can perform a sequence of operations. Engineers use the term *memory cycle time* to capture the idea. Specifically, two separate measures are used: the *read cycle time* (abbreviated *tRC*) and the *write cycle time* (abbreviated *tWC*).

We can summarize:

> *The* read cycle time *and* write cycle time *are used as measures of memory system performance because they assess how quickly the memory system can handle successive requests.*

11.8 Synchronous And Multiple Data Rate Technologies

Like most other digital circuits in a computer, a memory system uses a *clock* that controls exactly when a *read* or *write* operation begins. As Figure 11.3 indicates, a memory system must also coordinate with a processor. The controller may also coordinate with I/O devices. What happens if the processor's clock differs from the clock used for memory? The system still works because the controller can hold a request from the processor or a response from the memory until the other side is ready.

Unfortunately, the difference in clock rates can impact performance — although the delay is small, if delay occurs on every memory reference, the accumulated effect can be large. To eliminate the delay, some memory systems use a *synchronous* clock system. That is, the clock pulses used with the memory system are aligned with the clock pulses used to run the processor. As a result, a processor does not need to wait for memory references to complete. Synchronization can be used with DRAM or SRAM; the two technologies are named:

SDRAM– Synchronous Dynamic Random Access Memory

SSRAM– Synchronous Static Random Access Memory

In practice, synchronization has been effective; most computers now use synchronous DRAM as the primary memory technology.

In many computer systems, memory is the bottleneck — increasing memory performance improves overall performance. As a result, engineers have concentrated on finding memory technologies with lower cycle times. One approach uses a technique that runs the memory system at a multiple of the normal clock rate (e.g., double or quadruple). Because the clock runs faster, the memory can deliver data faster. The technologies are sometimes called *fast data rate* memories, typically *double data rate* or *quadruple data rate*. Fast data rate memories have been successful, and are now standard on most computer systems, including consumer systems such as laptops.

Although we have covered the highlights, our discussion of RAM memory technology does not begin to illustrate the range of choices available to an architect or the detailed differences among them. For example, Figure 11.4 lists a few commercially available RAM technologies:

Technology	Description
DDR-DRAM	**Double Data Rate Dynamic RAM**
DDR-SDRAM	**Double Data Rate Synchronous Dynamic RAM**
FCRAM	**Fast Cycle RAM**
FPM-DRAM	**Fast Page Mode Dynamic RAM**
QDR-DRAM	**Quad Data Rate Dynamic RAM**
QDR-SRAM	**Quad Data Rate Static RAM**
SDRAM	**Synchronous Dynamic RAM**
SSRAM	**Synchronous Static RAM**
ZBT-SRAM	**Zero Bus Turnaround Static RAM**
RDRAM	**Rambus Dynamic RAM**
RLDRAM	**Reduced Latency Dynamic RAM**

Figure 11.4 Examples of commercially available RAM technologies. Many other technologies exist.

11.9 Memory Organization

Recall that there are two key aspects of memory: the underlying technology and the memory organization. As we have seen, an architect can choose from a variety of memory technologies; we will now consider the second aspect. Memory organization refers to both the internal structure of the hardware and the external addressing structure that the memory presents to a processor. We will see that the two are related.

11.10 Memory Access And Memory Bus

To understand how memory is organized, we need to examine the access paradigm. Recall from Figure 11.3 that a *memory controller* provides the interface between a physical memory and a processor that uses the memory†. Several questions arise. What is the structure of the connection between a processor and memory? What values pass across the connection? How does the processor view the memory system?

To achieve high performance, memory systems use parallelism: the connection between the processor and controller consists of many wires that are used simultaneously. Each wire can transfer one data bit at any time. Figure 11.5 illustrates the concept.

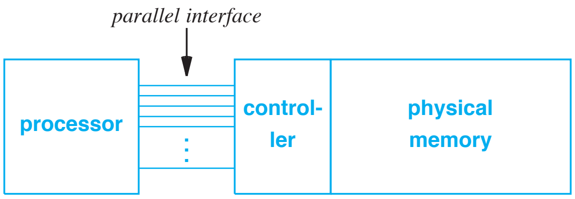


Figure 11.5 The parallel connection between a processor and memory. A connection that contains N wires allows N bits of data to be transferred simultaneously.

The technical name for the hardware connection between a processor and memory is *bus* (more specifically, *memory bus*). We will learn about buses in the chapters on I/O; for now, it is sufficient to understand that a bus provides parallel connections.

11.11 Words, Physical Addresses, And Memory Transfers

The parallel connections of a memory bus are pertinent to programmers as well as computer architects. From an architectural standpoint, using parallel connections can improve performance. From a programming point of view, the parallel connections define a *memory transfer size* (i.e., the amount of data that can be read or written to

†In later chapters, we will learn that I/O devices also access memory through the memory controller; for now, we will use a processor in the examples.

memory in a single operation). We will see that transfer size is a crucial aspect of memory organization.

To permit parallel access, the bits that comprise a physical memory are divided into blocks of N bits per block, where N is the memory transfer size. A block of N bits is sometimes called a *word*, and the transfer size is called the *word size* or the *width* of a word. We can think of memory being organized into an array. Each entry in the array is assigned a unique index known as a *physical memory address*; the approach is known as *word addressing*. Figure 11.6 illustrates the idea and shows that the physical memory address is exactly like an array index.

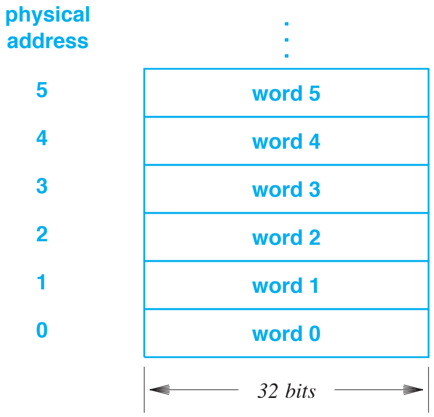


Figure 11.6 Physical memory addressing on a computer where a word is thirty-two bits. We think of the memory as an array of words.

11.12 Physical Memory Operations

The controller for physical memory supports two operations: *read* and *write*. In the case of a *read* operation, the processor specifies an address; in the case of a *write* operation, the processor specifies an address as well as data to be written. The fundamental idea is that the controller always accepts or delivers an entire word; physical memory hardware does not provide a way to read or write less than a complete word (i.e., the hardware does not allow the processor to access or alter part of a word).

The point is:

> *Physical memory is organized into words, where a word is equal to the memory transfer size. Each* read *or* write *operation applies to an entire word.*

11.13 Memory Word Size And Data Types

Recall that the parallel connection between a processor and a memory is designed for high performance. In theory, performance can be increased by adding more parallel wires. For example, an interface that has 128 wires can transfer data at twice the rate of an interface that has 64 wires. The question arises: how many wires should an architect choose? That is, what word size is optimal? The question is complicated by several factors. First, because memory is used to store data, the word size should accommodate common data values (e.g., the word should be large enough to hold an integer). Second, because memory is used to store programs, the word size should accommodate frequently used instructions. Third, because the connection of a processor to a memory requires pins on the processor, adding wires to the interface increases the pin requirements (the number of pins can be a limiting factor in the design of a CPU chip). Thus, the word size is chosen as a compromise between performance and various other considerations. A word size of thirty-two bits is popular, especially for low-power systems; many high-performance systems use a sixty-four-bit word size.

In most cases, an architect designs all parts of a computer system to work together. Thus, if an architect chooses a memory word size equal to thirty-two bits, the architect will make a standard integer and a single-precision floating point value each occupy thirty-two bits. As a result, a computer system is often characterized by stating the word size (e.g., a thirty-two-bit processor).

11.14 Byte Addressing And Mapping Bytes To Words

Programmers who use a conventional computer may be surprised to learn that physical memory is organized into words because most programmers are familiar with an alternate form of addressing known as *byte addressing*. Byte addressing is especially convenient for programming because it gives a programmer an easy way to access small data items such as characters.

Conceptually, when byte addressing is used, memory must be organized as an array of bytes rather than an array of words. The choice of byte addressing has two important consequences. First, because each byte of memory is assigned an address, byte addressing requires more addresses than word addressing. Second, because byte addressing allows a program to read or write a single byte, the memory controller must support byte transfer.

A larger word size results in higher performance because many bits can be transferred at the same time. Unfortunately, if the word size is equal to an eight-bit byte, only eight bits can be transferred at one time. That is, a memory system built for byte addressing will have lower performance than a memory system built for a larger word size. Interestingly, even when byte addressing is used, many transfers between a processor and memory involve multiple bytes. For example, an instruction occupies multiple bytes, as does an integer, a floating point value, and a pointer.

Can we devise a memory system that combines the higher speed of word addressing with the programming convenience of byte addressing? The answer is yes. To do so, we need an intelligent memory controller that can translate between the two addressing schemes. The controller accepts requests from the processor that specify a byte address and size. The controller uses word addressing to access the appropriate word(s) in the underlying memory and extract the specified bytes. Figure 11.7 shows an example of the mapping used between byte addressing and word addressing for a word size of thirty-two bits.

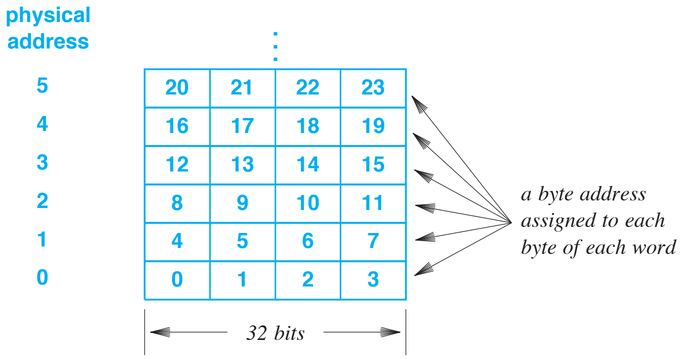


Figure 11.7 Example of a byte address assigned to each byte of memory even though the underlying hardware uses word addressing and a thirty-two-bit word size.

To implement the mapping shown in the figure, a controller must convert byte addresses issued by the processor to word addresses used by the memory system. For example, if the processor requests a *read* operation for byte address 17, the controller must issue a *read* request for word 4, and then extract the second byte from the word.

Because the memory can only transfer an entire word at a time, a byte *write* operation is expensive. For example, if a processor writes byte 11, the controller must read word 2 from memory, replace the rightmost byte, and then write the entire word back to memory.

Mathematically, the translation of addresses is straightforward. To translate a byte address, B, to the corresponding word address, W, the controller divides B by N, the number of bytes per word, and ignores the remainder. Similarly, to compute a byte offset, O, within a word, the controller computes the remainder of B divided by N. That is, the word address is given by:

$$W = \left\lfloor \frac{B}{N} \right\rfloor$$

and the offset is given by:

$$O = B \bmod N$$

As an example, consider the values in Figure 11.7, where $N = 4$. A byte address of 11 translates to a word address of 2 and an offset of 3, which means that byte 11 is found in word 2 at byte offset 3†.

11.15 Using Powers Of Two

Performing a division or computing a remainder is time consuming and requires extra hardware (e.g., an Arithmetic Logic Unit). To avoid computation, architects organize memory using powers of two. Doing so means that hardware can perform the two computations above simply by extracting bits. In Figure 11.7, for example, $N = 2^2$, which means that the offset can be computed by extracting the two low-order bits, and the word address can be computed by extracting everything except the two low-order bits. Figure 11.8 illustrates the idea:

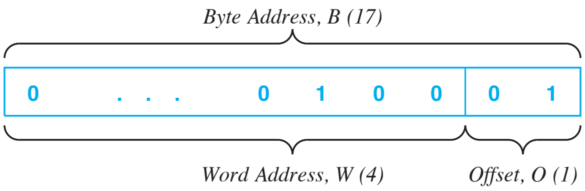


Figure 11.8 An example of a mapping from byte address 17 to word address 4 and offset 1. Using a power of two for the number of bytes per word avoids arithmetic calculations.

We can summarize:

> *To avoid arithmetic calculations, such as division or remainder, physical memory is organized such that the number of bytes per word is a power of two, which means the translation from a byte address to a word address and offset can be performed by extracting bits.*

11.16 Byte Alignment And Programming

Knowing how the underlying hardware works helps explain a concept that programmers encounter: *byte alignment*. We say that an integer value is *aligned* if the bytes of the integer correspond to a word in the underlying physical memory. In Figure 11.7, for example, an integer composed of bytes 12, 13, 14, and 15 is aligned, but an integer composed of bytes 6, 7, 8, and 9 is not.

†The offset is measured from zero.

On some architectures, byte alignment is required — the processor raises an error if a program attempts an integer access using an unaligned address. On other processors, arbitrary alignment is allowed, but unaligned accesses result in lower performance than aligned accesses. We can now understand why an unaligned address requires more accesses of physical memory: the memory controller must convert each processor request into operations on the underlying memory. If an integer spans two words, the controller must perform two *read* operations to obtain the requested bytes. Thus, even if the processor permits unaligned access, programmers are strongly encouraged to align data values.

We can summarize:

> *The organization of physical memory affects programming: even if a processor allows unaligned memory access, aligning data on boundaries that correspond to the physical word size can improve program performance.*

11.17 Memory Size And Address Space

How large can a memory be? It may seem that memory size is only an economic issue — more memory simply costs more money. However, size turns out be an essential aspect of memory architecture because overall memory size is inherently linked to other design choices. In particular, the addressing scheme determines a maximum memory size.

Recall that data paths in a processor consist of parallel hardware. When the processor is designed, the designer must choose a size for each data path, register, and other hardware units. The choice places a fixed bound on the size of an address that can be generated or passed from one unit to another. Typically, the address size is the same as the integer size. For example, a processor that uses thirty-two-bit integers uses thirty-two-bit addresses and a processor that uses sixty-four-bit integers uses sixty-four-bit addresses. As Chapter 3 points out, a string of k bits can represent 2^k values. Thus, a thirty-two-bit value can represent:

$$2^{32} = 4{,}294{,}967{,}296$$

unique addresses (i.e., addresses 0 through 4,294,967,295). We use the term *address space* to denote the set of possible addresses.

The tradeoff between byte addressing and word addressing is now clear: given a fixed address size, the amount of memory that can be addressed depends on whether the processor uses byte addressing or word addressing. Furthermore, if word addressing is used, the amount of memory that can be addressed depends on the word size. For example, on a computer that uses word addressing with four bytes per word, a thirty-two-bit value can hold enough addresses for 17,179,869,184 bytes (i.e., four times as much as when byte addressing is used).

11.18 Programming With Word Addressing

Many processors use byte addressing because byte addressing provides the most convenient interface for programmers. However, byte addressing does not maximize memory size. Therefore, specialized systems, such as processors designed for numeric processing, use word addressing to provide access to the maximum amount of memory for a given address size.

On a processor that uses word addressing, software must handle the details of byte manipulation. In essence, software performs the same function as a memory controller in a byte-addressed architecture. For example, to extract a single byte, software must read the appropriate word from memory, and then extract the byte. Similarly, to write a byte, software must read the word containing the byte, update the correct byte, and write the modified word back to memory. To optimize software performance, logical shifts and bit masking are used to manipulate an address rather than division or remainder computation. Similarly, shifts and logical operations are used to extract bytes from a word. For example, to extract the leftmost byte from a thirty-two-bit word, *w*, a programmer can code a C statement:

```
( w  >>  24 )  &  0xff
```

The code performs a *logical and* with constant 0xff to ensure that only the low-order eight bits are kept after the shift is performed. To understand why the *logical and* is needed, recall from Chapter 3 that a right shift propagates the sign bit. Thus, if *w* contains 0xa1b2c3d2, the expression *w*>>24 will produce 0xffffffa1. After the *logical and*, the result is 0xa1.

11.19 Memory Size And Powers Of Two

We said that a physical memory architecture can be characterized as follows:

> *Physical memory is organized into a set of* M *words that each contain* N *bytes; to make controller hardware efficient,* M *and* N *are each chosen to be powers of two.*

The use of powers of two for word and address space size has an interesting consequence: the maximum amount of memory is always a power of two rather than a power of ten. As a result, memory size is measured in powers of two. For example, a *Kilobyte* (*Kbyte*) is defined to consist of 2^{10} bytes, a *Megabyte* (*MB*) is defined to consist of 2^{20} bytes, and a *Gigabyte* (*GB*) is defined to consist of 2^{30} bytes. The terminology is confusing because it is an exception. In computer networking, for example, a measure of *Megabits per second* refers to base ten. Thus, one must be careful when mixing memory size with other measures (e.g., although there are eight bits per byte, one Kilobyte of data in memory is not eight times larger than one Kilobit of data sent across a network). We can summarize:

> *When used to refer to memory, the prefixes kilo, mega, and giga are defined as powers of 2; when used with other aspects of computing, such as computer networking, the prefixes are defined as powers of 10.*

11.20 Pointers And Data Structures

Memory addresses are important because they form the basis for commonly used data abstractions, such as linked lists, queues, and trees. Consequently, programming languages often provide facilities that allow a programmer to declare a *pointer* variable that holds a memory address, assign a value to a pointer, or dereference a pointer to obtain an item. In the C programming language, for example, the declaration:

```
char    *cptr;
```

declares variable *cptr* to be a pointer to a character (i.e., to a byte in memory). The compiler allocates storage for variable *cptr* equal to the size of a memory address, and allows the variable to be assigned the address of an arbitrary byte in memory.

The *autoincrement* statement:

```
cptr++;
```

increases the value of *cptr* by one (i.e., moves to the next byte in memory).

Interestingly, the C programming language has a heritage of both byte and word addressing. When performing arithmetic on pointers, C accommodates the size of the underlying item. As an example, the declaration:

```
int    *iptr;
```

declares variable *iptr* to be a pointer to an integer (i.e., a pointer to a word). The compiler allocates storage for variable *iptr* equal to the size of a memory address (i.e., the same size as allocated for *cptr* above). However, if the program is compiled and run on a processor that defines an integer to be four bytes, the *autoincrement* statement:

```
iptr++;
```

increases the value of *iptr* by four. That is, if *iptr* is declared to be the byte address of the beginning of a word in memory, autoincrement moves to the byte address of the next word in memory.

In fact, all the examples above assume a byte-addressable computer. The compiler generates code to increment a character pointer by one and an integer pointer by four. Although C has facilities that allow a pointer to move from one word to the next, the language is intended for use with a byte-addressable memory.

11.21 A Memory Dump

A trivial example will help us understand the relationship between pointers and memory addresses. Consider a linked list as Figure 11.9 illustrates.

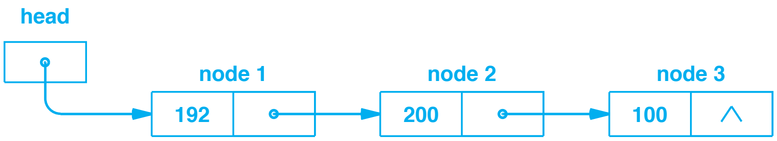


Figure 11.9 Example of a linked list. Each pointer in the list corresponds to a memory address.

To create such a list, a programmer must write a declaration that specifies the contents of a node, and then must allocate memory to hold the list. In our trivial example, each node in the list will contain two items: an integer count and a pointer to the next node on the list. In C, a *struct* declaration is used to define the contents of a node:

```
struct  node  {
      int count;
      struct node *next;
}
```

Similarly, a variable named *head* that serves as the head of the list is defined as:

```
struct  node  *head;
```

To understand how the list appears in memory, consider a *memory dump* as Figure 11.10 illustrates†.

```
   Address                       Contents Of Memory
0001bde0    |  00000000   0001bdf8   deadbeef   4420436f
0001bdf0    |  6d657200   0001be18   000000c0   0001be14
0001be00    |  00000064   00000000   00000000   00000002
0001be10    |  00000000   000000c8   0001be00   00000006
```

Figure 11.10 Illustration of a small portion of a memory dump that shows the contents of memory. The address column gives the memory address of the leftmost byte on the line, and all values are shown in hexadecimal.

†As the figure shows, a programmer can initialize memory to a hexadecimal value that makes it easy to identify items in a memory dump. In the example, a programmer has used the value *deadbeef*.

The example in the figure is taken from a processor that uses byte addressing. Each line of the figure corresponds to sixteen contiguous bytes of memory that are divided into four groups of four bytes. Each group contains eight hexadecimal digits to represent the values of four bytes. The address at the beginning of a line specifies the memory address of the first byte on that line. Therefore, the address on each line is sixteen greater than the address on the previous line.

Assume the head of a linked list is found at address 0x0001bde4, which is located on the first line of the dump. The first node of the list starts at address 0x0001bdf8, which is located on the second line of the dump, and contains the integer 192 (hexadecimal constant 000000c0).

The processor uses byte addressing, and bytes of memory are contiguous. In the figure, spacing has been inserted to divide the output into groups of bytes to improve readability. Specifically, the example shows groups of four-byte units, which implies that the underlying word size is four bytes (i.e., thirty-two bits).

11.22 Indirection And Indirect Operands

When we discussed operands and addressing modes in Chapter 7, the topic of indirection arose. Now that we understand memory organization, we can understand how a processor evaluates an indirect operand. As an example, suppose a processor executes an instruction in which an operand specifies an immediate value of 0x1be1f, and specifies indirection. Further suppose that the processor has been designed to use thirty-two-bit values. Because the operand specifies an immediate value, the processor first loads the immediate value (hexadecimal 1be1f). Because the operand specifies indirection, the processor treats the resulting value as an address in memory, and fetches the word from the address. If the values in memory correspond to the values shown in Figure 11.10, the processor will load the value from the rightmost word in the last line of the figure, and the final operand value will be 6.

11.23 Multiple Memories With Separate Controllers

Our discussion of physical memory has assumed a single memory and a single memory controller. In practice, however, some architectures contain multiple physical memories. When multiple memories are used, hardware parallelism can be employed to provide higher memory performance. Instead of a single memory and a single controller, the memory system can have multiple controllers that operate in parallel, as Figure 11.11 illustrates.

In the figure, interface hardware receives requests from the processor. The interface uses the address in the request to decide which memory should be used, and passes the request to the appropriate memory controller†.

Why does it help to have multiple memories, each with their own controller? Remember that after memory is accessed, the hardware must be reset before the next

†Chapter 13 explains that the interface acts as a *Memory Management Unit* (*MMU*), and explains the functionality in more detail.

access can occur. If two memories are available, a programmer can arrange to access one while the other resets, increasing the overall performance. That is, because memory controllers can operate in parallel, using two memory controllers allows more memory accesses to occur per unit time. In a Harvard Architecture, for example, higher performance results because instruction fetch does not interfere with data access and vice versa.

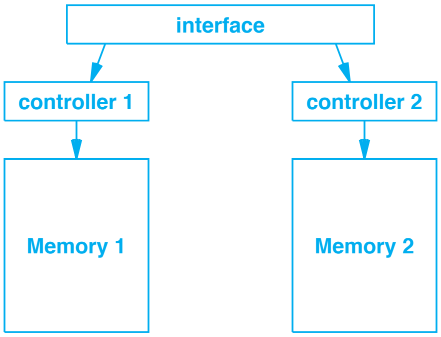


Figure 11.11 Illustration of connections for two memory modules with separate controllers.

11.24 Memory Banks

Multiple physical memories can also be used with a Von Neumann Architecture as a convenient way to form large memory by replicating small memory modules. The idea, known as *memory banks*, uses the interface hardware to map addresses onto two physical memories. For example, suppose two identical memory modules are each designed to have physical addresses 0 through $M-1$. The interface can arrange to treat them as two *banks* that form a contiguous large memory with twice the addresses as Figure 11.12 illustrates.

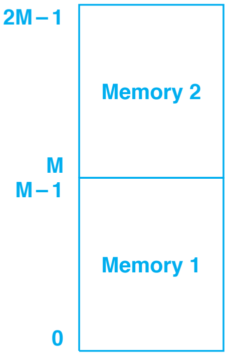


Figure 11.12 The logical arrangement of two identical memory banks to form a single memory that is twice the size.

In Figure 11.12, addresses 0 through *M–1* are assigned to one bank and addresses from *M* to *2M – 1* are assigned to the second bank. In Chapter 13, we will see that mapping an address can be extremely efficient.

Although the banks are arranged to give the illusion of one large memory, the underlying hardware is configured as Figure 11.11 shows. As a result, controllers for the two memory banks can operate in parallel. Thus, if instructions are placed in one bank and data in another, higher performance can result because instruction fetch will not interfere with data access and vice versa.

How do memory banks appear to a programmer? In most architectures, memory banks are transparent — memory hardware automatically finds and exploits parallelism. In embedded systems and other special-purpose architectures, a programmer may be responsible for placing items into separate memory banks to increase performance. For example, a programmer may need to place code at low memory addresses and data items at high memory addresses.

11.25 Interleaving

A related optimization used with physical memory systems is known as *interleaving*. To understand the optimization, observe that many programs access data from sequential memory locations. For example, if a long text string is copied from one place in memory to another or a program searches a list of items, sequential memory locations will be referenced. In a banked memory, sequential locations lie in the same memory bank, which means that successive accesses must wait for the controller to reset.

Interleaving uses the idea of separate controllers, but instead of organizing memories into banks, interleaving places consecutive words of memory in separate physical memory modules. Interleaving achieves high performance during sequential memory accesses because a word can be fetched while the memory for the previous word resets. Interleaving is usually hidden from programmers — a programmer can write code without knowing that the underlying memory system has mapped successive words into separate memory modules. The memory hardware handles all the details automatically.

We use the terminology *N-way interleaving* to describe the number of underlying memory modules (to make the scheme efficient, *N* is chosen to be a power of two). For example, Figure 11.13 illustrates how words of memory are assigned to memory modules in a four-way interleaving scheme.

How can interleaving be achieved efficiently? The answer lies in thinking about the binary representation. In Figure 11.13, for example, words 0, 4, 8, and so on all lie in memory module 0. What do the addresses have in common? When represented in binary, the values all have two low-order bits equal to 00. Similarly, the words assigned to module 1 have low-order bits equal to 01, the words assigned to module 2 have low-order bits equal to 10, and the words assigned to module 3 have low-order bits equal to 11. Thus, when given a memory address, the interface hardware extracts the low-order two bits, and uses them to select a module.

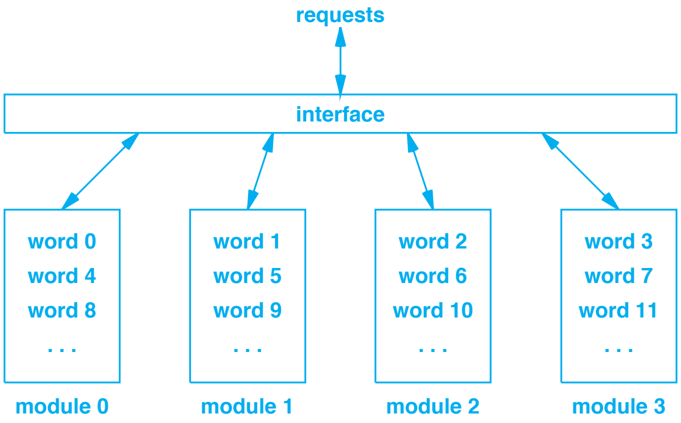


Figure 11.13 Illustration of 4-way interleaving that illustrates how successive words of memory are placed into memory modules to optimize performance.

Interestingly, accessing the correct word within a module is equally efficient. The modules themselves are standard memory modules that provide an array of words addressed 0 through *K–1*, for some value of *K*. The interface ignores the two low-order bits of an address and uses the rest of the bits as an index into the memory module. To see why it works, write 0, 4, 8, ... in binary, and remove the two low-order bits. The result is the sequence 0, 1, 2,... Similarly, removing the two low-order bits from 1, 5, 9... also results in the sequence 0, 1, 2,...

We can summarize:

If the number of modules is a power of two, the hardware for N-way interleaving is extremely efficient because low-order bits of an address are used to select a module and the remaining bits are used as an address within the module.

11.26 Content Addressable Memory

An unusual form of memory exists that blends the two key aspects we discussed: technology and memory organization. The form is known as a *Content Addressable Memory* (*CAM*). As we will see, a CAM does much more than merely store data items — it includes hardware for high-speed searching.

The easiest way to think about a CAM is to view it as memory that has been organized as a two-dimensional array. Each row, which is used to store an item, is called a *slot*. In addition to allowing a processor to place a value in each slot, a CAM allows a processor to specify a *search key* that is exactly as long as one slot. Once a search key has been specified, the hardware can perform a search of the table to determine whether any slot matches the search key. Figure 11.14 illustrates the organization of a CAM.

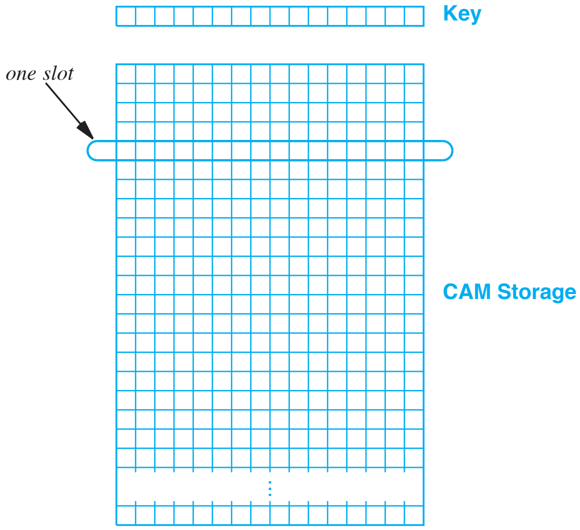


Figure 11.14 Illustration of a *Content Addressable Memory* (*CAM*). CAM provides both a memory technology and a memory organization.

For the most basic form of a CAM, the search mechanism performs an *exact match*. That is, the CAM hardware compares the key against each slot, and reports whether a match was found. Unlike an iterative search performed by a conventional processor, a CAM reports results instantly. In essence, each slot in a CAM contains hardware that performs the comparison. We can imagine wires leading from the bits of the key down through all the slots. Each slot contains gates that compare bits of the key to bits of the value in the slot. Because the hardware for all slots operates in parallel, the time required to perform the search does not depend on the number of slots.

Of course, parallel search hardware makes CAM extremely expensive because the search mechanism must be replicated for each slot. It also means CAM consumes much more power (and produces much more heat) than a conventional memory. Thus, an architect only uses a CAM when lookup speed is more important than cost and power consumption. For example, in a high-speed Internet router, the system must check each incoming packet to determine whether other packets have arrived previously from the same source. To handle high-speed connections, some designs use a CAM to store a list of source identifiers. The CAM allows a search to be performed fast enough to accommodate packets arriving at a high rate (i.e., over a high-speed network).

11.27 Ternary CAM

An alternative form of CAM, known as *Ternary CAM* (*TCAM*), extends the idea of CAM to provide *partial match searches*. In essence, each bit in a slot can have three values: zero, one, or "don't care." Like a standard CAM, a TCAM performs the search operation in parallel by examining all slots simultaneously. Unlike a standard CAM, a TCAM only performs the match on bits that have the value zero or one. Partial matching allows a TCAM to be used in cases where two or more entries in the CAM overlap — a TCAM can find the best match (e.g., the *longest prefix* match).

11.28 Summary

We examined two aspects of physical memory: the underlying technology and the memory organization. Many memory technologies exist. Differences among them include permanence (RAM or ROM), clock synchronization, and the read and write cycle times.

Physical memory is organized into words and accessed through a controller. Although programmers find byte addressing convenient, most underlying memory systems use word addressing. An intelligent memory controller can translate from byte addressing to word addressing. To avoid arithmetic computation in a controller, memory is organized so the address space and bytes per word are powers of two.

Programming languages, such as C, provide pointer variables and pointer arithmetic that allow a programmer to obtain and manipulate memory addresses. A memory dump, which shows the contents of memory along with the memory address of each location, can be used to relate data structures in a program to values in memory at runtime.

Memory banks and interleaving both employ multiple memory modules. Banks are used to organize a large memory out of smaller modules. Interleaving places successive words of memory in separate modules to speed sequential access.

Content Addressable Memory (CAM) combines memory technology and memory organization. A CAM organizes memory as an array of slots, and provides a high-speed search mechanism.

EXERCISES

11.1 Smart phones and other portable devices typically use DRAM rather than SRAM. Explain why.

11.2 Explain the purpose of a DRAM *refresh* mechanism.

11.3 Assume a computer has a physical memory organized into 64-bit words. Give the word address and offset within the word for each of the following byte addresses: 0, 9, 27, 31, 120, and 256.

11.4 Extend the above exercise by writing a computer program that computes the answer. The program should take a series of inputs that each consist of two values: a word size specified in bits and a byte address. For each input, the program should generate a word address and offset within the word. Note: although it is specified in bits, the word size must be a power of two bytes.

11.5 On an ARM processor, attempting to load an integer from memory will result in an error if the address is not a multiple of 4 bytes. What term do we use to refer to such an error?

11.6 If a computer has 64-bit addresses, and each address corresponds to one byte, how many gigabytes of memory can the computer address?

11.7 Compute the number of memory operations required for a two-address instruction if the instruction and both operands are unaligned.

11.8 Write a C function that declares a static integer array, *M*, and implements *fetch* and *store* operations that use shift and Boolean operations to access individual bytes.

11.9 Find the memory in a PC, identify the type of chips used, and look up the vendor's specification of the chips to determine the memory type and speed.

11.10 Redraw Figure 11.13 for an 8-way interleaved memory.

11.11 Emulate a physical memory. Write a C program that declares an array, *M*, to be an array of 10,000 integers (i.e., an array of words). Implement two functions, *fetch* and *store*, that use array *M* to emulate a byte-addressable memory. *fetch(i)* returns the i^{th} byte of the memory, and *store(i,ch)* stores the 8-bit character *ch* into the i^{th} byte of the memory. Do not use byte pointers. Instead, use the ideas in this chapter to write code that computes the correct word that contains the specified byte and the offset within the word.

11.12 Simulate a TCAM. Write a program that matches an input string with a set of patterns. For the simulation, use characters instead of bits. Allow each pattern to contain a string of characters, and interpret an asterisk as a "wild card" that matches any character. Can you find a way to make the match proceed faster than iterating through all patterns?

Chapter Contents

12

Caches And Caching

12.1 Introduction

The previous chapter discusses physical memory systems, focusing on the underlying technologies used to build memory systems and the organization of address spaces. The chapter also discusses the organization of a physical memory into words.

This chapter takes a different view of the problem: instead of concentrating on technologies used to construct memory systems, the chapter focuses on a technology used to improve memory system performance. The chapter presents the fundamental concept of caching, shows how caching is used in memory systems, explains why caching is essential, and describes why caching achieves high performance with low cost.

12.2 Information Propagation In A Storage Hierarchy

Recall from Chapter 10 that storage mechanisms are organized into a hierarchy that includes general-purpose registers, main memory, and secondary storage. Data items migrate up and down the hierarchy, usually under control of software. In general, items move up the hierarchy when they are read, and down the hierarchy when they are written. For example, when generating code for an arithmetic computation, a compiler arranges to move items from memory into registers. After the computation finishes, the result may be moved back to memory. If an item must be kept after the program finishes, the programmer will arrange to copy the item from memory to secondary storage. We will see how caching fits into the storage hierarchy, and will learn that a memory cache uses hardware rather than software to move items up and down in its part of the hierarchy.

12.3 Definition of Caching

The term *caching* refers to an important optimization technique used to improve the performance of any hardware or software system that retrieves information. In memory systems, caching can reduce the Von Neumann bottleneck†. A *cache* acts as an intermediary. That is, a cache is placed on the path between a mechanism that makes requests and a mechanism that answers requests, and the cache is configured to intercept and handle all requests.

The central idea in caching is high-speed, temporary storage: the cache keeps a local copy of selected data, and answers requests from the local copy whenever possible. Performance improvement arises because the cache is designed to return answers faster than the mechanism that normally fulfills requests. Figure 12.1 illustrates how a cache is positioned between a mechanism that makes requests and a mechanism that answers requests.

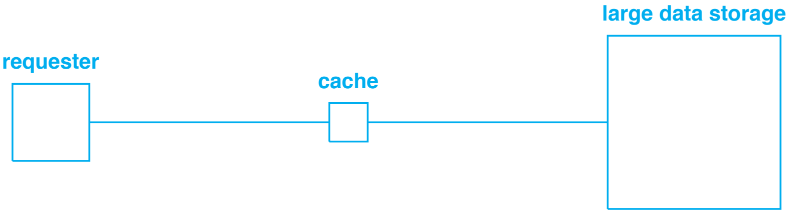


Figure 12.1 Conceptual organization of a cache, which is positioned on the path between a mechanism that makes requests and a storage mechanism that answers requests.

12.4 Characteristics Of A Cache

The above description is purposefully vague because caching is a broad concept that appears in many forms in computer and communication systems. This section clarifies the definition by explaining the concept in more detail; later sections give examples of how caching can be used.

Although a variety of caching mechanisms exist, they share the following general characteristics:

- Small
- Active
- Transparent
- Automatic

Small. To keep economic cost low, the amount of storage associated with a cache is much smaller than the amount of storage needed to hold the entire set of data items. Most cache sizes are less than ten percent of the main storage size; in many cases, a

†The Von Neumann bottleneck is defined on page 131.

cache holds less than one percent as much as the data store. Thus, one of the central design issues revolves around the selection of data items to keep in the cache.

Active. A cache contains an active mechanism that examines each request and decides how to respond. Activities include: checking to see if a requested item is available in the cache, retrieving a copy of an item from the data store if the item is not available locally, and deciding which items to keep in the cache.

Transparent. We say that a cache is *transparent*, which means that a cache can be inserted without making changes to the requester or data store. That is, the interface the cache presents to the requester is exactly the same as the interface a data store presents, and the interface the cache presents to the data store is exactly the same as the interface a requester presents.

Automatic. In most cases, a cache mechanism does not receive instructions on how to act or which data items to store in the cache storage. Instead, a cache implements an algorithm that examines the sequence of requests, and uses the requests to determine how to manage the cache.

12.5 Cache Terminology

Although caching is used in a variety of contexts, some of the terminology related to caching has universal acceptance across all types of caching systems. A *cache hit* (abbreviated *hit*) is defined as a request that can be satisfied by the cache without access to the underlying data store. Conversely, a *cache miss* (abbreviated *miss*) is defined as a request that cannot be satisfied by the cache.

Another term characterizes a sequence of references presented to a cache. We say that a sequence of references exhibits *high locality of reference* if the sequence contains repetitions of the same requests; otherwise, we say that the sequence has *low locality of reference*. We will see that high locality of reference leads to higher performance. Locality refers to items in the cache. Therefore, if a cache stores large data items (e.g., pages of memory), repeated requests do not need to be identical as long as they reference the same item in the cache (e.g., memory references to items on the same page).

12.6 Best And Worst Case Cache Performance

We said that if a data item is stored in the cache, the cache mechanism can return the item faster than the data store. As Figure 12.2 shows, we represent the costs of retrieval from the requester’s view.

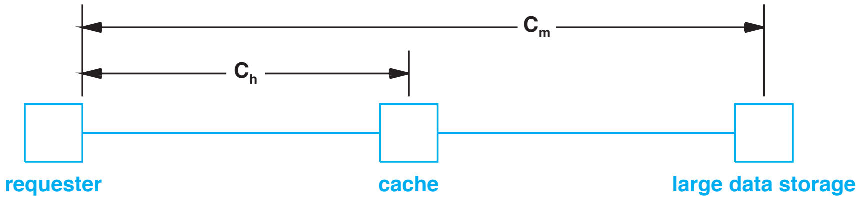


Figure 12.2 Illustration of access costs when using a cache. Costs are measured with respect to the requester.

In the figure, C_h is the cost if an item is found in the cache (i.e., a hit), and C_m is the cost if an item is not found in the cache (i.e., a miss). Interestingly, individual costs are not informative. Observe that because a cache uses the contents of requests to determine which items to keep, the performance depends on the sequence of requests. Thus, to understand caching, we must examine the performance on a sequence of requests. For example, we can easily analyze the best and worst possible behavior for a sequence of N requests. At one extreme, if each request references a new item, caching does not improve performance at all — the cache must forward each request to the data store. Thus, in the worst case, the cost is:

$$C_{worst} = N\ C_m \tag{12.1}$$

It should be noted that our analysis ignores the administrative overhead required to maintain the cache. If we divide by N to compute the average cost per request, the result is $C_{m.}$

At the other extreme, if all requests in the sequence specify the same data item (i.e., the highest locality of reference), the cache can indeed improve performance. When it receives the first request, the cache fetches the item from the data store and saves a copy; subsequent requests can be satisfied by using the copy in the cache. Thus, in the best case. the cost is:

$$C_{best} = C_m + (N-1)\ C_h \tag{12.2}$$

Dividing by N produces the cost per request:

$$C_{per\_request} = \frac{C_m + (N-1)\ C_h}{N} = \frac{C_m}{N} - \frac{C_h}{N} + C_h \tag{12.3}$$

As $N \rightarrow \infty$, the first two terms approach zero, which means that the cost per request in the best case becomes C_h. We can understand why caching is such a powerful tool:

> *If one ignores overhead, the worst case performance of caching is no worse than if the cache were not present. In the best case, the cost per request is approximately equal to the cost of accessing the cache, which is lower than the cost of accessing the data store.*

12.7 Cache Performance On A Typical Sequence

To estimate performance of a cache on a typical sequence of requests, we need to examine how the cache handles a sequence that contains both hits and misses. Cache designers use the term *hit ratio* to refer to the percentage of requests in the sequence that are satisfied from the cache. Specifically, the hit ratio is defined to be:

$$\textit{hit ratio} \;=\; \frac{\textit{number of requests that are hits}}{\textit{total number of requests}} \qquad (12.4)$$

The hit ratio is a value between zero and one. We define a *miss ratio* to be one minus the hit ratio.

Of course, the actual hit ratio depends on the specific sequence of requests. Experience has shown that for many caches, the hit ratio tends to be nearly the same across the requests encountered in practice. In such cases, we can derive an equation for the cost of access in terms of the cost of a miss and the cost of a hit:

$$\textit{Cost} \;=\; r\, C_h \;+\; (1-r)\, C_m \qquad (12.5)$$

where r is the hit ratio defined in Equation 12.4 above.

The cost of accessing the data store, given by C_m in the equation, is fixed. Thus, there are two ways a cache designer can improve the performance of a cache: increase the hit ratio or decrease the cost of a hit.

12.8 Cache Replacement Policy

How can a cache designer increase the hit ratio? There are two ways:

- Increase the cache size
- Improve the replacement policy

Increase the cache size. Recall that a cache is usually much smaller than a large data store. When it begins, a cache keeps a copy of each response. Once the cache storage is full, an item must be removed from the cache before a new item can be added. A larger cache can store more items.

Improve the replacement policy. A cache uses a *replacement policy* to decide which item to remove when a new item is encountered and the cache is full. The replacement policy specifies whether to ignore the new item or how to choose an item to evict to make space for the new item. A replacement policy that chooses to keep those items that will be referenced again can increase the hit ratio.

12.9 LRU Replacement

What replacement policy should be used? There are two issues. First, to increase the hit ratio, the replacement policy should retain those items that will be referenced most frequently. Second, the replacement policy should be inexpensive to implement, especially for a memory cache. One replacement policy that satisfies both criteria has become extremely popular. Known as *Least Recently Used (LRU)*, the policy specifies replacing the item that was referenced the longest time in the past†.

LRU is easy to implement. The cache mechanism keeps a list of data items that are currently in the cache. When an item is referenced, the item moves to the front of the list; when replacement is needed, the item at the back of the list is removed.

LRU works well in many situations. In cases where the set of requests has a high locality of reference (i.e., where a cache can improve performance), a few items will be referenced again and again. LRU tends to keep those items in the cache, which means the cost of access is kept low.

We can summarize:

> *When its storage is full and a new item arrives, a cache must choose whether to retain the current set of items or replace one of the current items with the new item. The Least Recently Used (LRU) policy is a popular choice for replacement because it is trivial to implement and tends to keep items that will be requested again.*

12.10 Multilevel Cache Hierarchy

One of the most unexpected and astonishing aspects of caching arises from the use of caching to improve the performance of a cache! To understand how such an optimization is possible, recall that the insertion of a cache lowers the cost of retrieving items by placing some of the items closer to the requester. Now imagine an additional cache placed between the requester and the existing cache as Figure 12.3 illustrates.

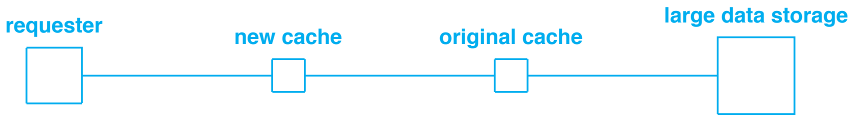


Figure 12.3 The organization of a system with an additional cache inserted.

†Note that “least recently” always refers to how long ago the item was last referenced, not to the number of accesses.

Can a second cache improve performance? Yes, provided the cost to access the new cache is lower than the cost to access the original cache (e.g., the new cache is closer to the requester). In essence, the cost equation becomes:

$$Cost \quad = \quad r_1 \, C_{h1} \quad + \quad r_2 \, C_{h2} \quad + \quad (1 - r_1 - r_2) \, C_m \tag{12.6}$$

where r_1 denotes the fraction of hits for the new cache, r_2 denotes the fraction of hits for the original cache, C_{h1} denotes the cost of accessing the new cache, and C_{h2} denotes the cost of accessing the original cache.

When more than one cache is used along the path from requester to data store, we say that the system implements a *multilevel cache hierarchy*. A set of Web caches provides an example of a multilevel hierarchy. The path between a browser running on a user's computer can pass through a cache at the user's ISP as well as the local cache mechanism used by the browser.

The point is:

> *Adding an additional cache can be used to improve the performance of a system that uses caching. Conceptually, the caches are arranged in a multilevel hierarchy.*

12.11 Preloading Caches

How can cache performance be improved further? Cache designers observe that although many cache systems perform well in the steady state (i.e., after the system has run for awhile), the system exhibits higher cost during startup. That is, the initial hit ratio is extremely low because the cache must fetch items from the data store. In some cases, the startup costs can be lowered by *preloading* the cache. That is, values are loaded into the cache before execution begins.

Of course, preloading only works in cases where the cache can anticipate requests. For example, an ISP's Web cache can be preloaded with *hot* pages (i.e., pages that have been accessed frequently in the past day or pages for which the owner expects frequent access). As an alternative, some caches use an automated method of preloading. In one form, the cache periodically places a copy of its contents on nonvolatile storage, allowing recent values to be preloaded at startup. In another form, the cache uses a reference to *prefetch* related data. For example, if a processor accesses a byte of memory, the cache can fetch 128 bytes. Thus, if the processor accesses the next byte, which is likely, the value will come from the cache.

Prefetching is especially important for Web pages. A typical Web page contains references to multiple images, and before the page can be displayed, a browser must download a copy of each image and cache the copy on the user's computer. As a page is being downloaded, a browser can scan for references to images, and can begin to prefetch each of the images without waiting for the entire page to download.

12.12 Caches Used With Memory

Now that we understand the basic idea of caching, we can consider some of the ways caches are used in memory systems. In fact, the concept of caching originated with computer memory systems†. The original motivation was higher speed at low cost. Because memory was both expensive and slow, architects looked for ways to improve performance without incurring the cost of higher-speed memory. The architects discovered that a small amount of high-speed cache improved performance dramatically. The result was so impressive that by the 1980s, most computer systems had a single cache located between the processor and memory. Physically, memory was on one circuit board and the cache occupied a separate circuit board, which allowed computer owners to upgrade the memory or the cache independently. As described above, a caching hierarchy can increase performance more than a single cache. Therefore, we will see that modern computers employ a hierarchy of memory caches and use caching in a variety of ways. The next sections present a few examples.

12.13 Physical Memory Cache

Caching has become popular as a way to achieve higher memory performance without significantly higher cost. Early computers used a physical memory system. That is, when it generated a request, the processor specified a physical address, and the memory system responded to the physical address. Thus, to be inserted on the path between a processor and the memory, a cache had to understand and use physical addresses.

It may seem that a physical memory cache is trivial. We can imagine the memory cache receiving a *fetch* request, checking to see if the request can be answered from the cache, and then, if the item is not present, passing the request to the underlying memory. Furthermore, we can imagine that once an item has been retrieved from the underlying memory, the cache saves a copy locally, and then returns the value to the processor.

In fact, our imagined scenario is misleading — a physical memory cache is much more complex than the above description. To understand why, we must remember that hardware achieves high speed through parallelism. For example, when it encounters a *fetch* request, a memory cache does not check the cache and then access the physical memory. Instead, the cache hardware performs two tasks in parallel: the cache simultaneously passes the request to the physical memory and searches for an answer locally. If it finds an answer locally, the cache must cancel the memory operation. If it does not find an answer locally, the cache must wait for the underlying memory operation to complete. Furthermore, when an answer does arrive from memory, the cache uses parallelism again by simultaneously saving a local copy of the answer and transferring the answer back to the processor. Parallel activities make the hardware complex. The point is:

†In addition to introducing the use of microcode, Maurice Wilkes is credited with inventing the concept of a memory cache in 1965.

> *To achieve high performance, a physical memory cache is designed to search the local cache and access the underlying memory simultaneously. Parallelism complicates the hardware.*

12.14 Write Through And Write Back

In addition to parallelism, memory caches are also complicated by *write* (i.e., *store*) operations. There are two issues: performance and coherence. Performance is easiest to understand: caching improves the performance for retrieval requests, but not for storage requests. That is, a *write* operation takes longer because a *write* operation must change the value in the underlying memory. More important, in addition to forwarding the request to the memory, a cache must also check to see whether the item is in the cache. If so, the cache must update its copy as well. In fact, experience has shown that a memory cache should always keep a local copy of each value that is written because programs tend to access a value a short time after it has been stored.

Initial implementations of memory caches handled *write* operations as described above: the cache kept a copy and forwarded the *write* operation to the underlying memory. We use the term *write-through cache* to describe the approach.

The alternative, known as *write-back* cache, keeps a copy of a data item that is written, and waits until later to update the underlying physical memory. To know whether the underlying physical memory must be updated, a write-back cache keeps an extra bit with each item that is known as the *dirty bit*. In a physical memory cache, a dirty bit is associated with each block in the cache. When an item is fetched and a copy is placed in the cache, the dirty bit is initialized to zero. When the processor modifies the item (i.e., performs a *write*), the dirty bit is set to one. When it needs to eject a block from the cache, the hardware first examines the dirty bit associated with the block. If the dirty bit is one, a copy of the block is written to memory. If the dirty is zero, however, the block can simply be overwritten because data in the block is exactly the same as the copy in memory. The point is:

> *A write-back cache associates a dirty bit with each block to record whether the block has been modified since it was fetched. When ejecting a block from the cache, the hardware writes a copy of a dirty block to memory, but simply overwrites the contents if the block is not dirty.*

To understand why write-back improves performance, imagine a *for loop* in a program that increments a variable in memory on each iteration of the loop. A write-back cache places the variable in the cache the first time the variable is referenced. On each successive iteration, changes to the variable only affect the cached copy. Assume that once the loop ends, the program stops referencing the variable. Eventually, the program generates enough other references so that the variable is the least recently used item in the cache, and will be selected for replacement. When a new item is referenced and a cache slot is needed, the cache writes the value of the variable to the underlying physical memory. Thus, although the variable can be referenced or changed many times, the memory system only has one access to the underlying physical memory†.

12.15 Cache Coherence

Memory caches are especially complex in a system with multiple processors (e.g., a multicore CPU). We said that a write-back cache achieves higher performance than a write-through cache. In a multiprocessor environment, performance is also optimized by giving each core its own cache. Unfortunately, the two optimizations conflict. To understand why, look at the architecture in Figure 12.4, which shows two processors that each have a private cache.

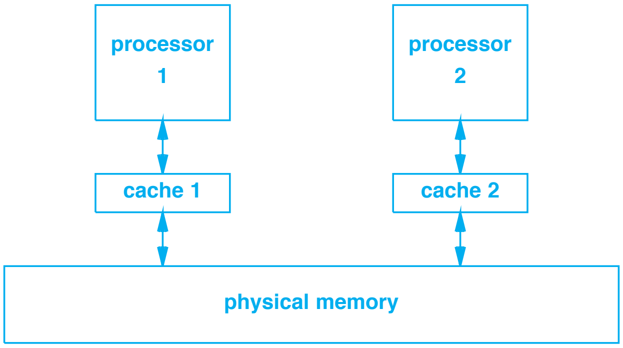


Figure 12.4 Illustration of two processors sharing an underlying memory. Because each processor has a separate cache, conflicts can occur if both processors reference the same memory address.

Now consider what happens if the two caches use a write-back approach. When processor 1 writes to a memory location *X*, cache 1 holds the value for *X*. Eventually, when it needs space, cache 1 writes the value to the underlying physical memory. Similarly, whenever processor 2 writes to a memory location, the value will be placed in cache 2 until space is needed. The problem should be obvious: without an additional mechanism, incorrect results will occur if both processors simultaneously issue *read* and *write* operations for a given address.

To avoid conflicts, all caches that access a given memory must follow a *cache coherence protocol* that coordinates the values. For example, when processor 2 reads from an address, A, the coherence protocol requires cache 2 to inform cache 1. If it currently holds address A, cache 1 writes A to the physical memory so cache 2 can obtain the most recent value. That is, a *read* operation on any processor triggers a write-

†An optimizing compiler can further improve performance by using a general-purpose register to hold the variable until the loop finishes (another form of caching).

back in any cache that currently holds a cached copy of the address. Similarly, if any processor issues a *write* operation for an address, A, all other caches must be informed to discard cached values of A. Thus, in addition to requiring additional hardware and a mechanism that allows the caches to communicate, cache coherency introduces additional delay.

12.16 L1, L2, and L3 Caches

We said that arranging multiple caches into a hierarchy can improve overall performance. Indeed, most computer memory systems have at least two levels of cache hierarchy. To understand why computer architects added a second level of cache to the memory hierarchy, we must consider four facts:

- A traditional memory cache was separate from both the memory and the processor.
- To access a traditional memory cache, a processor used pins that connect the processor chip to the rest of the computer.
- Using pins to access external hardware takes much longer than accessing functional units that are internal to the processor chip.
- Advances in technology have made it possible to increase the number of transistors per chip, which means a processor chip can contain more hardware.

The conclusion should be clear. We know that adding a second cache can improve memory system performance, we further know that placing the second cache on the processor chip will make the cache access times much lower, and we know that technology now allows chip vendors to add more hardware to their chips. So, it makes sense to embed a second memory cache in the processor chip itself. If the hit ratio is high, most data references will never leave the processor chip — the effective cost of accessing memory will be approximately the same as the cost of accessing a register.

To describe the idea of multiple caches, computer manufacturers originally adopted the terms *Level 1 cache* (*L1 cache*) to refer to the cache onboard the processor chip, *Level 2 cache* (*L2 cache*) to refer to an external cache, and *Level 3 cache* (*L3 cache*) to refer to a cache built into the physical memory. That is, an L1 cache was originally *on-chip* and an L2 or L3 cache was *off-chip*.

In fact, chip sizes have become so large that a single chip can contain multiple cores and multiple caches. In such cases, manufacturers use the term *L1 cache* to describe a cache that is associated with one particular core, the term *L2 cache* to describe an on-chip cache that may be shared, and the term *L3 cache* to describe an on-chip cache that is shared by multiple cores. Typically, all cores share an L3 cache. Thus, the distinction between on-chip and off-chip has faded.

We can summarize the terminology:

> *When using traditional terminology for a multilevel cache hierarchy, an L1 cache is embedded on the processor chip, an L2 cache is external to the processor, and an L3 cache is built into the physical memory. More recent terminology defines an L1 cache to be associated with a single core, whereas L2 and L3 refer to on-chip caches that all cores share.*

12.17 Sizes Of L1, L2, And L3 Caches

Most computers employ a cache hierarchy. Of course, the cache at the top of the hierarchy is the fastest, but also the smallest. Figure 12.5 lists example cache memory sizes. The L1 cache may be divided into separate instruction and data caches, as described in the next section.

Cache	Size	Notes
L1	64 KB to 96 KB	Per core
L2	256 KB to 2 MB	May be per core
L3	8 MB to 24 MB	Shared among all cores

Figure 12.5 Example cache sizes in 2016. Although absolute sizes continue to change; readers should focus on the amount of cache relative to RAM that is 4 GB to 32 GB.

12.18 Instruction And Data Caches

Should all memory references pass through a single cache? To understand the question, imagine instructions being executed and data being accessed. Instruction fetch tends to behave with high locality — in many cases, the next instruction to be executed is found at an adjacent memory address. Furthermore, the most time-consuming loops in a program are usually small, which means the entire loop can fit into a cache. Although the data access in some programs exhibits high locality, the data access in others does not. For example, when a program accesses a hash table, the locations referenced appear to be random (i.e., the location referenced in one instant is not necessarily close to the location referenced in the next).

Differences between instruction and data behavior raise the question of how intermixing the two types of references affects a cache. In essence, the more random the sequence of requests becomes, the worse a cache performs (because the cache will save each value, even though the value will not be needed again). We can state a general principle:

> *Inserting random references in the series of requests tends to worsen cache performance; reducing the number of random references that occurs tends to improve cache performance.*

12.19 Modified Harvard Architecture

Is performance optimized by having a separate cache for instructions and data? The simplistic answer is obvious. When both data and instructions are placed in the same cache, data references tend to push instructions out of the cache, lowering performance. Adding a separate instruction cache will improve performance.

The simplistic answer above is insufficient, however, because the question is not whether additional hardware will help, but how to choose among tradeoffs. Because additional hardware will generate more heat, consume more power, and in portable devices, deplete the battery faster, an architect must weigh all the costs of an additional cache. If an architect does decide to add more cache hardware, the question is how best to use the hardware. We know, for example, that increasing the size of a single cache will increase performance by avoiding collisions. If a cache becomes sufficiently large, intermixing instructions and data references will work fine. Would it be better to add a separate instruction cache or to retain a single cache and increase the size?

Many architects have decided that the optimal way to use a modest amount of additional hardware lies in introducing a new *I-cache* (*instruction cache*) and using the existing cache as a *D-cache* (*data cache*). Separating instruction and data caches is trivial in a Harvard Architecture because an I-cache is associated with the instruction memory and a D-cache is associated with the data memory. Should architects abandon the Von Neumann Architecture?

Many architects have adopted a compromise in which a computer has separate instruction and data caches, but the caches lead to a single memory. We use the term *Modified Harvard Architecture* to characterize the compromise. Figure 12.6 illustrates the modified architecture.

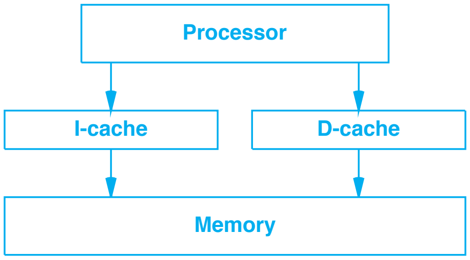


Figure 12.6 A Modified Harvard Architecture with separate instruction and data caches leading to the same underlying memory.

12.20 Implementation Of Memory Caching

Conceptually, each entry in a memory cache contains two values: a memory address and the value of the byte found at that address. In practice, storing a complete address with each entry is inefficient. Therefore, memory caches use clever techniques to reduce the amount of space needed. The two most important cache optimization techniques are known as:

- Direct mapped memory cache
- Set associative memory cache

We will see that, like virtual memory schemes, both cache implementations use powers of two to avoid arithmetic computation.

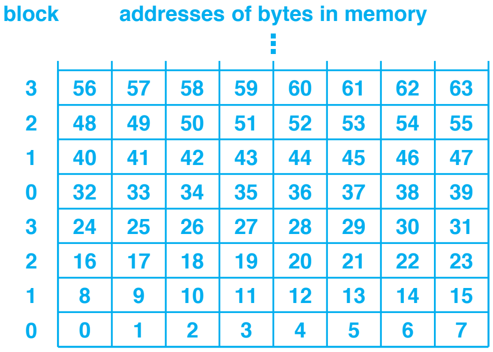
12.21 Direct Mapped Memory Cache

A *direct mapped memory cache* uses a mapping technique to avoid overhead. Although memory caches are used with byte-addressable memories, a cache does not record individual bytes. Instead, a cache divides both the memory and the cache into a set of fixed-size blocks, where the block size, *B* (measured in bytes), is chosen to be a power of two. The hardware places an entire block in the cache whenever a byte in the block is referenced. Using cache terminology, we refer to a block in the cache as a *cache line*; the size of a direct mapped memory cache is often specified by giving the number of cache lines times the size of a cache line. For example, the size might be specified as 4K lines with 8 bytes per line. To envision such a cache, think of bytes in memory being divided into 8-byte segments and assigned to lines of the cache. Figure 12.7 illustrates how bytes of memory would be assigned for a block size of eight in a cache that has four lines. (Note: a memory cache usually holds many more than four lines; a small cache size has been chosen merely as a simplified example for the figure.)

Observe that the blocks in memory are numbered modulo *C*, where *C* is the number of slots in the cache. That is, blocks are numbered from zero through C−1 (C is 4 in the figure). Interestingly, using powers of two means that no arithmetic is required to map a byte address to a block number. Instead, the block number can be found by extracting a set of bits. In the figure, the block number can be computed by extracting the fourth and fifth bits of an address. For example, consider the byte with address 57 ($1\underline{11}001$ in binary, shown with the forth and fifth bits underlined). The bits 11 are 3 in decimal, which agrees with the block number in the figure. In address 44 ($1\underline{01}100$ in binary), the fourth and fifth bits are 01 and the block number is 1. We can express the mapping in programming language terms as:

```
b = (byte_address >> 3) & 0x03;
```

In terms of a memory cache, no computation is needed — the hardware places the value in an internal register and extracts the appropriate bits to form a block number.



block	addresses of bytes in memory							
				⋮				
3	56	57	58	59	60	61	62	63
2	48	49	50	51	52	53	54	55
1	40	41	42	43	44	45	46	47
0	32	33	34	35	36	37	38	39
3	24	25	26	27	28	29	30	31
2	16	17	18	19	20	21	22	23
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

Figure 12.7 An example assignment of block numbers to memory locations for a cache of four blocks with eight bytes per block.

The key to understanding a direct mapped memory cache arises from the following rule: only a memory block numbered *i* can be placed in cache slot *i*. For example, the block with addresses 16 through 23 can be placed in slot 2, as can the block with addresses 48 through 55.

If multiple memory blocks can be placed in a given slot, how does the cache know which block is currently in a slot? The cache attaches a unique *tag* to each group of *C* blocks. For example, Figure 12.8 illustrates how tag values are assigned to memory blocks in our example cache that has four slots.

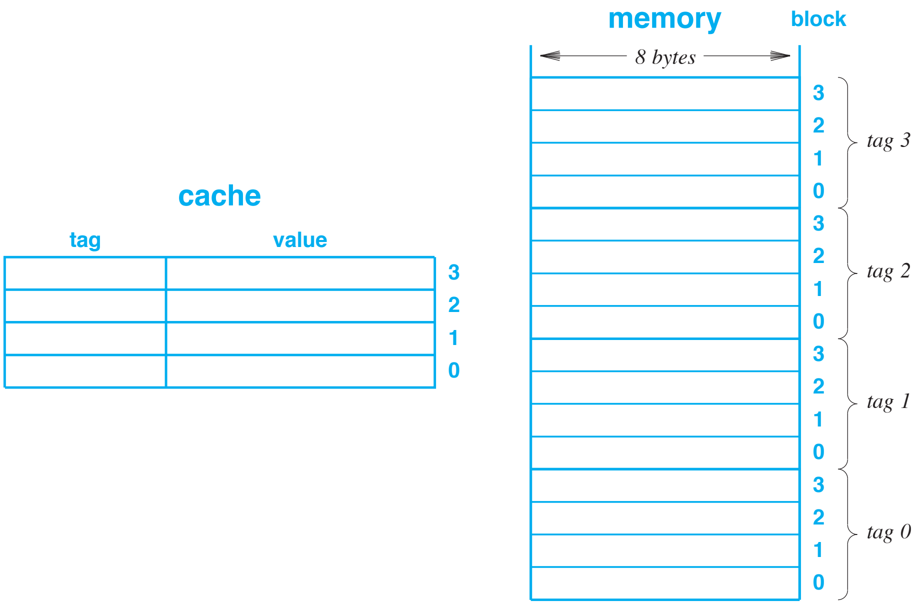


Figure 12.8 An example memory cache with space for four blocks and a memory divided into conceptual blocks of 8 bytes. Each group of four blocks in memory is assigned a unique tag.

To identify the block currently in a slot of the cache, each cache entry contains a tag value. Thus, if slot zero in the cache contains tag K, the value in slot zero corresponds to block zero from the area of memory that has tag K.

Why use tags? A cache must uniquely identify the entry in a slot. Because a tag identifies a large group of blocks rather than a single byte of memory, using a tag requires fewer bits to identify a section of memory than using a full memory address. Furthermore, as the next section explains, choosing the block size and the size of memory identified by a tag to be powers of two makes cache lookup extremely efficient.

12.22 Using Powers Of Two For Efficiency

Although the direct mapping described above may seem complex, using powers of two simplifies the hardware implementation. In fact, the hardware is elegant and extremely efficient. Instead of modulo arithmetic, both the tag and block number can be computed by extracting groups of bits from a memory address. The high-order bits of the address are used as the tag, the next set of bits forms a block number, and the final set of bits gives a byte offset within the block. Figure 12.9 illustrates the division.

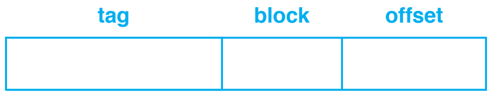


Figure 12.9 Illustration of how using powers of two allows a cache to divide a memory address into three separate fields that correspond to a tag, a block number, and a byte offset within the block.

Once we know that all values can be obtained via bit extraction, the algorithm for lookup in a direct-mapped memory cache is straightforward. Think of the cache as an array. The idea is to extract the block number from the address, and then use the block number as an index into the array. Each entry in the array contains a tag and a value. If the tag in the address matches the tag in the cache slot, the cache returns the value. If the tag does not match, the cache hardware must fetch the block from memory, place a copy in the cache, and then return the value. Algorithm 12.1 summarizes the steps.

The algorithm omits an important detail. Each slot in the cache has a *valid bit* that specifies whether the slot has been used. Initially (i.e., when the computer boots), all valid bits are set to 0 (to indicate that none of the slots contain blocks from memory). When it stores a block in a slot, the cache hardware sets the valid bit to 1. When it examines the tag for a slot, the hardware reports a mismatch if the valid bit is set, which forces a copy of the block to be loaded from memory.

Algorithm 12.1

```
Given:
        A memory address
Find:
        The data byte at that address
Method:
        Extract the tag number, t, block number, b, and offset, o,
          from the address by selecting the appropriate bit fields
        Examine the tag in slot b of the cache
        If the tag in slot b of the cache matches t  {
                Use o to select the appropriate byte from the
                block in slot b, and return the byte
        } else {  /* Update the cache */
                Fetch block b from the underlying memory
                Place a copy in slot b
                Set the tag on slot b to t
                Use o to select the appropriate byte from the
                block in slot b, and return the byte
        }
```

Algorithm 12.1 Cache Lookup In A Direct Mapped Memory Cache

12.23 Hardware Implementation Of A Direct Mapped Cache

Algorithm 12.1 describes cache lookup as if the cache is an array and separate steps are taken to extract items and index the array. In fact, slots of a cache are not stored in an array in memory. Instead, they are implemented with hardware circuits, and the circuits work in parallel. For example, the first step that extracts items from the address can be implemented by placing the address in an internal register (a hardware circuit that consists of a set of latches), and arranging each bit of the address to be represented by the output of one latch. That is, once the address has been placed in the register, each bit of the address will be represented by a separate wire. The items *t*, *b*, and *o* in the address can be obtained merely by dividing the output wires into groups.

The second step in Algorithm 12.1 requires the cache hardware to examine one of the slots. The hardware uses a decoder to select exactly one of the slots. All slots are connected to common output wires; the hardware is arranged so that only a selected slot puts its output on the wires. A comparator circuit is used to compare the tag in the address with the tag in the selected slot. Figure 12.10 gives a simplified block diagram of the hardware to perform cache lookup.

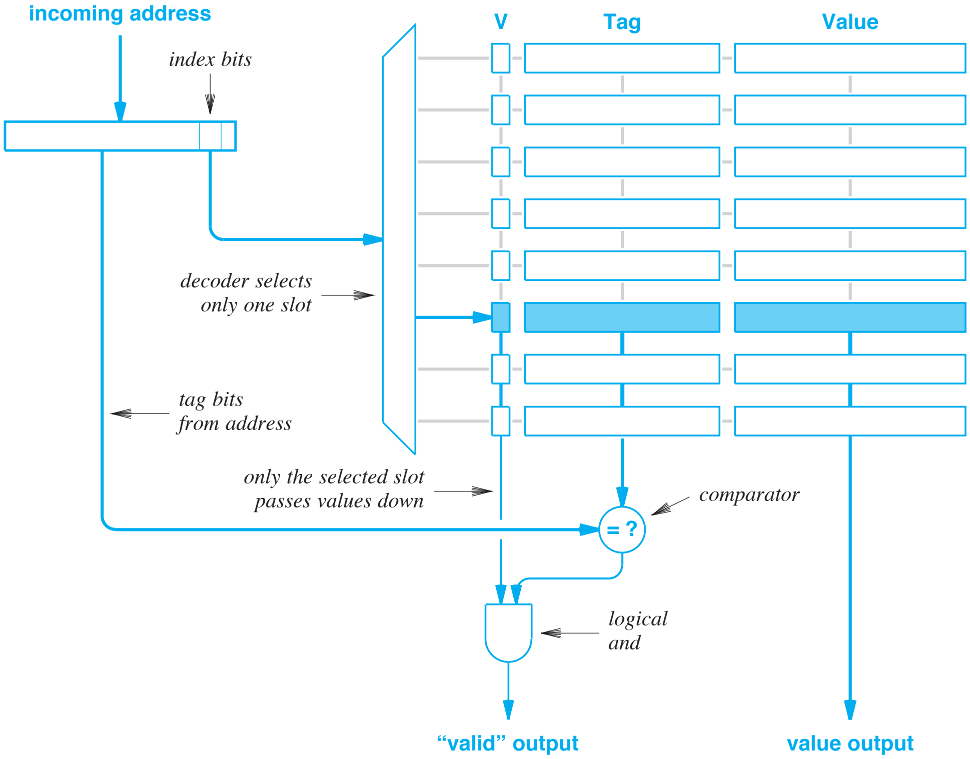


Figure 12.10 Block diagram of the hardware used to implement lookup in a memory cache.

The circuit takes a memory address as an input, and produces two outputs. The *valid* output is 1 if and only if the specified address is found in the cache (i.e., the cache returns a value). The *value* output is the contents of memory at the specified address.

In the figure, each slot has been divided into a valid bit, a tag, and a value to indicate that separate hardware circuits can be used for each field. Horizontal lines from the decoder to each slot indicate a connection that can be used to activate the circuits for the slot. At any time, however, the decoder only selects one slot (in the figure, the selected slot is shown shaded).

Vertical lines through the slots indicate parallel connections. Hardware in each slot connects to the wires, but only a selected slot places a value on the vertical wires. Thus, in the example, the input to the *and* gate only comes from the *V* circuit of the selected slot, the input to the comparator only comes from the *Tag* circuit of the selected slot, and the *value output* only comes from the *Value* circuit of the selected slot. The key point is that cache lookup can be performed quickly by combinatorial circuits.

12.24 Set Associative Memory Cache

The chief alternative to a direct mapped memory cache is known as a *set associative memory cache*. In essence, a set associative memory cache uses hardware parallelism to provide more flexibility. Instead of maintaining a single cache, the set associative approach maintains multiple underlying caches, and provides hardware that can search all of them simultaneously. More important, because it provides multiple underlying caches, a set associative memory cache can store more than one block that has the same number.

As a trivial example, consider a set associative cache in which there are two copies of the underlying hardware. Figure 12.11 illustrates the architecture.

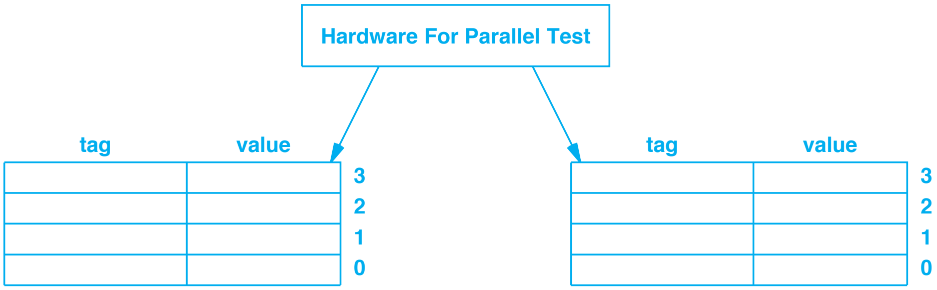


Figure 12.11 Illustration of a set associative memory cache with two copies of the underlying hardware. The cache includes hardware to search both copies in parallel.

To understand the advantages of the set associative approach, consider a reference string in which a program alternately references two addresses, A_1 and A_2, that have different tags, but both have block number zero. In a direct mapped memory cache, the two addresses contend for a single slot in the cache. A reference to A_1 loads the value of A_1 into slot 0 of the cache, and a reference to A_2 overwrites the contents of slot 0 with the value of A_2. Thus, in an alternating sequence of references, every reference results in a cache miss. In a set associative memory cache, however, A_1 can be placed in one of the two underlying caches, and A_2 can be placed in the other. Thus, every reference results in a cache hit.

As the amount of parallelism increases, performance of a set associative memory cache increases. In the extreme case, a cache is classified as *fully associative*, if each of the underlying caches contains only one slot, but the slot can hold an arbitrary value.

Note that the amount of parallelism determines a point on a continuum: with no parallelism, we have a direct mapped memory cache, and with full parallelism, we have the equivalent of a *Content Addressable Memory* (*CAM*).

12.25 Consequences For Programmers

Experience has shown that caching works well for most computer programs. The code that programmers produce tends to contain loops, which means a processor will repeatedly execute a small set of instructions before moving on to another set. Similarly, programs tend to reference data items multiple times before moving on to a new data item. Furthermore, some compilers are aware of caching, and help optimize the generated code to take advantage of the cache.

Despite the overwhelming success of caching, programmers who understand how a cache works can write code that exploits a cache. For example, consider a program that must perform many operations on each element of a large array. It is possible to perform one operation at a time (in which case the program iterates through the array many times) or to perform all the operations on a single element of the array before moving to the next element (in which case the program iterates through the array once). From the point of view of caching, the latter is preferable because the element will remain in the cache.

12.26 Summary

Caching is a fundamental optimization technique that can be used in many contexts. A cache intercepts requests, automatically stores values, and answers requests quickly, whenever possible. Variations include a multilevel cache hierarchy and preloaded caches.

Caches provide an essential performance optimization for memories. Most computer systems employ a multilevel memory cache. Originally, an L1 cache resided on an integrated circuit along with the processor, and an L2 cache was located external to the processor, and an L3 cache was associated with the memory. As integrated circuits became larger, manufacturers moved L2 and L3 caches onto the processor chip, using the distinction that an L1 cache is associated with a single core and L2/L3 caches are shared among multiple cores.

A technology known as a direct mapped memory cache handles lookup without keeping a list of cached items. Although we think of the lookup algorithm as performing multiple steps, a hardware implementation of a direct mapped memory cache can use combinatorial logic circuits to perform the lookup without needing a processor. A set associative memory cache extends the concept of direct mapping to permit parallel access.

EXERCISES

12.1 What does the term *transparent* mean when applied to a memory cache?

12.2 If the hit ratio for a given piece of code is 0.2, the time required to access the cache is 20 nanoseconds, and the time to access the underlying physical memory is 1 microsecond, what is the effective memory access time for the piece of code?

12.3 A physicist writes C code to iterate through a large 2-dimensional array:

```
float  a[32768, 32768], sum;
• • •
for (i=0; i<32768; i++) {
        for (j=0; j<32768; j++) {
        sum += a[j,i];
        }
}
```

The physicist complains that the code is running very slowly. What simple change can you make that will increase execution speed?

12.4 Consider a computer where each memory address is thirty-two-bits long and the memory system has a cache that holds up to 4K entries. If a naive cache is used in which each entry of the cache stores an address and a byte of data, how much total storage is needed for the cache? If a direct mapped memory cache is used in which each entry stores a tag and a block of data that consists of four bytes, how much total storage is needed?

12.5 Extend the previous exercise. Assume the size of the cache is fixed, and find an alternative to the naive solution that allows the storage of more data items. Hint: what values are placed in the cache if the processor always accesses four-byte integers in memory?

12.6 Consult vendors' specifications and find the cost of memory access and the cost of a cache hit for a modern memory system (C_h and C_m in Section 12.6).

12.7 Use the values obtained in the previous exercise to plot the effective memory access cost as the hit ratio varies from zero to one.

12.8 Using the values of C_h and C_m obtained in Exercise 12.6, what value of the hit ratio, r, is needed to achieve an improvement of 30% in the mean access time of a memory system (as compared to the same memory system without a cache)?

12.9 State two ways to improve the hit ratio of a cache.

12.10 What is cache coherence, and what type of system needs it?

12.11 Write a computer program to simulate a direct mapped memory cache using a cache of 64 blocks and a block size of 128 bytes. To test the program, create a 1000 x 1000 array of integers. Simulate the address references if a program walks the array in row-major order and column-major order. What is the hit ratio of your cache in each case?

12.12 The hardware diagram in Figure 12.10 only shows the circuits needed for lookup. Extend the diagram to include circuits that load a value into the cache from memory.

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

13

Virtual Memory Technologies And Virtual Addressing

13.1 Introduction

The previous chapters discuss physical memory and caching. The chapter on physical memory considers the hardware technologies used to create memory systems, the organization of physical memory into words, and the physical addressing scheme used to access memory. The chapter on caching describes how a memory cache is organized, and explains why a memory cache improves performance dramatically.

This chapter considers the important concept of virtual memory. It examines the motivation, the technologies used to create virtual address spaces, and the mapping between virtual and physical memory. Although our focus is primarily on the hardware systems, we will learn how an operating system uses virtual memory facilities.

13.2 Definition Of Virtual Memory

We use the term *Virtual Memory* (*VM*) to refer to a mechanism that hides the details of the underlying physical memory to provide a more convenient memory environment. In essence, a virtual memory system creates an illusion — an address space and a memory access scheme that overcome limitations of the physical memory and physical addressing scheme. The definition may seem vague, but we need to encompass a wide variety of technologies and uses. The next sections will define the concept more precisely by giving examples of virtual memory systems that have been created and the technologies used to implement each. We will learn that the variety in virtual memory schemes arises because no single scheme is optimal in all cases.

We have already seen an example of a memory system that fits our definition of virtual memory in Chapter 11: an intelligent memory controller that provides byte addressing with an underlying physical memory that uses word addressing. The implementation consists of a controller that allows a processor to specify requests using byte addressing. We further saw that choosing sizes to be powers of two avoids arithmetic computation and makes the translation of byte addresses to word addresses trivial.

13.3 Memory Management Unit And Address Space

Architects use the term *Memory Management Unit* (*MMU*) to describe an intelligent memory controller. An MMU creates a *virtual address space* for the processor. The addresses a processor uses are *virtual addresses* because the MMU translates each address into an underlying physical memory. We classify the entire mechanism as a *virtual memory system* because it is not part of the underlying physical memory.

Informally, to help distinguish virtual memory from physical memory, engineers use the adjective *real* to refer to a physical memory. For example, they might use the term *real address* to refer to a physical address, or the term *real address space* to refer to the set of addresses recognized by the physical memory.

13.4 An Interface To Multiple Physical Memory Systems

An MMU that can map from byte addresses to underlying word addresses can be extended to create more complex memory organizations. For example, Intel designed a network processor that used two types of physical memory: SRAM and DRAM. Recall that SRAM is faster than DRAM, but costs more, so the system had a smaller amount of SRAM (intended for items that were accessed frequently) and a large amount of DRAM (intended for items that were not accessed frequently). Furthermore, the SRAM physical memory was organized with four bytes per word and the DRAM physical memory was organized with eight bytes per word. Intel's network processor used an embedded RISC processor that could access both memories. More important, the RISC processor used byte addressing. However, rather than using separate instructions or operand types to access the two memories, the Intel design followed a standard approach: it integrated both physical memories into a single virtual address space.

To implement a uniform virtual address space out of two dissimilar physical memory systems, the memory controller must perform the necessary translations. In essence, the MMU must supply an abstraction that hides details of the underlying memory systems. Figure 13.1 illustrates the overall architecture.

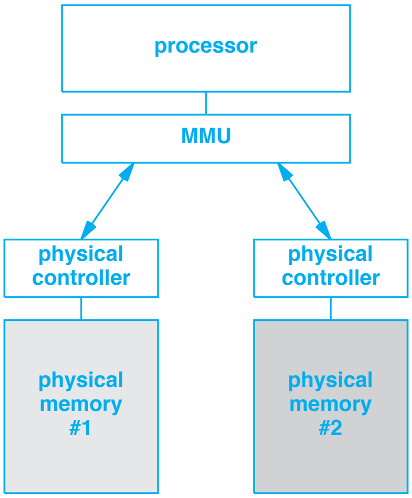


Figure 13.1 Illustration of an architecture in which two dissimilar memories connect to a processor. The processor can use either memory.

In the figure, the processor connects to a Memory Management Unit. The MMU receives memory requests from the processor, translates each request, and forwards the request to the controller for physical memory 1 or to the controller for physical memory 2. The controllers for the two physical memories operate as described in Chapter 11 — a controller accepts a request that specifies byte addressing, and translates the request into operations that use word addressing.

How can the hardware in Figure 13.1 provide a virtual address space? The answer is related to the memory banks described in Chapter 11. Conceptually, the MMU divides the address space into two parts, which the MMU associates with physical memory 1 and physical memory 2. For example, if each physical memory contains a gigabyte (0x40000000 bytes) of RAM, the MMU can create a virtual address space that maps addresses 0 through 0x3fffffff to the first memory and addresses 0x40000000 through 0x7fffffff to the second memory. Figure 13.2 illustrates the resulting virtual memory system.

13.5 Address Translation Or Address Mapping

Each of the underlying memory systems in Figure 13.2 operates like an independent physical memory — the hardware expects requests to reference addresses beginning at zero. Thus, each of the two memories recognizes the same set of addresses. For memory 1, the virtual addresses associated with the memory cover the same range as the hardware expects. For memory 2, however, the processor generates virtual addresses starting at 0x40000000, so the MMU must *map* an address to the lower range (i.e., real addresses 0 through 0x3fffffff) before passing a request to memory 2. We say that the MMU *translates* the address.

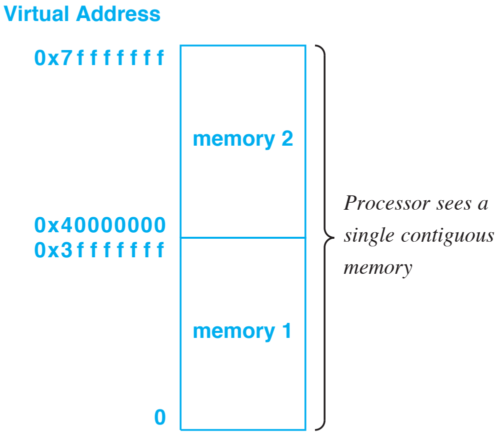


Figure 13.2 Illustration of a virtual memory system that divides an address space among two physical memories†. The MMU uses an address to decide which memory to access.

Mathematically, the address mapping for memory 2 is straightforward: the MMU merely subtracts 0x40000000 from an address. Figure 13.3 explains the concept.

```
Receive a virtual memory request from processor;
Let V be the address in the request;
if ( V < 0x40000000 ) {
        Pass the unmodified request (address V) to memory 1;
} else {     /* map the address for memory 2 */
        V2 = V − 0x40000000;
        Pass the modified request (address V2) to memory 2;
}
```

Figure 13.3 The sequence of steps used by a Memory Management Unit to create the virtual memory depicted in Figure 13.2. The MMU maps the virtual address space onto two physical memories.

The point is:

> *An MMU can combine multiple underlying physical memory systems to create a virtual address space that provides a processor with the illusion of a single, uniform memory system. Because each underlying memory uses addresses that start at zero, the MMU must translate between the addresses generated by the processor and the addresses used by each memory.*

†We have chosen to label an address space with address zero at the bottom; some documentation uses the convention of placing zero at the top of the address space.

13.6 Avoiding Arithmetic Calculation

In practice, an MMU does not use subtraction to implement address translation because subtraction requires substantial hardware (e.g., an ALU) and takes too much time to perform for each memory reference. The solution consists of using powers of two to simplify the hardware. For example, consider the mapping in Figure 13.2. Addresses 0 through 0x3fffffff map to memory 1, and addresses 0x40000000 through 0x7fffffff onto memory 2. Figure 13.4 shows that when expressed in binary, the addresses occupy thirty-one bits, and the ranges differ only in the high-order bit.

Addresses	Values In Binary (31 bits)
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
to	to
0x3fffffff	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0x40000000	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
to	to
0x7fffffff	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Figure 13.4 The binary values for addresses in the range 0 through 2 gigabytes. Except for the high-order bit, values above 1 gigabyte are the same as those below.

As the example shows, choosing a power of two can eliminate the need for subtraction because low-order bits can be used as a physical address. In the example, when mapping an address to one of the two underlying physical memories, an MMU can use the high-order bit of an address to determine which physical memory should receive the request. To form a physical address, the MMU merely extracts the remaining bits of the virtual address.

To summarize:

> *Dividing a virtual address space on a boundary that corresponds to a power of two allows the MMU to choose a physical memory and perform the necessary address translation without requiring arithmetic operations.*

13.7 Discontiguous Address Spaces

Figure 13.2 shows an example of a *contiguous virtual address space*, an address space in which all addresses are mapped onto an underlying physical memory. That is, the processor can reference any address from zero to the highest address because each address corresponds to a location in one of the physical memories. Interestingly, most computers are designed to be flexible — the physical memory is designed to allow the computer's owner to determine how much memory to install. The computer contains physical slots for memory, and the owner can choose to populate all the slots with memory chips or leave some of the slots empty.

Consider the consequence of allowing an owner to install an arbitrary amount of memory. Because it is defined when the computer is created, the virtual address space includes an address for each possible physical memory location (i.e., addresses for the maximum amount of memory that can be installed in the computer). If an owner decides to omit some of the memory, part of the virtual address space becomes unusable — if the processor references an address that does not correspond to physical memory, an error results. The virtual address space is not contiguous because regions of valid addresses are separated by invalid addresses. For example, Figure 13.5 shows how a virtual address space might appear if the virtual address space is mapped onto two physical memories, and part of each physical memory is omitted.

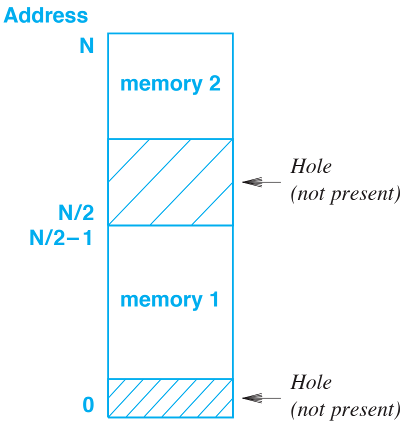


Figure 13.5 Example of a noncontiguous virtual address space of N bytes that is mapped onto two physical memories. Some addresses do not correspond to physical memory.

When part of a virtual address space does not map onto physical memory, we say that the address space contains a *hole*. In Figure 13.5, for example, the virtual address space contains two holes†.

We can summarize:

A virtual address space can be contiguous, in which case every address maps to a location of an underlying physical memory, or noncontiguous, in which case the address space contains one or more holes. If a processor attempts to read or write any address that does not correspond to physical memory, an error results.

†We will see further examples of address spaces that contain holes when we discuss I/O.

Many other possibilities exist for mapping a virtual address space onto physical memories. For example, recall from Chapter 11 that the two low-order bits of an address can be used to interleave memory among four separate physical memory modules (i.e., banks), and the remaining bits of the address can be used to select a byte within a module. One of the chief advantages of interleaving bytes among a set of modules arises from the ability of underlying hardware to access separate physical memories simultaneously. Using low-order bits to select a module means that successive bytes of memory come from different modules. In particular, if a processor accesses a data item composed of thirty-two bits, the underlying memory system can fetch all four bytes simultaneously.

13.8 Motivations For Virtual Memory

The trivial examples above show that a memory system can present a processor with a virtual address space that differs from the underlying physical memory. The rest of the chapter explores more complex virtual memory schemes. In most cases, the schemes incorporate and extend the concepts discussed above. We will learn that there are four main motivations for the use of complex virtual memory:

- Homogeneous integration of hardware
- Programming convenience
- Support for multiprogramming
- Protection of programs and data

Homogeneous Integration Of Hardware. Our examples explain how a virtual memory system can provide a homogeneous interface to a set of physical memories. More important, the scheme allows *heterogeneity* in the underlying memories. For example, some of the underlying physical memories can use a word size of thirty-two bits, while others use a word size of sixty-four bits. Some of the memories can have a much faster cycle time than others, or some of the memories can consist of RAM while others consist of ROM. The MMU hides the differences by allowing the processor to access all memories from a single address space.

Programming Convenience. One of the chief advantages of a virtual memory system arises from the ease of programming. If separate physical memories are not integrated into a uniform address space, a processor needs special instructions (or special operand formats) for each memory. Programming memory accesses becomes painful. More important, if a programmer decides to move an item from one memory to another, the program must be rewritten, which means that the decision cannot be made at run time.

Support For Multiprogramming. Modern computer systems allow multiple applications to run at the same time. For example, a user who is editing a document can leave a word processor open, temporarily launch a Web browser to check a reference, and listen to music at the same time. The terms *multiprogramming* and *multiprocessing* each characterize a computer system that allows multiple programs to run at the same time. We will see that a virtual memory system is needed to support multiprogramming.

Protection Of Programs And Data. We said that a CPU uses *modes of execution* to determine which instructions are allowed at any time. We will see that virtual memory is inherently linked to a computer's protection scheme.

13.9 Multiple Virtual Spaces And Multiprogramming

Early computer designers thought that multiprogramming was impractical. To understand why, consider how an instruction set works. The operands that specify indirection each reference a memory address. If two programs are loaded into a single memory and run at the same time, a conflict can occur if the programs attempt to use the same memory location for two different purposes. Thus, programs can only run together if they are written to avoid using the same memory addresses.

The most common technology for multiprogramming uses virtual memory to establish a separate virtual address space for each program. To understand how a virtual memory system can be used, consider an example. Figure 13.6 illustrates a straightforward mapping.

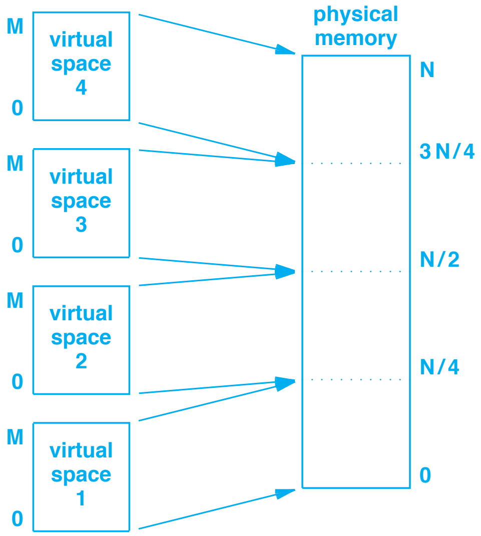


Figure 13.6 Illustration of four partitions mapped onto a single physical memory. Each virtual address space starts at address zero.

The mechanism in the figure divides the physical memory into equal-size areas that are known as *partitions*. Partitioned memory systems were used on early mainframe computers in the 1960s and 1970s, but have since been replaced. One of the main drawbacks of partitioned memory is that the memory available to a given program is a fraction of total physical memory on the computer. As Figure 13.6 illustrates, systems that used partitioned memory typically divided memory into four partitions, which meant that one-fourth of the total memory was dedicated to each program.

The diagram in Figure 13.6 implies that an MMU translates multiple virtual address spaces onto a single physical memory. In practice, however, MMU hardware can perform additional mappings. For example, an MMU can translate from virtual address space 1 to an intermediate virtual address space, and then translate the intermediate virtual address space onto one or more underlying physical memories (which may implement further translation from byte addresses to word addresses).

13.10 Creating Virtual Spaces Dynamically

How should a virtual memory system be created? In the simplistic examples above, we implied that the mapping from virtual address space(s) to physical memories is chosen when the hardware is built. Although some small, special-purpose systems have the mappings designed into hardware, general-purpose computer systems usually do not. Instead, the MMU in a general-purpose system can be changed dynamically at run time. That is, when the system boots, the processor tells the MMU exactly how to map the virtual address space onto the physical memory.

How can a program running on a processor change the address space and continue to run? In general, the address space to be used is part of the *processor mode*. The processor begins running in *real mode*, which means that the processor passes all memory references directly to the physical memory without using the MMU. While operating in real mode, the processor can interact with the MMU to establish a mapping. Once a mapping has been specified, the processor can execute an instruction that changes the mode, enables the MMU, and branches to a specified location. The MMU translates each memory reference according to the mapping that was configured.

The next sections examine technologies that have been used to create dynamic virtual memory systems. We will consider three examples:

- Base-Bound Registers
- Segmentation
- Demand Paging

13.11 Base-Bound Registers

A mechanism known by the name *base-bound* is among the oldest and easiest dynamic virtual memory schemes to understand. In essence, the *base-bound* scheme creates a single virtual address space and maps the space onto a region of physical memory. The name refers to a pair of registers that are part of the MMU; both must be loaded before the MMU is enabled. The base register holds an address in physical memory that specifies where to map the virtual address space, and the bound register holds an integer that specifies the size of the address space. Figure 13.7 illustrates the mapping.

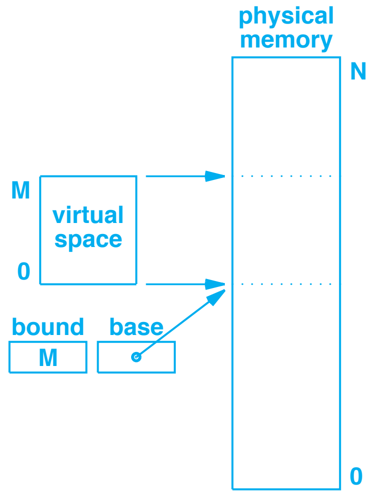


Figure 13.7 Illustration of a virtual memory that uses a base-bound mechanism. The base register specifies the location of the virtual address space, and the bound register specifies the size.

13.12 Changing The Virtual Space

It may seem that a base-bound mechanism is uninteresting because it only provides a single virtual address space. We must remember, however, that a base-bound mechanism is dynamic (i.e., easy to change). The idea is that an operating system can use the base-bound mechanism to move among multiple virtual address spaces. For example, suppose the operating system has loaded two application programs at different locations in memory. The operating system, which runs in real mode, controls the MMU. When an application, A, is ready to run, the operating system configures the MMU to point to A's section of the memory, enables the MMU mapping, and branches to the application. Later, when control returns to the operating system, the operating system selects another application to run, B, configures the MMU to point to B's memory, enables the MMU, and branches to the code for B. Each application's virtual address space starts at zero; the application remains unaware of its location in physical memory.

The point is that an operating system can use a base-bound mechanism to provide as much functionality as the static virtual memory mechanisms considered earlier. We can summarize:

> *A base-bound mechanism uses two values in the MMU to specify how a virtual address space maps onto the physical address space. The base-bound mechanism is powerful because an operating system can change the mapping dynamically.*

13.13 Virtual Memory And Protection

Why is a bound register used in the base-bound approach? The answer is *protection*: although a base register is sufficient to establish the mapping from virtual address to physical address, the mapping does not prevent a program from accidentally or maliciously referencing arbitrarily large memory locations. In Figure 13.7, for example, addresses higher than *M* lie beyond the region allocated to the program (i.e., the addresses may be allocated to another application).

The base-bound scheme uses the bound register to guarantee that a program will not exceed its allocated space. Of course, to implement protection, the MMU must check each memory reference and raise an error if the program attempts to reference an address greater than *M*. The protection offered by a base-bound mechanism provides an example of an important concept:

> *A virtual memory system that supports multiprogramming must also provide protection that prevents one application from reading or altering memory that has been allocated to another application.*

13.14 Segmentation

The memory mappings described above are intended to map a complete address space (i.e., all memory that is needed for an application to run, including the compiled program and the data the program uses). We say that a virtual memory technology that maps an entire address space is a *coarse granularity mapping*. The alternative, which consists of mapping parts of an address space, is known as a *fine granularity mapping*.

To understand the motivation for a fine granularity mapping, consider a typical application program. The program consists of a set of functions, and flow passes from one function to another through a procedure call. Early computer architects observed that although memory was a scarce resource, coarse granularity virtual systems required an entire application to occupy memory. Most of the memory was unused because only one function was actively executing at any time.

To reduce the amount of memory needed, the architects proposed that each program be divided into variable-size blocks, and only the blocks of the program that are needed at any time be loaded in memory. That is, pieces of the program are kept on an external storage device, typically a disk, until one of them is needed. At that time, the operating system finds an unused region of memory that is large enough, and loads the piece into memory. The operating system then configures the MMU to establish the mapping between the virtual addresses that the piece uses and the physical addresses used to hold the piece. When a program piece is no longer used, the operating system copies the piece back to disk, thereby making the memory available for another piece.

The variable-size piece scheme is known as *segmentation*, and the pieces of programs are known as *segments*. Once proposed, segmentation generated many questions. What hardware support would be needed to make segmentation efficient? Should the hardware dictate an upper bound on the size of a segment?

After much research and a few hardware experiments, segmentation faded. The central problem with segmentation arises after an operating system begins to move blocks in and out of memory. Because segments are variable size, the memory tends toward a situation in which the unused memory is divided into many small blocks. Computer scientists use the term *fragmentation* to describe the situation, and say that the memory becomes *fragmented*†. We can summarize:

> *Segmentation refers to a virtual memory scheme in which programs are divided into variable-size blocks, and only the blocks currently needed are kept in memory. Because it leads to a problem known as memory fragmentation, segmentation is seldom used.*

13.15 Demand Paging

An alternative to segmentation was invented that has become extremely successful. Known as *demand paging*, the technique follows the same general scheme as segmentation: divide a program into pieces, keep the pieces on external storage until they are needed, and load an individual piece when the piece is referenced.

The most significant difference between demand paging and segmentation lies in how the program is divided. Instead of variable-size segments that are large enough to hold a complete function, demand paging uses fixed-size blocks called *pages*. Initially, when memories and application programs were much smaller, architects chose a page size of 512 bytes or 1 Kbyte; current architectures use larger page sizes (e.g., Intel processors use 4 Kbyte pages).

13.16 Hardware And Software For Demand Paging

A combination of two technologies is needed for an effective virtual memory system that supports demand paging:

- Hardware to handle address mapping efficiently, record when each page is used, and detect missing pages
- Software to configure the hardware, monitor page use, and move pages between external store and physical memory

†To avoid memory fragmentation, some architects experimented with larger, fixed-size segments (e.g., 64 Kbytes per segment).

Demand Paging Hardware. Technically, the hardware architecture provides an address mapping mechanism and allows software to handle the demand aspect. That is, software (usually an operating system) configures the MMU to specify which pages from a virtual address space are present in memory and where each page is located. Then, the operating system runs an application that uses the virtual address space that has been configured. The MMU translates each memory address until the application references an address that is not available (i.e., an address on one of the pages that is not present in memory).

A reference to a missing page is called a *page fault*, and is treated as an error condition (e.g., like a division by zero). That is, instead of fetching the missing page from external storage, the hardware merely informs the operating system that a fault has occurred and allows the operating system to handle the problem. Typically the hardware is arranged to raise an *exception*. The hardware saves the current state of the computation (including the address of the instruction that caused the fault), and then uses the exception vector. Thus, from the operating system's point of view, a page fault acts like an interrupt. Once the fault has been handled, the operating system can instruct the processor to restart execution at the instruction that caused the fault.

Demand Paging Software. Software in the operating system is responsible for management of the memory: software must decide which pages to keep in memory and which to keep on external storage. More important, the software fetches pages *on demand*. That is, once the hardware reports a page fault, paging software takes over. The software identifies the page that is needed, locates the page on secondary storage, locates a slot in memory, reads the page into memory, and reconfigures the MMU. Once the page has been loaded, the software resumes executing the application, and the fetch-execute cycle continues until another page fault occurs.

Of course, the paging hardware and software must work together. For example, when a page fault occurs, the hardware must save the state of the computation in such a way that the values can be reloaded later when execution resumes. Similarly, the software must understand exactly how to configure the MMU.

13.17 Page Replacement

To understand paging, we must consider what happens after a set of applications has been running a long time. As applications reference pages, the virtual memory system moves the pages into memory. Eventually, the memory becomes full. The operating system knows when a page is needed because the application references the page. A difficult decision involves selecting one of the existing pages to evict to make space for an incoming page. Moving a page between external storage and memory takes time, so performance is optimized by choosing to move a page that will not be needed in the near future. The process is known as *page replacement*.

Because page replacement is handled by software, the discussion of algorithms and heuristics is beyond the scope of this text. We will see, however, that the hardware provides mechanisms that assist the operating system in making a decision.

13.18 Paging Terminology And Data Structures

The term *page* refers to a block of a program's address space, and the term *frame* refers to a slot in physical memory that can hold a page. Thus, we say that software *loads* a page into a frame of memory. When a page is in memory, we say that the page is *resident*, and the set of all pages from an address space that are currently in memory is called the *resident set*.

The primary data structure used for demand paging is known as a *page table*. The easiest way to envision a page table is to imagine a one-dimensional array that is indexed by a page number. That is, entries in the table have index zero, one, and so on. Each entry in the page table either contains a null value (if the page is not resident) or the address of the frame in physical memory that currently holds the page. Figure 13.8 illustrates a page table.

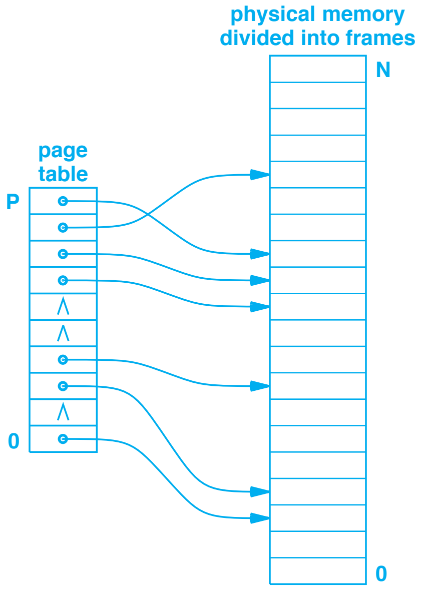


Figure 13.8 Illustration of an active page table with some entries pointing to frames in memory. A null pointer in a page table entry (denoted by Λ) means the page is not currently resident in memory.

13.19 Address Translation In A Paging System

Items in Figure 13.8 correspond to frames, not individual words. To understand paging hardware, imagine an address space divided into fixed-size pages as Figure 13.9 illustrates.

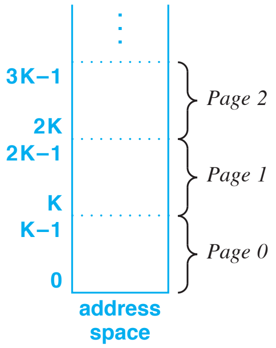


Figure 13.9 Illustration of a virtual address space divided into pages of K bytes per page.

We will see that the addresses associated with each page are important. As the figure shows, if each page contains K bytes, bytes on page zero have addresses zero through K–1, bytes on page 1 have addresses K through 2K–1, and so on.

Conceptually, translation of a virtual address, *V*, to a corresponding physical address, *P*, requires three steps:

1. Determine the number of the page on which address *V* lies.
2. Use the page number as an index into the page table to find the location of the frame in memory that holds the page.
3. Determine how far into the page *V* lies, and move that far into the frame in memory.

Figure 13.9 illustrates how addresses are associated with pages. Mathematically, the page number on which an address lies, *N*, can be computed by dividing the address by the number of bytes per page, *K*:

$$page\_number = N = \left\lfloor \frac{V}{K} \right\rfloor \qquad (13.1)$$

Similarly, the offset of the address within the page, *O*, can be computed as the remainder of the division†.

$$offset = O = V \ modulo \ K \qquad (13.2)$$

Thus, a virtual address, *V*, is translated to a corresponding physical address, *P*, by using the page number and offset, *N* and *O*, as follows:

$$physical\_address = P = pagetable\,[N] + O \qquad (13.3)$$

†Note that the computation of a byte address within a page is similar to the computation of a byte address within a word discussed on page 212.

13.20 Using Powers Of Two

As Chapter 11 discusses, an arithmetic operation, such as division, is too expensive to perform on each memory reference. Therefore, like other parts of a memory system, a paging system is designed to avoid arithmetic computation. The number of bytes per page is chosen to be a power of two, 2^q, which means that the address of the first byte in each frame has q low-order bits equal to zero. Interestingly, because the low-order bits of a frame address are always zero, a page table does not need to store a full address. The consequence of using a power of two is that the division and modulo operations specified in the mathematical equations can be replaced by extracting bits. Furthermore, the addition operation can be replaced by a *logical or*. As a result, instead of using Equations 13.1 through 13.3, the MMU performs the following computation to translate a virtual address, *V*, into a physical address, *P*:

$$P \;=\; pagetable\,[\,high\_order\_bits\,(V)\,] \quad or \quad low\_order\_bits\,(V) \tag{13.4}$$

Figure 13.10 illustrates how MMU hardware performs a virtual address mapping. When considering the figure, remember that hardware can move bits in parallel. Thus, the arrow that points from the low-order bits in the virtual address to the low-order bits in the physical address represents a parallel data path — the hardware sends all the bits at the same time. Also, the arrow from the page table entry to the high-order bits in the physical address means that all bits from the page table entry can be transferred in parallel.

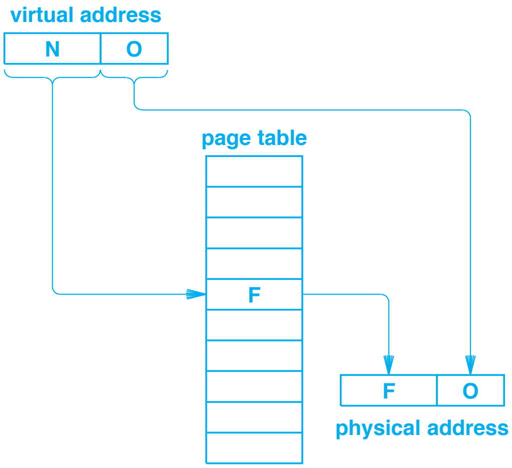


Figure 13.10 Illustration of how an MMU performs address translation on a paging system. Making the page size a power of two eliminates the need for division and remainder computation.

13.21 Presence, Use, And Modified Bits

Our description of paging hardware omits several details. For example, in addition to a value that specifies the frame in which a page is located, each page table entry contains control bits that the hardware and software use to coordinate. Figure 13.11 lists three control bits found in most paging hardware.

Control Bit	Meaning
Presence bit	Tested by hardware to determine whether page is currently resident in memory
Use bit	Set by hardware whenever page is referenced
Modified bit	Set by hardware whenever page is changed

Figure 13.11 Examples of control bits found in each page table entry and the actions hardware takes with each. The bits are intended to assist the page replacement software in the operating system.

Presence Bit. The most straightforward control bit is called a *presence bit*, which specifies whether the page is currently in memory. The bit is set by software and tested by the hardware. Once it has loaded a page and filled in other values in the page table entry, the operating system sets the presence bit to one; when it removes a page from memory, the operating system sets the presence bit to zero. When it translates an address, the MMU examines the presence bit in the page table entry — if the presence bit is one, translation proceeds, and if the presence bit is zero, the hardware declares a page fault has occurred.

Use Bit. The *use bit*, which provides information needed for page replacement, is initialized to zero and later tested by software. The bit is set by hardware. The mechanism is straightforward: whenever it accesses a page table entry, the MMU sets the use bit to one. The operating system periodically sweeps through the page table, testing the use bit to determine whether the page has been referenced since the last sweep. A page that has not been referenced becomes a candidate for eviction; otherwise, the operating system clears the use bit and leaves the page for the next sweep.

Modified Bit. The *modified bit* is initialized and later tested by software. The bit is set by hardware. The paging software sets the bit to zero when a page is loaded. The MMU sets the bit to one whenever a *write* operation occurs to the page. Thus, the modified bit is one if any byte on the page has been written since the page was loaded. The value is used during page replacement — if a page is selected for eviction, the value of the modified bit tells the operating system whether the page must be written back to external storage or can be discarded (i.e., whether the page is identical to the copy on external storage).

13.22 Page Table Storage

Where do page tables reside? Some systems store page tables in a special MMU chip that is external from the processor. Of course, because memory references play an essential role in processing, the MMU must be designed to work efficiently. In particular, to ensure that memory references do not become a bottleneck, some processors use a special-purpose, high-speed hardware interface to access an MMU. The interface contains parallel wires that allow the processor and MMU to send many bits at the same time.

Surprisingly, many processors are designed to store page tables in memory! That is, the processor (or the MMU) contains a special purpose register that the operating system uses to specify the location of the current page table. The location of a page table must be specified by giving a physical address. Typically, such systems are designed to divide memory into three regions as Figure 13.12 illustrates.

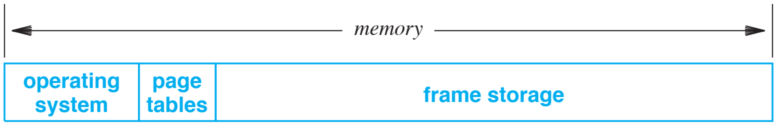


Figure 13.12 Illustration of how physical memory might be divided in an architecture that stores page tables in memory. A large area of physical memory is reserved for frames.

The design in the figure illustrates one of the motivations for a memory system composed of heterogeneous technologies: because page tables are used frequently, the memory used to store page tables needs high performance (e.g., SRAM). However, because high performance memory is expensive, overall cost can be reduced by using a lower-cost memory (e.g., DRAM) to store frames. Thus, an architect can design a system that uses SRAM to hold page tables and DRAM for frame storage.

13.23 Paging Efficiency And A Translation Lookaside Buffer

A central question underlies all virtual memory architectures: how efficient is the resulting system? To understand the question, it is important to realize that address translation must be performed on *every* memory reference: each instruction fetch, each operand that references memory, and each store of a result. Because memory is so heavily used, the mechanisms that implement address translation must be extremely efficient or translation will become a bottleneck. Architects are primarily concerned with the amount of time the MMU uses to translate a virtual address to a physical address; they are less concerned with the amount of time it takes for the operating system to configure page tables.

One technique used to optimize the performance of a demand paging system stands out as especially important. The technique uses special, high-speed hardware known as a *Translation Lookaside Buffer* (*TLB*) to achieve faster page table lookups. A TLB is a form of Content Addressable Memory that stores recently used values from a page table. When it first translates an address, the MMU places a copy of the page table entry in the TLB. On successive lookups, the hardware performs two operations in parallel: the standard address translation steps depicted in Figure 13.10 and a high-speed search of the TLB. If the requested information is found in the TLB, the MMU aborts the standard translation and uses the information from the TLB. If the entry is not in TLB, the standard translation proceeds.

To understand why a TLB improves performance, consider the fetch-execute cycle. A processor tends to fetch instructions from successive locations in memory. Furthermore, if the program contains a branch, probability is extremely high that the destination will be nearby, probably on the same page. Thus, rather than randomly accessing pages, a processor tends to fetch successive instructions from the same page. A TLB improves performance because it optimizes successive lookups by avoiding indexing into a page table. The difference in performance is especially dramatic for architectures that store page tables in memory; without a TLB, such systems are too slow to be useful. We can summarize:

> *A special high-speed hardware device, called a Translation Lookaside Buffer (TLB), is used to optimize performance of a paging system. A virtual memory that does not have a TLB can be unacceptably slow.*

13.24 Consequences For Programmers

Experience has shown that demand paging works well for most computer programs. The code that programmers produce tends to be organized into functions that each fit onto a page. Similarly, data objects, such as character strings, are designed so the data occupies consecutive memory locations, which means that once a page has been loaded, the page tends to remain resident for multiple references. Finally, some compilers understand paging, and optimize performance by placing data items onto pages.

One way that programmers can affect virtual memory performance arises from array access. Consider a two-dimensional array in memory. Most programming systems allocate an array in *row-major order*, which means that rows of an array are placed in contiguous memory as Figure 13.13 illustrates.

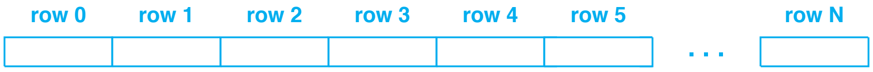


Figure 13.13 An illustration of a two-dimensional array stored in row-major order. A row is contiguous in memory.

As the figure shows, rows of the matrix occupy successive locations in memory. Thus, if A is a two-dimensional array of bytes, the location of A[i, j] is given by:

$$\text{location(A)} + i \times Q + j$$

where Q is the number of bytes per row.

The chief alternative to row-major order is known as *column-major order*. When an array is stored in column-major order, the elements of a column occupy contiguous memory locations. The choice between row-major or column-major order is usually determined by the programming language and compiler, not by a programmer.

A programmer can control how a program iterates through an array, and a good choice can optimize virtual memory performance. For example, if a large array of characters, A[N,M], is stored in row-major order, the nested loops shown here:

```
for i = 1 to N {
        for j = 1 to M {
                A [ i, j ] = 0;
        }
}
```

will require less time to execute than a loop that varies the indices in the opposite order:

```
for j = 1 to M {
        for i = 1 to N {
                A [ i, j ] = 0;
        }
}
```

The difference in time arises because varying the row index will force the virtual memory system to move from one page of memory to another for each reference, but varying the column index means M successive references stay on the same page.

13.25 The Relationship Between Virtual Memory And Caching

Two of the key technologies in virtual memory systems are related to caching: a TLB and the demand page relacement. Recall that a TLB consists of a small, high-speed hardware mechanism that improves the performance of a demand paging system dramatically. In fact, a TLB is nothing more than a cache of address mappings: whenever it looks up a page table entry, the MMU stores the entry in the TLB. A successive lookup for the same page will receive an answer from the TLB.

Like many cache systems, a TLB usually uses the Least Recently Used replacement strategy. Conceptually, when an entry is referenced, the TLB moves the entry to the front of the list; when a new reference occurs and the cache is full, the TLB discards the page table entry on the back of the list to make space for the new entry. Of course, the TLB cannot afford to keep a linked list in memory. Instead, the TLB contains digital circuits that move values into a special-purpose *Content Addressable Memory* (*CAM*) at high speed.

Demand paging can be viewed as a form of caching. The cache corresponds to main memory, and the data store corresponds to the external storage where pages are kept until needed. Furthermore, the page replacement policy serves as a cache replacement policy. In face, paging borrows the phrase *replacement policy* from caching.

Interestingly, thinking of demand paging as a cache can help us understand an important concept: how a virtual address space can be much larger than physical memory. Like a cache, physical memory only holds a fraction of the total pages. From our analysis of caching, we know that the performance of a demand-paged virtual memory can approach the performance of physical memory. In other words:

> *The analysis of caching in the previous chapter shows that using demand paging on a computer system with a small physical memory can perform almost as well as if the computer had a physical memory large enough for the entire virtual address space.*

13.26 Virtual Memory Caching And Cache Flush

If caching is used with virtual memory, should the cache be placed between the processor and the MMU or between the MMU and physical memory? That is, should the memory cache store pairs of virtual address and contents or physical address and contents? The answer is complex. On the one hand, using virtual addresses increases memory access speed because the cache can respond before the MMU translates the virtual address into a physical address. On the other hand, a cache that uses virtual addresses needs extra hardware that allows the cache to interact with the virtual memory system. To understand why, observe that a virtual memory system usually supplies the same address range to each running application (i.e., each process has addresses that start at zero). Now consider what happens when the operating system performs a context switch that stops running one process and runs another process. Suppose the memory cache contains an entry for address 2000 before the switch occurs. If the cache is unchanged during the context switch and the new process accesses location 2000, the cache will return the value from location 2000 in the old process. Therefore, when it changes from one process to another, the operating system must also change items in the cache.

How can a cache be engineered to avoid the problem of ambiguity that occurs when multiple processes use the same range of addresses? Architects use two solutions:

- A cache flush operation
- A disambiguating identifier

Cache Flush. One way to ensure that a cache does not report incorrect values consists of removing all existing entries from the cache. We say that the cache is *flushed*. In architectures that use flushing, the operating system must flush the cache whenever it performs a context switch to move from one application to another.

Disambiguation. The alternative to cache flushing involves the use of extra bits that identify the running process (or more precisely, the address space). The processor contains an extra hardware register that contains an address space ID. Many operating systems create an address space for each associated process, and use the process ID (an integer) to identify an address space. Whenever it switches to an application, the operating system loads the application's process ID into the address space ID register. As Figure 13.14 shows, the cache prepends the contents of the ID register onto the virtual address when it stores an item in the cache, which means that even if process 1 and process 2 both reference address 0, the two entries in the cache will differ.

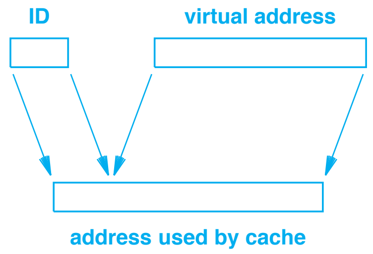


Figure 13.14 Illustration of an ID register used to disambiguate among a set of virtual address spaces. Each address space is assigned a unique number, which the operating system loads into the ID register.

As the figure illustrates, the cache is designed to use a longer address than the memory system. Before passing a request to the cache, the processor creates an artificially long address by concatenating a virtual address onto the process ID. The processor then passes the longer address to the cache. From the cache's point of view, there is no ambiguity: even if two applications reference the same virtual address, the ID bits distinguish between the two addresses.

13.27 Summary

Virtual memory systems present an abstract address space to a processor and to each application program running on the processor. A virtual memory system hides details of the underlying physical memory.

Several virtual memory architectures are possible. The virtual memory system can hide details of word addressing or can present a uniform address space that incorporates multiple, possibly heterogeneous, memory technologies.

Virtual memory offers convenience for programmers, support for multiprogramming, and protection. When multiple programs run concurrently, virtual memory can be used to provide each program with an address space that begins at zero.

Virtual memory technologies include base-bound, segmentation, and demand paging; demand paging is the most popular. A demand paging system uses page tables to map from a virtual address to a physical address; a high-speed search mechanism known as a TLB makes page table lookup efficient.

To avoid arithmetic computation, virtual memory systems make physical memory and page sizes a power of two. Doing so allows the hardware to translate addresses without using arithmetic or logical operations.

Either physical or virtual addresses can be cached. If a cache uses virtual addresses, an ambiguity problem can arise when multiple applications (processes) use the same range of virtual addresses. Two techniques can be used to solve the ambiguity problem: the operating system can flush the cache whenever it switches from one application to another, or the cache hardware can be designed to use artificially long addresses where the high-order bits consist of an address space ID (typically, a process ID).

EXERCISES

13.1 Consider a computer using the virtual address space illustrated in Figure 13.2. If a C programmer writes:

```
char   c;
char   *p;

p = (char *)1073741826;
c = *p;
```

Which memory module will be referenced, and where in the module will the referenced byte be located within the memory?

13.2 A traditional Intel PC has a *hole* in its memory address space between 640 kilobytes and 1 megabyte. Use Figure 13.5 as an example, and draw a figure to scale showing the hole in a PC address space if the PC has 2 megabytes of memory.

13.3 Which of the four motivations for virtual memory help programmers? Explain.

13.4 Does demand paging require special hardware or special software? Explain.

13.5 Conceptually, a page table is an array. What is found in each element of the page table array, and how is it interpreted?

13.6 Consider the presence, use, and modified bits. For each bit, tell when the bit changes and whether the hardware or software makes the change.

13.7 Assuming a page size of 4K bytes, compute the page number and the offset for addresses 100, 1500, 8800, and 10000.

13.8 Write a computer program that takes two input values, a page size and an address, and computes the page number and offset for the address.

13.9 Extend the program in the previous exercise. If the page size is a power of two, do not use division or modulus operations when computing the answer.

13.10 Calculate the amount of memory needed to hold an example page table. Assume that each page table entry occupies thirty-two bits, the page size is 4 Kbytes, and a memory address occupies thirty-two bits.

13.11 Write a computer program that takes as input a page size and an address space size, and performs the calculation in the previous exercise. (You may restrict sizes to powers of two.)

13.12 What is page replacement, and is it performed by hardware or software?

13.13 Consider a two-level paging scheme. Assume the high-order ten bits of an address are used as an index into a *directory table* to select a page table. Assume each page table contains 1024 entries and the next ten bits of the address are used to select a page table entry. Also assume the final twelve bits of the address are used to select a byte on the page. How much memory is required for the directory table and page tables?

13.14 What is a TLB, and why is it necessary?

13.15 Write a program that references all locations in a large two-dimensional array stored in row-major order. Compare the execution times when the program iterates through rows and touches each column within a row to the time required when the program iterates through all columns and touches each row within a column. Explain the results.

13.16 If a memory system caches virtual addresses and each process has a virtual address space that starts at zero, what must an operating system do when changing from one process to another? Why?

Part IV

Input And Output

External Connections
And Data Movement

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

14

Input / Output Concepts And Terminology

14.1 Introduction

Previous chapters of the text describe two of the major components found in a computer system: processors and memories. In addition to describing technologies used for each component, the chapters illustrate how processors and memory interact.

This chapter introduces the third major aspect of architecture, connections between a computer and the external world. We will learn that on most computers, the connection between a processor and an I/O device uses the same basic paradigm as the connection between a processor and memory. Furthermore, we will see that although they operate under control of a processor, I/O devices can interact directly with memory.

14.2 Input And Output Devices

The earliest electronic computers, which consisted of a numerical processor plus a memory, resembled a calculator more than a modern computer. The human interface was crude — values were entered through a set of manual switches, and results of calculations were viewed through a series of lights. By the late 1940s, it had become obvious that better interfaces were needed before digital computers could be useful for more than basic calculations. Engineers began devising ways to connect computers to external devices, which became known as *Input* and *Output* (*I/O*) devices. Modern I/O devices include cameras, earphones, and microphones, as well as keyboards, mice, monitors, sensors, hard disks, DVD drives and printers.

14.3 Control Of An External Device

The earliest external devices attached to computers consisted of independent units that operated under control of the CPU. That is, an external device usually occupied a separate physical cabinet, had an independent source of electrical power, and contained internal circuitry that was separate from the computer. The small set of wires that connected the computer to the external device, only carried control signals (i.e., signals from the digital logic in the computer to the digital logic in the device). Circuitry in the device monitored the control signals, and changed the device accordingly.

Many early computers provided a set of lights that displayed values. Typically, the display contained one light for each bit in the computer's accumulator — the light was on when the bit was set to one, and off when the bit was zero. However, it is not possible to connect a light bulb directly to the accumulator circuit because even a small light bulb requires more power than a digital circuit can deliver. Therefore, a display unit contains circuitry that receives a set of digital logic signals and controls a set of light bulbs accordingly. Figure 14.1 illustrates how the hardware is organized.

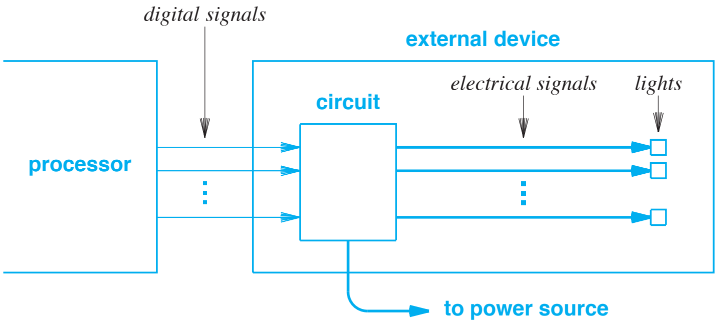


Figure 14.1 Example of an external circuit that controls a set of lights. The device contains circuitry that converts incoming digital logic signals into the signals needed to operate a set of light bulbs.

As the figure illustrates, we think of an external device as independent from the processor except for digital signals that pass between them. In practice, of course, some devices reside in the same enclosure with the processor, and both receive power from a common source. We will ignore such details and concentrate on the control signals.

A computer interacts with a device in two ways: the computer controls a device and the computer exchanges data with a device. For example, a processor can start a disk spinning, control the volume on an external speaker, tell a camera to snap a picture, or turn off a printer. In the next chapter, we will learn how a computer passes control information to external devices.

14.4 Data Transfer

Although control of external devices is essential, for most devices, control functions are secondary. The primary function of external devices is *data transfer*. Indeed, most of the architectural choices surrounding external devices focus on mechanisms that permit the device and processor to exchange data.

We will consider several questions regarding data transfer. First, exactly how is data communicated? Second, which side initiates transfer (i.e., does the processor request a transfer or does the device)? Third, what techniques and mechanisms are needed to enable high-speed transfers?

Other questions that are less pertinent to programmers concern low-level details. What voltages are used to communicate with an external device, and how is data represented? The answers depend on the type of device, the speed with which data must be transferred, the type of cabling used, and the distance between the processor and the device. However, as Figure 14.1 illustrates, the digital signals used internally by a processor are insufficient to drive circuits in an external device.

Because the voltages and encodings used for external connections differ from those used internally, special hardware is needed to translate between the two representations. We use the term *interface controller* to refer to the hardware that provides the interface to an external device. Figure 14.2 illustrates that interface controllers are needed at both ends of a physical connection.

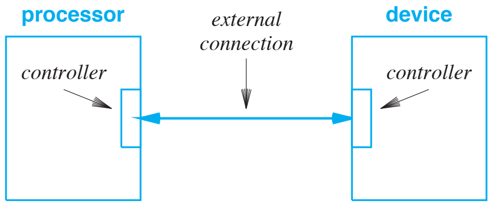


Figure 14.2 Illustration of controller hardware on each end of an external connection. The voltages and signals used on the external connection can differ from the voltages used internally.

14.5 Serial And Parallel Data Transfers

All the I/O interfaces on a computer can be classified in two broad categories:

- Parallel interface
- Serial interface

Parallel Interface. An interface between a computer and an external device is classified as *parallel* if the interface allows the transfer of multiple bits of data simultaneously. In essence, a parallel interface contains many wires — at any instant, each wire carries one bit of data.

We use the term *interface width* to refer to the number of parallel wires an interface uses. Thus, one might hear an engineer talk about an eight-bit interface or a sixteen-bit interface. We will learn more about how interfaces use parallel wires in the next chapter.

Serial Interface. The alternative to a parallel interface is one in which only one bit of data can be transferred at any time; an interface that transfers one bit at a time is classified as *serial*.

The chief advantages of a serial interface are fewer wires and less interference from signals traveling at the same time. In principle, only two wires are needed for serial data transmission — one to carry the signal and a second to serve as an electrical ground against which voltage can be measured. The chief disadvantage of a serial interface arises from increased latency: when sending multiple bits, serial hardware must wait until one bit has been sent before sending another.

14.6 Self-Clocking Data

Recall that digital circuits operate according to a *clock*, a signal that pulses continuously. Clocks are especially significant for I/O because each I/O device and processor can have a separate clock rate (i.e., each controller can have its own clock). Thus, one of the most significant aspects of an external interface concerns how the interface accommodates differences in clock rates.

The term *self-clocking* describes a mechanism in which signals sent across an interface contain information that allows the receiver to determine exactly how the sender encoded the data. For example, some external devices use a method similar to the clockless logic mechanism that Chapter 2 describes†. Others employ an extra set of wires that pass clocking information: when transmitting data, the sender uses the extra wires to inform the receiver about the location of bit boundaries in the data.

14.7 Full-Duplex And Half-Duplex Interaction

Many external I/O devices provide *bidirectional transfer* which means the processor can send data to the device or the device can send data to the processor. For example, a disk drive supports both *read* and *write* operations. Interface hardware uses two methods to accommodate bidirectional transfer:

- Full-duplex interaction
- Half-duplex interaction

Full-Duplex Interaction. An interface that allows transfer to proceed in both directions simultaneously is known as a *full-duplex* interface. In essence, full-duplex

†The description of clockless logic can be found on page 37.

hardware consists of two parallel devices with two independent sets of wires connecting them. One set is used to transfer data in each direction.

Half-Duplex Interaction. The alternative to a full-duplex interface, known as a *half-duplex* interface, only allows transfer to proceed in one direction at a time. That is, a single set of wires that connects the processor and the external device must be shared. In the next chapter, we will see that sharing requires negotiation — before it can perform a transfer, a processor or device must wait for the current transfer to finish, and must obtain exclusive use of the underlying wires.

14.8 Interface Throughput And Latency

The *throughput* of an interface is measured in the number of bits that can be transferred per unit time, and is usually measured in *Megabits per second* (*Mbps*) or *Megabytes per second* (*MBps*). It may seem that a serial interface would always have a lower throughput because serial transmission only transfers one bit at a time, whereas a parallel interface can transfer multiple bits at the same time. However, when parallel wires are close together, the data rate must be limited or electromagnetic interference can result. Therefore, in some cases, engineers have been able to send bits over a serial interface with higher throughput than a parallel interface.

The second major measure of an interface is *latency*. Latency refers to the delay between the time a bit is sent and the time the bit is received (i.e., how long it takes to transfer a single bit), which is usually measured in nanoseconds (ns). As we have seen with memories, we must be careful to distinguish between latency and throughput because some devices need low latency and others need high throughput.

We can summarize:

> *The latency of an interface is a measure of the time required to perform a transfer, the throughput of an interface is a measure of the data that can be transferred per unit time.*

14.9 The Fundamental Idea Of Multiplexing

It may seem that choosing an interface is trivial: we want full-duplex, low latency, and high throughput. Despite the desire for high performance, many other factors make the choice of interfaces complex. Recall, for example, each integrated circuit has a fixed number of *pins* that provide external connections. A wider interface uses more pins, which means fewer pins for other functions. An interface that provides full-duplex capability uses approximately twice as many pins as an interface that provides half-duplex capability.

Most architects choose a compromise for external connections. The connection has *limited parallelism*, and the hardware uses a technique known as *multiplexing* to send data. Although details are complex, the concept of multiplexing is easy to understand. The idea is that the hardware breaks a large data transfer into pieces and sends one piece at a time. We use the terms *multiplexor* and *demultiplexor* to describe the hardware that handles data multiplexing. For example, Figure 14.3 illustrates how multiplexing hardware can divide sixty-four bits of data into sixteen-bit chunks and transmit chunks over an interface that has a width of sixteen bits. Only one chunk can be sent at a given time.

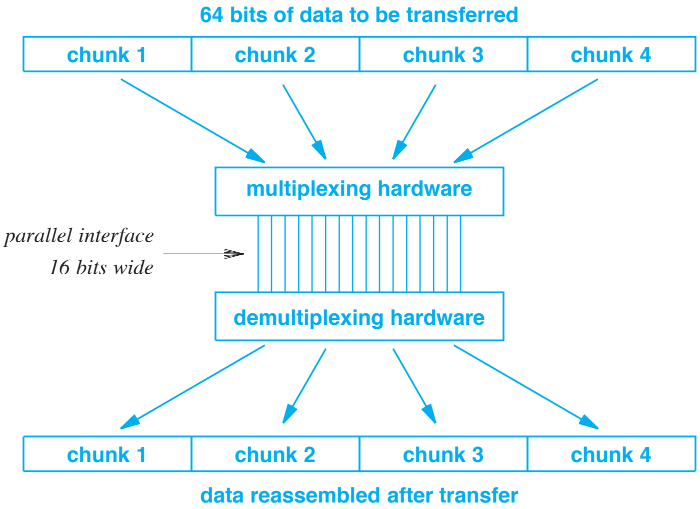


Figure 14.3 Illustration of the transfer of sixty-four bits of data over a sixteen-bit interface. Multiplexing hardware divides the data into sixteen-bit units and sends one unit at a time.

In practice, most physical connections between a processor and external devices use multiplexing. Doing so allows the processor to transfer arbitrary amounts of data without devoting many physical pins to the connection. In the next chapter, we will learn how multiplexing also improves CPU performance.

To summarize:

> *Multiplexing is used to construct an I/O interface that can transfer arbitrary amounts of data over a fixed number of parallel wires. Multiplexing hardware divides the data into blocks, and transfers each block independently.*

Note that our definition applies equally to serial transmission — we simply interpret a serial interface as multiplexing transfers over a single wire. Thus, the chunk size for a serial interface is a single bit.

14.10 Multiple Devices Per External Interface

The examples in this chapter imply that each external connection from a processor attaches to one device. To help conserve pins and external connections, most processors do not have a single device per external connection. Instead, a set of pins attaches to multiple devices, and the hardware is configured to permit the processor to communicate with one of the devices at a given time. The next chapter explains the concept in detail and gives examples.

14.11 A Processor's View Of I/O

Recall that interface controller hardware is associated with an external connection. Thus, when a processor interacts with an external device, the processor must do so through the controller. The processor makes requests to the controller, and receives replies; the controller translates each request into the appropriate external signals that perform the requested function on the external device. The point is that the processor can only interact with the interface controller and not with the external device.

To capture the architectural concept, we say that the controller presents a *programming interface* to the processor. Interestingly, the programming interface does not need to model the operations of the underlying device exactly. In the next chapter, we will see an example of a widely used programming interface that casts all external interactions into a simplified paradigm. To summarize:

> *A processor uses interface controller hardware to interact with a device; the controller translates requests into the appropriate external signals.*

14.12 Summary

Computer systems interact with external devices either to control the device (e.g., to change the status) or to transfer data. An external interface can use a serial or parallel approach; the number of bits that can be sent simultaneously is known as the *width* of a parallel interface. A bidirectional interface can use full-duplex or half-duplex interaction.

There are two measures of interface performance. Latency refers to the time required to send a bit from a given source to a given destination (e.g., from memory to a printer), and throughput refers to the number of bits that can be sent per unit time.

Because the number of pins is limited, a processor does not have arbitrarily wide external connections. Instead, interface hardware is designed to multiplex large data transfers over fewer pins. In addition, multiple external devices can attach to a single external connection; the interface controller hardware communicates with each device separately.

EXERCISES

14.1 The speaker in a smart phone or laptop is, in fact, an analog device in which the volume is proportional to the voltage supplied. Does that mean a processor must have an analog output for audio? Explain.

14.2 What are the primary and secondary functions associated with external devices?

14.3 Can a device that operates on 3.3-volt digital signals be connected to a processor that operates on 5-volt digital signals? Explain.

14.4 If the *interface width* is 16, is the interface parallel or serial? Explain.

14.5 USB is classified as a serial interface. What does the classification mean?

14.6 Suppose you are purchasing a network I/O device, and the vendor gives you the choice of a half-duplex or full-duplex interface. Which do you choose and why?

14.7 If the interface between a processor and storage device has a width of thirty-two bits, how can the processor transfer a data item that consists of sixty-four bits?

14.8 Create a parallel interface that is self-clocking and can send data from one side to the other. Hint: have two wires that the two ends use to coordinate and additional wires that are used to transfer data.

14.9 Suppose a serial interface has a latency of 200 microseconds. How long does it take to transfer one bit over the interface? How long does it take to transfer sixty-four bits over the interface?

14.10 Suppose a parallel interface has a width of thirty-two bits and a latency of 200 microseconds. How long does it take to transfer thirty-two bits over the interface? How long does it take to transfer sixty-four bits over the interface? Explain.

Chapter Contents

15

Buses And Bus Architectures

15.1 Introduction

The chapters on memory discuss the external connection between a processor and the memory system. The previous chapter discusses connections with external I/O devices, and shows how a processor uses the connections to control the device or transfer data. The chapter reviews concepts such as serial and parallel transfer, defines terminology, and introduces the idea of multiplexing data transfer over a set of wires.

This chapter extends the ideas by explaining a fundamental architectural feature present in all computer systems, a bus. It describes the motivation for using a bus, explains the basic operation, and shows how both memory and I/O devices can share a common bus. We will learn that a bus defines an address space and understand the relationship between a bus address space and a memory address space.

15.2 Definition Of A Bus

A *bus* is a digital communication mechanism that allows two or more functional units to transfer control signals or data. Most buses are designed for use inside a single computer system; some are used within a single integrated circuit. Many bus designs exist because a bus can be optimized for a specific purpose. For example, a *memory bus* is intended to interconnect a processor with a memory system, and an *I/O bus* is intended to interconnect a processor with a set of I/O devices. We will see that general-purpose designs are possible.

15.3 Processors, I/O Devices, And Buses

The notion of a bus is broad enough to encompass most external connections (i.e., a connection between a processor and a coprocessor). Thus, instead of viewing the connection between a processor and a device as a set of wires (as in Chapter 14), we can be more precise: the two units are interconnected by a bus. Figure 15.1 uses a graphic that is common in engineering diagrams to illustrate the concept.

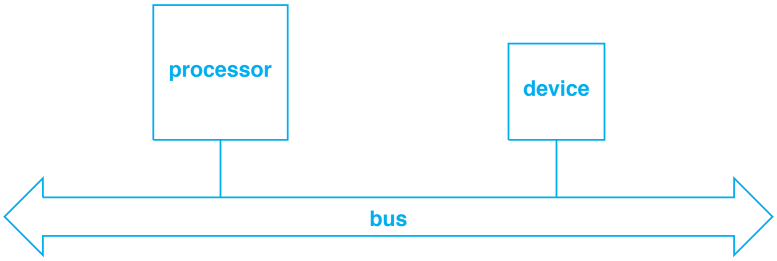


Figure 15.1 Illustration of a bus used to connect a processor and an external device. Buses are used for most external connections.

We can summarize:

> *A* bus *is the digital communication mechanism that interconnects functional units of a computer system. A computer contains one or more buses that interconnect the processors, memories, and external I/O devices.*

15.3.1 Proprietary And Standardized Buses

A bus design is said to be *proprietary* if the design is owned by a private company and not available for use by other companies (i.e., covered by a patent). The alternative to a proprietary bus is known as a *standardized bus*, which means the specifications are available. Because they permit equipment from two or more vendors to communicate and interoperate, standardized buses allow a computer system to contain devices from multiple vendors. Of course, a bus standard must specify all the details needed to construct hardware, including the exact electrical specifications (e.g., voltages), timing of signals, and the encoding used for data. Furthermore, to ensure correctness, each device that attaches to the bus must implement the bus standard precisely.

15.3.2 Shared Buses And An Access Protocol

We said that a bus can be used to connect a processor to an I/O device. In fact, most buses are *shared*, which means that a single bus is used to connect the processor to a set of I/O devices. Similarly, if a computer contains multiple processors, all the processors can connect to a shared bus.

To permit sharing, an architect must define an *access protocol* to be used on the bus. The access protocol specifies how an attached device can determine whether the bus is available or is in use, and how attached devices take turns using the bus.

15.3.3 Multiple Buses

A typical computer system contains multiple buses. For example, in addition to a central bus that connects the processor, I/O devices, and memory, some computers have a special-purpose bus used to access coprocessors. Other computers have multiple buses for convenience and flexibility — a computer with several standard buses can accommodate a wider variety of devices.

Interestingly, most computers also contain buses that are *internal* (i.e., not visible to the computer's owner). For example, many processors have one or more internal buses on the processor chip. A circuit on the chip uses an onboard bus to communicate with another circuit (e.g., with an onboard cache).

15.3.4 A Parallel, Passive Mechanism

As Chapter 14 describes, an interface is either classified as using *serial* data transfer or *parallel* data transfer. Most of the buses used in computer systems are parallel. That is, a bus is capable of transferring multiple bits of data at the same time.

The most straightforward buses are classified as *passive* because the bus itself does not contain electronic components. Instead, each device that attaches to a bus contains the electronic circuits needed to communicate over the bus. Thus, we can imagine a bus to consist of parallel wires to which devices attach†.

15.4 Physical Connections

Physically, a bus can consist of tiny wires etched in silicon on a single chip, a cable that contains multiple wires, or a set of parallel metal wires on a circuit board. Most computers use the third form for an I/O bus: the bus is implemented as a set of parallel wires on the computer's main circuit board, which is known as a *mother board*. In addition to a bus, the mother board contains the processor, memory, and other functional units.

A set of sockets on the mother board connects to the bus to allow devices to be plugged in or removed easily (i.e., a device can be connected to the bus merely by plugging the device into a socket). Typically, the bus and the sockets are positioned near

†In practice, some buses do contain a digital circuit known as a *bus arbiter* that coordinates devices attached to the bus. However, such details are beyond the scope of this text.

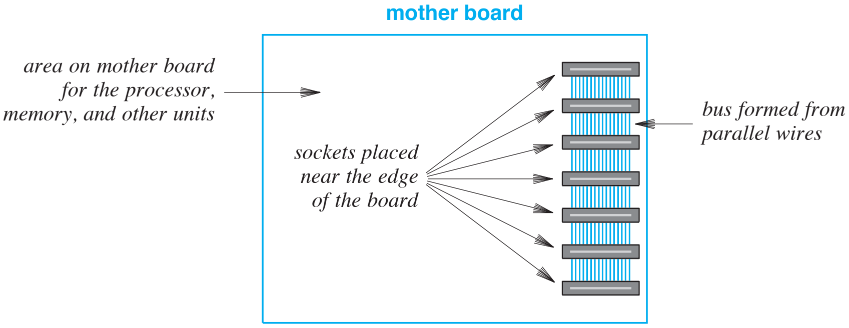

mother board
area on mother board for the processor, memory, and other units
sockets placed near the edge of the board
bus formed from parallel wires

Figure 15.2 Illustration of a bus that consists of parallel wires that connect to sockets on a mother board. The mother board contains other components that are not shown.

the edge of the mother board to make them easily accessible from outside. Figure 15.2 illustrates a bus and sockets on a mother board.

15.5 Bus Interface

Attaching a device to a bus is nontrivial. To operate correctly, a device must adhere to the bus standard. Recall, for example, that a bus is shared and that the bus specifies an *access protocol* that is used to determine when a given device can access the bus to transfer data. To implement the access protocol, each device must have a digital circuit that connects to the bus and follows the bus standard. Known as a *bus interface* or a *bus controller*, the circuit implements the bus access protocol and controls exactly when and how the device uses the bus. If the bus protocol is complicated, the interface circuit can be large; many bus interfaces require multiple chips.

What is the physical connection between a bus interface circuit and the bus itself? Interestingly, the sockets of many buses are chosen to make it possible to plug a printed circuit board directly into the socket. The circuit board must have a region cut to the exact size of a socket, and must have metal *fingers* that align exactly with metal contacts in the socket. Figure 15.3 illustrates the concept.

The figure helps us envision how a physical computer system is constructed. If the mother board lies in the bottom of a cabinet, the circuit boards for individual devices that plug into the mother board will be perpendicular, meaning that the device circuit boards will be vertical. A key piece of the physical arrangement concerns the placement of sockets — by locating sockets near the edge of a mother board, a designer can guarantee that device boards are located adjacent to the side of the cabinet. Choosing a location near the side means a short connection between the circuit board and the outside of the cabinet. The arrangement is used in a typical PC.

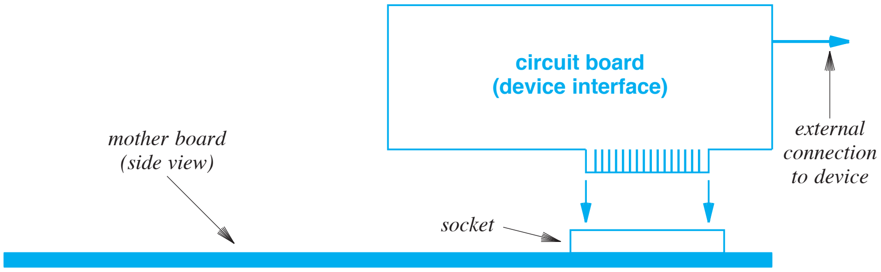


Figure 15.3 Side view of a mother board illustrating how a printed circuit board plugs into the socket of a bus. Metal fingers on the circuit board press against metal contacts in the socket.

15.6 Control, Address, And Data Lines

Although the physical structure of a bus provides interesting engineering challenges, we are more concerned with the logical structure. We will examine how the wires are used, the operations the bus supports, and the consequences for programmers.

Informally, the wires that comprise a bus are called *lines*. There are three conceptual functions for the lines:

- Control of the bus
- Specification of address information
- Transfer of data

To help us understand how a bus operates, we will assume that the bus contains three separate sets of lines that correspond to the three functions†. Figure 15.4 illustrates the concept.

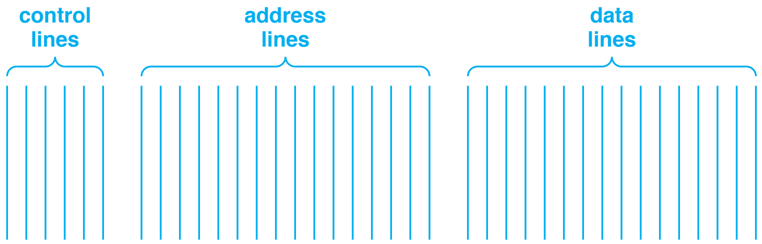


Figure 15.4 Conceptual division of wires that comprise a bus into lines for control, addresses, and data.

†The description here simplifies details; a later section explains how the functionality can be achieved without physically separate groups of wires.

As the figure implies, bus lines need not be divided equally among the three uses. In particular, control functions usually require fewer lines than other functions.

15.7 The Fetch-Store Paradigm

Recall from Chapter 10 that memory systems use the *fetch-store* paradigm in which a processor can either *fetch* (i.e., *read*) a value from memory or *store* (i.e., *write*) a value to memory. A bus uses the same basic paradigm. That is, a bus only supports *fetch* and *store* operations. As unlikely as it seems, we will learn that when a processor communicates with a device or transfers data across a bus, the communication always uses *fetch* or *store* operations. Interestingly, the fetch-store paradigm is used with all devices, including microphones, video cameras, sensors, and displays, as well as with storage devices, such as disks.

We will see later how it is possible to control all devices with the fetch-store paradigm. For now, it is sufficient to understand the following:

> *Like a memory system, a bus employs the fetch-store paradigm; all control or data transfer operations are performed as either a* fetch *or a* store.

15.8 Fetch-Store And Bus Size

Knowing that a bus uses the fetch-store paradigm helps us understand the purpose of the three conceptual categories of lines that Figure 15.4 illustrates. All three categories are used for either a *fetch* or *store* operation. Control lines are used to ensure that only one pair of entities attempts to communicate over the bus at any time, and to allow two communicating entities to interact meaningfully. The address lines are used to pass an address, and the data lines are used to transfer a data value.

Figure 15.5 explains how the three categories of lines are used during a *fetch* or *store* operation. The figure lists the steps that are taken for each operation, and specifies which group of lines is used for each step.

We said that most buses use parallel transfer — the bus contains multiple data lines, and can simultaneously transfer one bit over each data line. Thus, if a bus contains K data lines, the bus can transfer K bits at a time. Using the terminology from Chapter 14, we say that the bus has a *width* of K bits. Thus, a bus that has thirty-two data lines (i.e., can transfer thirty-two bits at the same time) is called a *thirty-two-bit bus*.

Of course, some buses are serial rather than parallel. A serial bus can only transfer one bit at a time. Technically, a serial bus has a width of one bit. However, engineers do not usually talk about a bus having a width of one bit; they simply call it a *serial bus*.

Fetch

1. Use the control lines to obtain access to the bus
2. Place an address on the address lines
3. Use the control lines to request a *fetch* operation
4. Test the control lines to wait for the operation to complete
5. Read the value from the data lines
6. Set the control lines to allow another device to use the bus

Store

1. Use control lines to obtain access to the bus
2. Place an address on the address lines
3. Place a value on the data lines
4. Use the control lines to specify a *store* operation
5. Test the control lines to wait for the operation to complete
6. Set the control lines to allow another device to use the bus

Figure 15.5 The steps taken to perform a *fetch* or *store* operation over a bus, and the group of lines used in each step.

15.9 Multiplexing

How wide should a bus be? Recall from Chapter 14 that parallel interfaces represent a compromise: although increasing the width increases the throughput, greater width also takes more space and requires more electronic components in the attached devices. Furthermore, at high data rates, signals on parallel wires can interfere with one another. Thus, an architect chooses a bus width as a compromise between space, cost, and performance.

One technique stands out as especially helpful in reducing the number of lines in a bus: *multiplexing*. A bus can use multiplexing in two ways: *data multiplexing* alone or a combination of *address and data multiplexing*.

Data Multiplexing. In Chapter 14, we learned how data multiplexing works. In essence, when a device attached to a bus has a large amount of data to transfer, the device divides the data into blocks that are exactly as large as the bus is wide. The device then uses the bus repeatedly, by sending one block at a time.

Address And Data Multiplexing. The motivation for multiplexing addresses is to reduce the number of lines. To understand how address multiplexing works, consider the steps in Figure 15.5 carefully. In the case of a *fetch* operation, the address lines and data lines are never used at the same time (i.e., in the same step). Thus, an architect can use the same lines to send an address and receive data. For a *store* operation, multiplexing can be used: bus hardware sends the address first and then sends the data†.

†Of course, a device that receives a request over a multiplexed bus must store the address while the data is transferred.

Most buses make heavy use of multiplexing. Thus, instead of three conceptual sets of lines, a typical bus has two: a few lines used for control, and a set of lines used to send either an address or data. Figure 15.6 illustrates the idea.

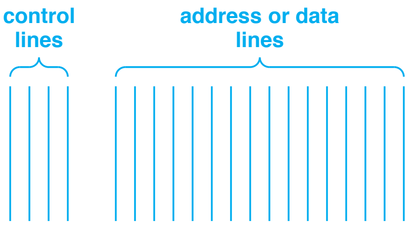


Figure 15.6 Illustration of a bus in which a single set of lines is used for both addresses and data. Using one set of lines helps reduce cost.

Multiplexing offers two advantages. On the one hand, multiplexing allows an architect to design a bus that has fewer lines. On the other hand, if the number of lines in a bus is fixed, multiplexing produces higher overall performance. To see why, consider a data transfer. If K of the lines in the bus are reserved for addresses, those K lines cannot be used during a data transfer. If all the lines are shared, however, an additional K bits can be transferred on each bus cycle, which means higher overall throughput.

Despite its advantages, multiplexing does have two disadvantages. First, multiplexing takes more time because a *store* operation requires two bus cycles (i.e., one to transfer the address and another to transfer the data item). Second, multiplexing requires a more sophisticated bus protocol, and therefore, more complex bus interface hardware. Despite the disadvantages, many bus designs use multiplexing. In the extreme case, a bus can be designed that multiplexes control information over the same set of lines used for data and addresses.

15.10 Bus Width And Size Of Data Items

The use of multiplexing helps explain another aspect of computer architecture: uniform size of all data objects, including addresses. We will see that all data transfers among a processor, memories, and devices occur over a bus. Furthermore, because the bus multiplexes the transfers over a fixed number of lines, a data item that exactly matches the bus width can be transferred in one cycle, but any item that is larger than the bus width requires multiple cycles. Thus, it makes sense for an architect to choose a single size for the bus width, the size of a general-purpose register, and the size of a data value that the ALU or functional units use (e.g., the size of an integer or a floating point value). More important, because addresses are also multiplexed over the bus lines, it makes sense for the architect to choose the same size for an address as for other data items. The point is:

> *In many computers, both addresses and data values are multiplexed over the same bus. To optimize performance of the hardware, an architect chooses a single size for both data items and addresses.*

15.11 Bus Address Space

A *memory bus* (i.e., a bus that the processor uses to access memory) is the easiest form of a bus to understand. Previous chapters discuss the concepts of memory access and a memory address space; we will see how a bus is used to implement the concepts. As Figure 15.7 illustrates, a memory bus provides a physical interconnection among a processor and one or more memories.

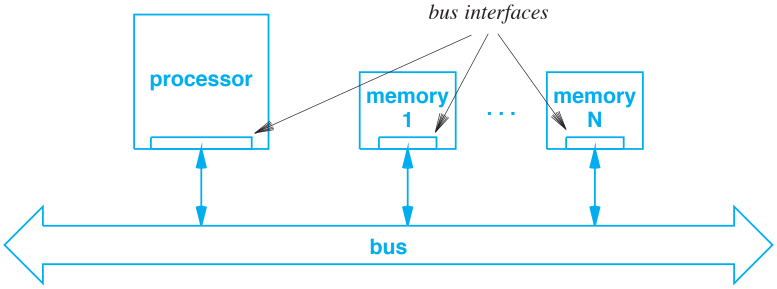


Figure 15.7 Physical interconnections of a processor and memory using a memory bus. A controller circuit in each device handles the details of bus access.

As the figure shows, the processor and memory modules connected to a memory bus each contain an interface circuit. The interface implements the bus protocol, and handles all bus communication. The interface uses the control lines to gain access to the bus, and then sends addresses or data values to carry out the operation. Thus, only the interface understands the bus details, such as the voltage to use and the timing of control signals.

From a processor's point of view, the bus interface provides the fetch-store paradigm. That is, the processor can only perform two operations: *fetch* from a bus address and *store* to a bus address. When the processor encounters an instruction that references memory, the processor hardware invokes the bus interface. For example, on many architectures, a *load* instruction fetches a value from memory and places the value in a general-purpose register. When the processor executes a *load*, the hardware issues a *fetch* instruction to the bus interface. Similarly, if the processor executes an instruction that deposits a value in memory, the hardware uses a *store* operation on the bus interface.

From a programmer's point of view, the bus interface is invisible. The programmer thinks of the bus as defining an *address space*. The key to creating a single address space lies in memory configuration — each memory is configured to respond to a specific set of bus addresses. That is, the interface for memory 1 is assigned a different set of addresses than the interface for memories 2, 3, 4, and so on. When a processor places a *fetch* or *store* request on the bus, all memory controllers receive the request. Each memory controller compares the address in the request to the set of addresses the memory module has been assigned. If the address in the request lies within the controller's set, the controller responds. Otherwise, it ignores the request. The point is:

> *When a request passes across a bus, all attached memory modules receive the request. A memory module only responds if the address in the request lies in the range that has been assigned to the module.*

15.12 Potential Errors

Figure 15.8. lists the conceptual steps that each memory module interface implements.

```
Let R be the range of addresses assigned to this module
Repeat forever {
        Monitor the bus until a request appears;
        if (the request specifies an address in R) {
                respond to the request
        } else {
                ignore the request
        }
}
```

Figure 15.8 The steps the bus interface in a memory module follows.

An error that the bus hardware reports is referred to as a *bus error*; a typical bus protocol includes mechanisms that detect and report each type of bus error. Allowing each memory module to act independently means two types of bus errors can occur:

- Address conflict
- Unassigned address

Address Conflict. We use the term *address conflict* to describe a bus error that results when two or more interfaces are misconfigured so they each respond to a given address. Some bus hardware is designed to detect and report address conflicts when the system boots. Other hardware is designed to prevent conflicts. In any case, most bus protocols include a test for address conflicts that occur at run-time — if two or more interfaces attempt to respond to a given request, the bus hardware detects the problem and sets a control line to indicate that an error occurred. When it uses a bus, a processor checks the bus control lines and takes action if an error occurs.

Unassigned Address. An *unassigned address* bus error occurs if a processor attempts to access an address that has not been assigned to any interface. To detect an unassigned address, most bus protocols use a *timeout* mechanism — after sending a request over the bus, the processor starts a timer. If no interface responds, the timer expires, which causes the processor hardware to report the bus error. The same timeout mechanism used to detect unassigned addresses also detects malfunctioning hardware (e.g., a memory module that is not responding to requests).

15.13 Address Configuration And Sockets

Some bus hardware prevents bus errors. Unassigned addresses pose a thorny problem for prevention. On the one hand, to prevent bus errors, each possible address must be assigned to a memory module. On the other hand, most memory systems are designed to accommodate expansion. That is, a bus typically contains enough wires to address more memory than is installed in the computer (i.e., some addresses will be unassigned).

Fortunately, architects have devised a scheme that helps avoid the problem of two modules that both answer to a given request: *special sockets*. The idea is straightforward. Memory is manufactured on small printed circuits that each plug into a socket on the mother board. To avoid problems caused by misconfiguration, all memory boards are identical, and no configuration is required before a board is plugged in. Instead, circuitry and wiring is added to the mother board so that the first socket only receives requests for address 0 through $K-1$, the second socket only receives requests for address K through $2K-1$, and so on. When a socket does recognize an address, the socket passes the low-order bits of the address on to the memory. The point is:

> *To avoid memory configuration problems, architects can place memory on small circuit boards that each plug into a socket on the mother board. An owner can install memory without configuring the hardware because each socket is configured with the range of addresses to which the memory should respond.*

As an alternative, some computers contain sophisticated circuitry that allows the MMU to configure socket addresses when the computer boots. The MMU determines which sockets are populated, and assigns each a range of addresses. Although it adds cost, the extra circuitry to prevent conflicts makes installing memory much easier — an owner can purchase memory modules and plug them into sockets without configuring the modules and with no danger of conflicts.

15.14 The Question Of Multiple Buses

Should a computer system have multiple buses? If so, how many? Computers designed for high performance (e.g., mainframe computers) often contain several buses. Each bus is optimized for a specific purpose. For example, a mainframe computer might have one bus for memory, another for high-speed I/O devices, and another for slow-speed I/O devices. As an alternative, less powerful computers (e.g., personal computers) often use a single bus for all connections. The chief advantages of a single bus are lower cost and more generality. A processor does not need multiple bus interfaces, and a single bus interface can be used for both memory and devices.

Of course, designing a single bus for all connections means choosing a compromise. That is, the bus may not be optimal for any given purpose. In particular, if the processor uses a single bus to access instructions and data in memory as well as perform I/O, the bus can easily become a bottleneck. Thus, a system that uses a single bus often needs a large memory cache that can answer most of the memory requests without using the bus.

15.15 Using Fetch-Store With Devices

Recall that a bus is used as the primary connection between a processor and an I/O device, and that all operations on a bus must use the fetch-store paradigm. The two statements may seem contradictory — although it works well for data transfer, fetch-store does not appear to handle device control. For example, consider an operation like testing whether a wireless radio is currently in range of an access point or moving paper through a printer. It seems that *fetch* and *store* operations are insufficient, and that devices require a large set of control commands.

To understand how a bus works, we must remember that a bus provides a way to communicate a set of bits from one unit to another without specifying what each bit means. The names *fetch* and *store* mislead us into thinking about values in memory. On a bus, however, the interface hardware of each device provides a unique interpretation of the bits. Thus, a device can interpret certain bits as a control operation rather than as a request to transfer data.

An example will clarify the relationship between the fetch-store paradigm and device control. Imagine a simplistic hardware device that contains sixteen status lights, and suppose we want to attach the device to a bus. Because the bus only offers *fetch* and *store* operations, we need to build interface hardware that uses the fetch-store paradigm for control. An engineer who designs a device interface begins by listing the operations to be performed. Figure 15.9 lists the five functions for our imaginary device.

- **Turn the display on**
- **Turn the display off**
- **Set the display brightness**
- **Turn the ith status light on**
- **Turn the ith status light off**

Figure 15.9 An example of functionality needed for an imaginary status light display. Each function must be implemented using the fetch-store paradigm.

To cast control operations in the fetch-store paradigm, a designer chooses a set of bus addresses that are not used by other devices, and assigns meanings to each address. For example, if our imaginary status light device is attached to a bus that has a width of thirty-two bits, a designer might choose bus addresses 10000 through 10011, and might assign meanings according to Figure 15.10.

Address	Operation	Meaning
10000 – 10003	store	Nonzero data value turns the display on, and a zero data value turns the display off
10000 – 10003	fetch	Returns zero if display is currently off, and nonzero if display is currently on
10004 – 10007	store	Change brightness. Low-order four bits of the data value specify brightness value from zero (dim) through fifteen (bright)
10008 – 10011	store	The low order sixteen bits each control a status light; a zero bit sets the corresponding light off and a one bit sets the light on

Figure 15.10 Example assignment of addresses, operations, and meanings for the device control functions listed in Figure 15.9.

15.16 Operation Of An Interface

Although bus operations are named *fetch* and *store*, a device interface does not act like a memory — data is not stored for later recall. Instead, a device treats the address, operation, and data in a bus request merely as a set of bits. The interface contains logic circuits that compare the address bits in each request to the addresses assigned to the device. If a match occurs, the interface enables a circuit that responds to the *fetch* or *store* operation. For example, the first item in Figure 15.10 can be implemented by a circuit that tests bits on the bus for a *store* request with address 10000 and uses the data to take action. In essence, the circuit performs the following test:

```
if ( address == 10000 && op == store && data != 0 )
        turn_on_display ;
} else if ( address == 10000 && op == store && data == 0 ) {
        turn_off_display ;
}
```

Although we have used programming language notation to express the operations, interface hardware does not perform the test sequentially. Instead, an interface is constructed from Boolean circuits that can test the address, operation, and data values in parallel and take the appropriate action.

15.17 Asymmetric Assignments And Bus Errors

The example in Figure 15.10 does not define the effect of *fetch* or *store* operations on some of the addresses. For example, the specification does not define a *fetch* operation for address 10004. To capture the idea that *fetch* and *store* operations do not need to be defined for each address, we say that the assignment is *asymmetric*. The specification in Figure 15.10 is asymmetric because the processor can store a value to the four bytes starting at address 10004, but a bus error results if the processor attempts to read from address 10004.

15.18 Unified Memory And Device Addressing

In some computers, a single bus provides access to both memory and I/O devices. In such an architecture, the assignment of addresses on the bus defines the processor's view of the address space. For example, imagine a computer system with a single bus as Figure 15.11 illustrates.

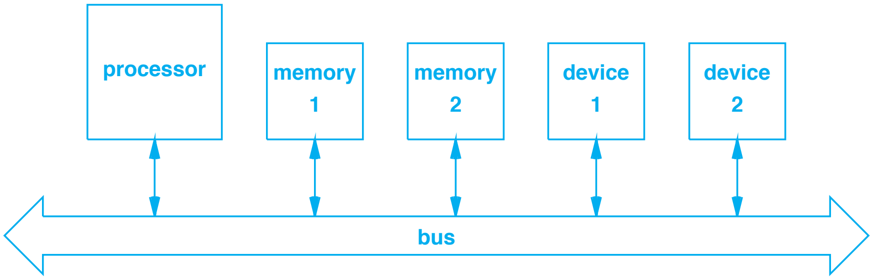


Figure 15.11 Illustration of a computer architecture that uses a single bus. The bus connects memories as well as devices.

In the figure, the bus defines a single address space that the processor can use. Each memory module and each device must be assigned a unique address range of bus addresses. For example, if we assume the memories are each 1 Mbyte and each device requires twelve memory locations, four address ranges must be assigned for use on the bus as Figure 15.12 illustrates.

Device	Address Range		
memory 1	0x000000	through	0x0fffff
memory 2	0x100000	through	0x1fffff
device 1	0x200000	through	0x20000b
device 2	0x20000c	through	0x200017

Figure 15.12 One possible assignment of bus addresses for the set of devices shown in Figure 15.11.

We can also imagine the address space drawn graphically like the illustrations of a memory address space in Chapter 11. Of course the space occupied by each device is extremely small compared to the space occupied by a memory, which means the diagram will not show much detail. For example, Figure 15.13 shows the diagram that results from the assignments in Figure 15.12.

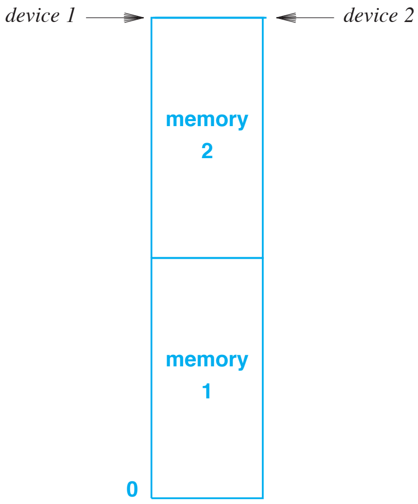


Figure 15.13 Illustration of the address space that results from the address assignments in Figure 15.12. The amount of space taken by each device (twelve bytes) is insignificant compared to the amount of space taken by each memory (1 Mbyte).

15.19 Holes In A Bus Address Space

The address assignment in Figure 15.12 is said to be *contiguous*, which means that the address ranges do not contain gaps — the first byte assigned to one range is the immediate successor of the last byte assigned to the previous range. Contiguous address assignment is not required — if the software accidentally accesses an address that has not been assigned, the bus hardware detects the problem and reports a bus error.

Using the terminology from Chapter 13, we say that if an assignment of addresses is not contiguous, the assignment leaves one or more *holes* in the address space. For example, a bus may reserve lower addresses for memory and assign devices to high addresses, leaving a hole between the two areas.

15.20 Address Map

As part of the specification, a bus standard specifies exactly which type of hardware can be used at each address. We call the specification an *address map*. Note that an address map is not the same as an address assignment because a map only specifies what assignments are possible. For example, Figure 15.14 gives an example of an address map for a sixteen-bit bus.

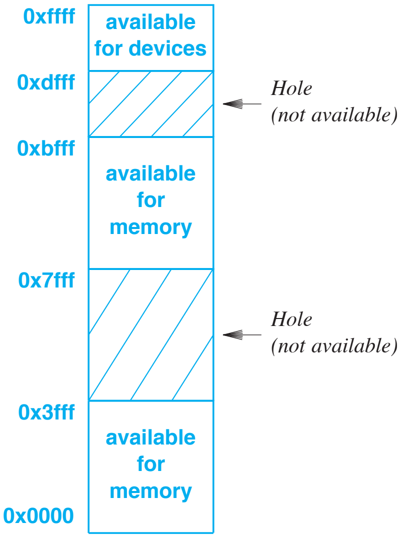


Figure 15.14 One possible address map for a sixteen-bit bus. Two areas are available for memory, and one area is available for devices.

In the figure, the two areas of the address space available for memory are not contiguous. Instead, a hole is located between them. Furthermore, a hole is located between the second memory area and the device area.

When a computer system is constructed, the owner must follow the address map. For example, the sixteen-bit bus in Figure 15.14 only allows two blocks of memory that total 32,768 bytes. The owner can choose to install less than a full complement of memory, but not more.

The device space in a bus address map is especially interesting because the space reserved for devices is often much larger than necessary. In particular, most address maps reserve a large piece of the address space for devices, making it possible for the bus to accommodate extreme cases with thousands of devices. However, a typical computer has fewer than a dozen devices, and a typical device uses a few addresses. The consequence is:

> *In a typical computer, the part of the address space available to devices is sparsely populated — only a small percentage of the available addresses are used.*

15.21 Program Interface To A Bus

From a programmer's point of view, there are two ways to use a bus. Either a processor provides special instructions used to access each bus or the processor interprets all memory operations as references to the bus. The latter is known as a *memory mapped architecture*. As an example of using a memory-mapped approach, consider address assignment of the imaginary light display described in Figure 15.10[†]. To turn the device on, the program must store a nonzero value in bytes 10000 through 100003. If we assume an integer consists of four bytes (i.e., thirty-two bits) and the processor uses little-endian byte order, the program only needs to store a nonzero value into the integer at location 10000. A programmer can use the following C code to perform the operation:

```
int     *ptr;          /* declare ptr to be a pointer to an integer */
ptr = (*int)10000;     /* set pointer to address 10000 */
*ptr = 1;              /* store nonzero value in addresses 10000 - 10003 */
```

We can summarize:

> *A processor can use special instructions to access a bus or can use a memory-mapped approach in which normal memory operations are used to communicate with devices as well as memory.*

[†] Figure 15.10 appears on page 301.

15.22 Bridging Between Two Buses

Although a single bus offers the advantage of simplicity and lower cost, a given device may only work with a specific bus. For example, some earphones require a *Universal Serial Bus* (*USB*) and some Ethernet devices require a *Peripheral Component Interconnect* (*PCI*) bus. Clearly, a computer that has multiple buses can accommodate a greater variety of devices. Of course, a system with multiple buses can be expensive and complex. Therefore, designers have created inexpensive and straightforward ways to attach multiple buses to a computer. One approach uses a hardware device, known as a *bridge*, that interconnects two buses as Figure 15.15 illustrates.

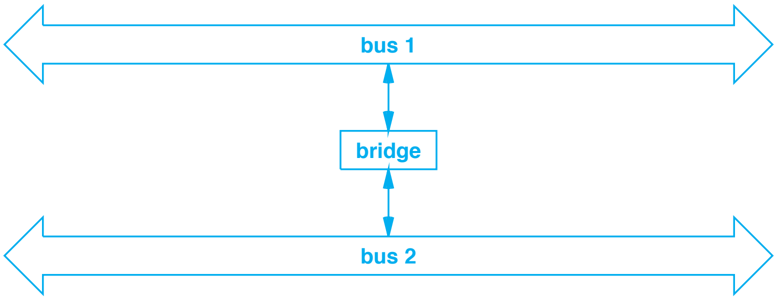


Figure 15.15 Illustration of a bridge connecting two buses. The bridge must follow the standard for each bus.

The bridge uses a set of K addresses. Each bus chooses an address range of size K and assigns it to the bridge. The two assignments are not usually the same; the bridge is designed to perform translation. Whenever an operation on one bus involves the addresses assigned to the bridge, circuits in the bridge translate the address and perform the operation on the other bus. Thus, if a processor on bus 1 performs a *store* operation to one of the bridged addresses, the bridge hardware performs an equivalent *store* operation on bus 2. In fact, bridging is *transparent* in the sense that processors and devices are unaware that multiple bridges are involved.

15.23 Main And Auxiliary Buses

Logically, a bridge performs a one-to-one mapping from the address space of one bus to the address space of another. That is, the bridge maps a set of addresses on one bus into the address space of the other. Figure 15.16 illustrates the concept of address mapping. In the figure, both bus address spaces start at zero, and the address space of the auxiliary bus is smaller than the address space of the main bus. More important, the architect has chosen to map only a small part of the auxiliary bus address space, and has specified that it maps onto a region of the main bus that is reserved for devices. As a

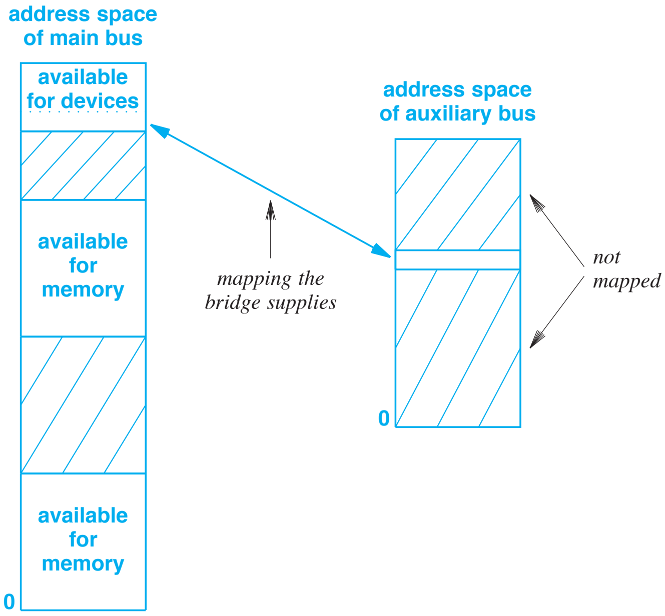

address space of main bus
available for devices
available for memory
available for memory
0
mapping the bridge supplies
address space of auxiliary bus
0
not mapped

Figure 15.16 Illustration of a mapping that a bridge can provide between the address space of an auxiliary bus and the address space of a main bus. Only some bus addresses need to be mapped.

result, any device on the auxiliary bus that responds to addresses in the mapped region appears to be connected to the computer's main bus.

To understand why bridging is popular, consider a common case where a new device must be added to a computer that already has a bus. If the interface on the new device does not match the computer's main bus, new adapter hardware can be created or a bridge can be used to add an *auxiliary bus* to the system. Using a bridge has two advantages: bridging is simpler than adding a bus interface to each new device, and once a bridge has been installed, a computer owner can add additional devices to the auxiliary bus without changing the hardware further.

To summarize:

> *A bridge is a hardware device that interconnects two buses and maps addresses between them. Bridging allows a computer to have one or more auxiliary buses that are accessed through the computer's main bus.*

15.24 Consequences For Programmers

As Figure 15.16 shows, the sets of mapped addresses do not need to be identical in both address spaces. The goal is to make a bridge so transparent that software does not know about the auxiliary bus. Unfortunately, a programmer who writes device driver software or someone who configures computers may need to understand the mapping. For example, when a device is installed in an auxiliary bus, the device obtains a bus address, A. As part of the initialization sequence, the device may report its bus address to the driver software†. Because a bridge only translates addresses, communication between the device and the driver that uses data lines will not be changed. Thus, to generate an address on the main bus, the driver software may need to understand how the bridge maps addresses.

15.25 Switching Fabrics As An Alternative To Buses

Although a bus is fundamental to most computer systems, a bus has a disadvantage: bus hardware can only perform one transfer at a time. That is, although multiple hardware units can attach to a given bus, at most one pair of attached units can communicate at any time. The basic paradigm always consists of three steps: wait for exclusive use of the bus, perform a transfer, and release the bus so another transfer can occur.

Some buses extend the paradigm by permitting multiple attached units to transfer N bytes of data each time they obtain the bus. For situations where bus architectures are insufficient, architects have invented alternative technologies that permit multiple transfers to occur simultaneously. Known as *switching fabrics*, the technologies use a variety of forms. Some fabrics are designed to handle a few attached units, and other fabrics are designed to handle hundreds or thousands. Similarly, some fabrics restrict transfers so only a few attached units can initiate transfers at the same time, and other fabrics permit many simultaneous transfers. One of the reasons for the variety of architectures arises from economics: higher performance (i.e., more simultaneous exchanges) can cost much more, and the higher cost may not be justified.

Perhaps the easiest switching fabric to understand consists of a *crossbar switch*. We can imagine a crossbar to be a matrix with N inputs and M outputs. The crossbar contains $N \times M$ electronic switches that each connect an input to an output. At any time, the crossbar can turn on switches to connect pairs of inputs and outputs as Figure 15.17 illustrates.

†Chapter 17 explains why a device driver needs address information.

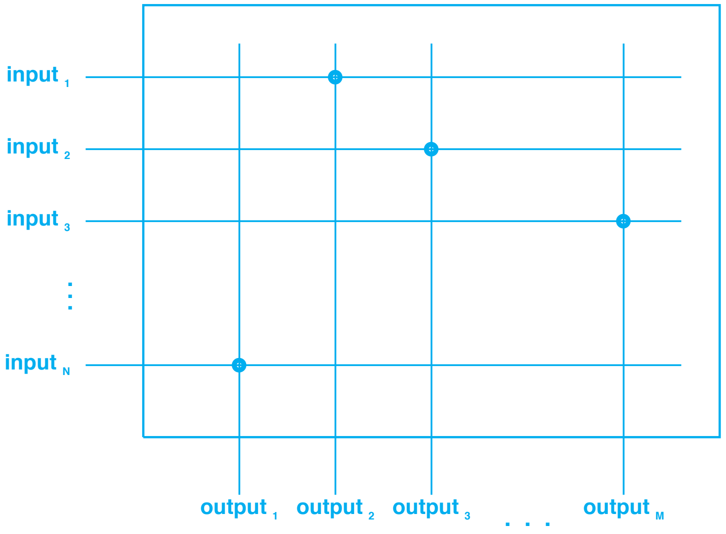


Figure 15.17 A conceptual view of a crossbar switch with N inputs and M outputs with a dot showing an active connection. The crossbar mechanism ensures that only one connection is active for a given row or a given column at any time.

The figure helps us understand why switching fabrics are expensive. First, each line in the diagram represents a parallel data path composed of multiple wires. Second, each potential intersection between an input and output requires an electronic switch that can connect the input to the output at that point. Thus, a crossbar requires $N \times M$ switching components, each of which must be able to switch a parallel connection. By comparison, a bus only requires $N + M$ electronic components (one to connect each input and each output to the bus). Despite the cost, switching fabrics are popular for high-performance systems.

15.26 Summary

A bus is the fundamental mechanism used to interconnect memory, I/O devices, and processors within a computer system. Most buses operate in parallel, meaning that the bus consists of parallel wires that permit multiple bits to be transferred simultaneously.

Each bus defines a protocol that attached devices use to access the bus. Most bus protocols follow the fetch-store paradigm; an I/O device connected to a bus is designed to receive *fetch* or *store* operations and interpret them as control operations on the device.

Conceptually, a bus protocol specifies three separate forms of information: control information, address information, and data. In practice, a bus does not need independent wires for each form because a bus protocol can multiplex communication over a small set of wires.

A bus defines an address space that may contain holes (i.e., unassigned addresses). A computer system can have a single bus to which memory and I/O devices attach, or can have multiple buses that each attach to specific types of devices. As an alternative, a hardware device called a *bridge* can be used to add multiple auxiliary buses to a computer by mapping all or part of the auxiliary bus address space onto the address space of the computer's main bus.

The chief alternative to a bus is known as a *switching fabric*. Although they achieve higher throughput by using parallelism, switching fabrics are restricted to high-end systems because a switching fabric is significantly more expensive than a bus.

EXERCISES

15.1 A hardware architect asks you to choose between a single, thirty-two bit bus design that multiplexes both data and address information across the bus or two sixteen-bit buses, one used to send address information and one used to send data. Which design do you choose? Why?

15.2 In a computer, what is a bus, and what does it connect?

15.3 Your friend claims that their computer has a special bus that is patented by Apple. What term do we use to characterize a bus design that is owned by one company?

15.4 What are the three conceptual categories of wires in the bus?

15.5 What is the fetch-store paradigm?

15.6 If the lines on a bus are divided into control lines and other lines, what are the main two uses of the other lines?

15.7 What is the advantage of having a separate socket for each memory chip?

15.8 Suppose a device has been assigned addresses 0x4000000 through 0x4000003. Write C code that stores the value 0xff0001A4 into the addresses.

15.9 If a bus can transfer 64 bits in each cycle and runs at a rate of 66 MHz, what is the bus throughput measured in megabytes per second?

15.10 What is a switching fabric, and what is its chief advantage over a bus?

15.11 How many simultaneous transfers can occur over a crossbar switching fabric of N inputs and M outputs?

15.12 Search the Internet, and make a list of switching fabric designs.

15.13 Look on the Internet for an explanation of a *CLOS network*, which is used in switching fabrics, and write a short description.

15.14 What does a bridge connect?

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

16

Programmed And Interrupt-Driven I/O

16.1 Introduction

Earlier chapters introduce I/O. The previous chapter explains how a bus provides the connection between a processor and a set of I/O devices. The chapter discusses the bus address space, and shows how an address space can hold a combination of both memory and I/O devices. Finally, the chapter explains that a bus uses the fetch-store paradigm, and shows how *fetch* and *store* operations can be used to interrogate or control an external device.

This chapter continues the discussion. The chapter describes and compares the two basic styles of interaction between a processor and an I/O device. It focuses on interrupt-driven I/O, and explains how device driver software in the operating system interacts with an external device.

The next chapter takes a different approach to the subject by examining I/O from a programmer's perspective. The chapter looks at individual devices, and describes how they interact with the processor.

16.2 I/O Paradigms

We know from the previous chapter that I/O devices connect to a bus, and that a processor can interact with the device by issuing *fetch* and *store* operations to bus addresses that have been assigned to the device. Although the basic mechanics of I/O are easy to specify, several questions remain unanswered. What control operations should each device support? How does application software running on the processor access a given device without understanding the hardware details? Can the interaction between a processor and I/O devices affect overall system performance?

16.3 Programmed I/O

The earliest computers took a straightforward approach to I/O: an external device consisted of basic digital circuits that controlled the hardware in response to *fetch* and *store* operations; the CPU handled all the details. For example, to write data on a disk, the CPU activated a set of circuits in the device, one at a time. The circuits positioned the disk arm and caused the head to write a block of data. To capture the idea that an early peripheral device consisted only of basic circuits that respond to commands from the CPU we say that the device contained no intelligence, and characterize the form of interaction by saying that the I/O is *programmed*.

16.4 Synchronization

It may seem that writing software to perform programmed I/O is trivial: a program merely assigns a value to an address on the bus. To understand I/O programming, however, we need to remember two things. First, a nonintelligent device cannot remember a list of commands. Instead, circuits in the device perform each command precisely when the processor sends the command. Second, a processor operates much faster than an I/O device — even a slow processor can execute thousands of instructions in the time it takes for a motor or mechanical actuator to move a physical mechanism.

As an example of a mechanical device, consider a printer. The print mechanism can spray ink across the page, but can only print a small vertical band at any time. Printing starts at the top of the page. After printing one horizontal band, the paper must be advanced before the next horizontal band can be printed. If a processor merely issues instructions to print an item, advance the paper, and print another item, the second item may be printed while the paper is still moving, resulting in a smear. In the worst case, if the print mechanism is not designed to operate while the paper advance mechanism operates, the hardware may be damaged.

To prevent problems, programmed I/O relies on *synchronization*. That is, once it issues a command, the processor must interact with the device to wait until the device is ready for another command. We can summarize:

> *Because a processor operates orders of magnitude faster than an I/O device, programmed I/O requires the processor to synchronize with the device that is being controlled.*

16.5 Polling

The basic form of synchronization that a processor uses with an I/O device is known as *polling*. In essence, polling requires the processor to ask repeatedly whether an operation has completed before the processor starts the next operation. Thus, to perform a *print* operation, a processor can use polling at each step. Figure 16.1 shows an example.

- Test to see if the printer is powered on
- Cause the printer to load a blank sheet of paper
- Poll to determine when the paper has been loaded
- Specify data in memory that tells what to print
- Poll to wait for the printer to load the data
- Cause the printer to start spraying a band of ink
- Poll to determine when the ink mechanism finishes
- Cause the printer to advance the paper to the next band
- Poll to determine when the paper has advanced
- Repeat the above six steps for each band to be printed
- Cause the printer to eject the page
- Poll to determine when the page has been ejected

Figure 16.1 Illustration of synchronization between a processor and an I/O device. The processor must wait for each step to complete.

16.6 Code For Polling

How does software perform polling? Because a bus follows the fetch-store paradigm, polling must use a *fetch* operation. That is, one or more of the addresses assigned to the device correspond to status information — when the processor fetches a value from the address, the device responds by giving its current status.

To understand how polling appears to a programmer, we need to know the exact details of a hardware device. Unfortunately, most devices are incredibly complex. For example, many vendors sell three-in-one printers that can function as scanners or fax machines as well as printers. To keep an example simple, we will imagine a simple printing device, and create a programming interface for the device. Although our example device is indeed much simpler than commercial devices, the general approach is exactly the same.

Recall that a device is assigned addresses in the address space, and the device is engineered to respond to *fetch* and *store* instructions to those addresses. When a device is created, the designer does not specify the addresses that will be used, but instead writes a *relative* specification by giving addresses 0 though *N–1*. Later, when the device is installed in a computer, actual addresses are assigned. The use of relative addresses in the specification means a programmer can write software to control the device without knowing the actual address. Once the device is installed, the actual address can be passed to the software as an argument.

An example will clarify the concept. Our imaginary printer defines thirty-two contiguous bytes of addresses. Furthermore, the design has grouped the addresses into eight words that are each thirty-two bits long. The use of words is typical. The specification in Figure 16.2 shows how the device interprets *fetch* and *store* operations for each of the addresses.

Addresses	Operation	Meaning
0 – 3	fetch	Nonzero if the printer is powered on
4 – 7	store	Nonzero starts loading a sheet of paper
8 – 11	store	Memory address of data to print
12 – 15	store	Nonzero causes printer to pick up address
16 – 19	store	Start the inkjet spraying current band
20 – 23	store	Nonzero advances paper to the next band
24 – 27	fetch	Busy: nonzero when device is busy
28 – 31	fetch	CMYK ink levels in four octets

Figure 16.2 A specification for the bus interface on an imaginary printing device. A processor uses *fetch* and *store* to control the device and determine its status.

The figure gives the meaning of *fetch* and *store* operations on addresses assigned to our imaginary I/O device. As described above, addresses in the specification start at zero because they are relative. When the device is connected to a bus, the device will be assigned thirty-two bytes somewhere in the bus address space, and software will use the actual addresses when communicating with the device.

Once a programmer is given a hardware specification similar to the one in Figure 16.2, writing code that controls a device is straightforward. For example, assume our printing device has been assigned the starting bus address 0x110000. Addresses 0 through 3 in the figure will correspond to actual addresses 0x110000 through 0x110003. To determine whether the printer is powered on, the processor merely needs to access the value in addresses 0x110000 through 0x110003. If the value is zero, the printer is off. The code to access the device status appears to be a memory fetch. In C, the status test code can be written:

```
int     *p = (int *)0x110000;

if (*p != 0) {                  /* Test whether printer is on */
        /* printer is on */
} else {
        /* printer is off */
}
```

The example assumes an integer size of four bytes. The code declares *p* to be a pointer to an integer, initializes *p* to 0x110000, and then uses **p* to obtain the value at address 0x110000.

Now that we understand how software communicates with a device, we can consider a sequence of steps and synchronization. Figure 16.3 shows C code that performs some of the steps found in Figure 16.1.

```
int     *p;                 /* Pointer to the device address area */

p = (int *)0x110000;        /* Initialize pointer to device address */
if (*p == 0)                /* Test if printer is powered on */
        error("printer not on");
*(p+1) = 1;                 /* Start loading paper */
while (*(p+6) != 0)         /* Poll to wait for the load to complete */
        ;
*(p+2) = &mydata;           /* Specify the location of data in memory */
*(p+3) = 1;                 /* Cause printer to pick up data */
while (*(p+6) != 0)         /* Poll to wait for printer to complete loading data  */
        ;
*(p+4) = 1;                 /* Start inkjet spraying */
while (*(p+6) != 0)         /* Poll to wait for the inkjet to finish */
        ;
*(p+5) = 1;                 /* Advance the paper to the next band */
while (*(p+6) != 0)         /* Poll to wait for the paper advance to complete*/
        ;
```

Figure 16.3 Example C code that uses polling to carry out some of the steps from Figure 16.1 on the imaginary printing device specified in Figure 16.2.

Code in the figure assumes the device has been assigned address 0x110000, and that data structure *mydata* contains the data to be printed in exactly the form the printer expects. To understand the use of pointers, remember that the C programming language defines *pointer arithmetic*: adding K to an integer pointer advances the pointer by KN bytes, where N is the number of bytes in an integer. Thus, if variable p has the value 0x110000, $p+1$ equals 0x110004.

The example code illustrates another feature of many devices that may seem strange to a programmer: multiple steps for a single operation. In our example, the data to be printed is located in memory and two steps are used to specify data. In the first step, the address of the data is passed to the printer. In the second step, the printer is instructed to load a copy of the data. Having two steps may seem unnecessary — why doesn't the printer start loading data automatically once an address has been specified? To understand, remember that each *fetch* and *store* instruction controls circuits in the device. A device designer might choose such a design because he or she finds it easier to build hardware that has two separate circuits, one to accept a memory address and one to load data from memory.

Programmers who have not written a program to control a device may find the code shocking because it contains four occurrences of a while statement that each appear to be an infinite loop. If such a statement appeared in a conventional application program, the statement would be in error because the loop continually tests the value at a memory location without making any changes. In the example, however, pointer *p* references a device instead of a memory location. Thus, when the processor fetches a value from location $p+6$, the request passes to a device, which interprets it as a request for status information. So, unlike a value in memory, the value returned by the device will change over time — if the processor polls enough times, the device will complete its current operation and return zero as the status value. The point is:

> *Although polling code appears to contain infinite loops, the code can be correct because values returned by a device can change over time.*

16.7 Control And Status Registers

We use the term *Control and Status Registers* (*CSRs*) to refer to the set of addresses that a device uses. More specifically, a *control register* corresponds to a contiguous set of addresses (usually the size of an integer) that respond to a *store* operation, and a *status register* corresponds to a contiguous set of addresses that respond to a *fetch* operation.

In practice, CSRs are usually more complicated than the simplified version listed in Figure 16.2. For example, a typical status register assigns meanings to individual bits (e.g., the low-order bit of the status word specifies whether the device is in motion, the next bit specifies whether an error has occurred, and so on). More important, to conserve addresses, many devices combine control and status functions into a single set of addresses. That is, a single address can serve both functions — a *store* operation to the address controls the device, and a *fetch* operation to the same address reports the device status.

As a final detail, some devices interpret a *fetch* operation as both a request for status information and a *control* operation. For example, a trackpad delivers bytes to indicate motion of a user's fingers. The processor uses a *fetch* operation to obtain data from the trackpad. Furthermore, each *fetch* automatically resets the hardware to measure the next motion.

16.8 Using A Structure To Define CSRs

The example code in Figure 16.3 uses a pointer and pointer arithmetic to reference individual items. In practice, programmers usually create a C struct that defines the CSRs, and then use named members of the struct to reference items in the CSRs. For example, Figure 16.4 shows how the code from Figure 16.3 appears when a struct is used to define the CSRs.

```
struct   csr {                    /* Template for printer CSRs */
    int  csr_power;               /* Is printer powered on? */
    int  csr_load;                /* Load a sheet of paper */
    int  csr_addr;                /* Specify address of data to print */
    int  csr_getdata;             /* Upload data from memory */
    int  csr_spray;               /* Start inkjet spraying */
    int  csr_advance;             /* Advance paper to next band */
    int  csr_dev_busy;            /* Nonzero => device busy */
    int  csr_levels;              /* CMYK Ink levels in 4 bytes */
}
struct   csr   *p;                /* Pointer to the device address area */

p = (struct csr *)0x110000;       /* Set p to device address */
if (p->csr_power == 0);           /* Test if printer is on */
    error("printer not on");
p->csr_load = 1;                  /* Start loading paper */
while (p->csr_dev_busy)           /* Poll to wait for the load to complete */
    ;
p->csr_addr = &mydata             /* Specify the location of data in memory */
p->csr_getdata = 1;               /* Cause printer to pick up data */
while (p->csr_dev_busy)           /* Poll to wait for printer to complete loading data  */
    ;
p->csr_spray = 1;                 /* Start the inkjet spraying */
while (p->csr_dev_busy)           /* Poll to wait for the inkjet to finish */
    ;
p->csr_ = 1;                      /* Advance the paper to the next band */
while (p->csr_dev_busy)           /* Poll to wait for the paper advance to complete*/
    ;
```

Figure 16.4 The code from Figure 16.3 rewritten to use a C struct.

As the example shows, code that uses a struct is much easier to read and debug. Because members of the struct can be given meaningful names, a programmer reading the code can guess at the purpose, even if they are not intimately familiar with a device. In addition, using a struct improves program organization because all the offsets of individual CSRs are specified in one place instead of being embedded throughout the code. To summarize:

> *Instead of distributing CSR references throughout the code, a programmer can improve readability by declaring a structure that defines all the CSRs for a device and then referencing fields in the structure.*

16.9 Processor Use And Polling

The chief advantage of a programmed I/O architecture arises from the economic benefit: because they do not contain sophisticated digital circuits, devices that rely on programmed I/O are inexpensive. The chief disadvantage of programmed I/O arises from the computational overhead: each step requires the processor to interact with the I/O device.

To understand why polling is especially undesirable, we must recall the fundamental mismatch between I/O devices and computation: because they are electromechanical, I/O devices operate several orders of magnitude slower than a processor. Furthermore, if a processor uses polling to control an I/O device, the amount of time the processor waits is fixed, and is independent of the processor speed.

The important point can be summarized:

> *Because a typical processor is much faster than an I/O device, the speed of a system that uses polling depends only on the speed of the I/O device; using a fast processor will not increase the rate at which I/O is performed.*

Turning the statement around produces a corollary: if a processor uses polling to wait for an I/O device, using a faster processor merely means that the processor will execute more instructions waiting for the device (i.e., loops, such as those in Figure 16.3, will run faster). Thus, a faster processor merely wastes more cycles waiting for an I/O device — if the processor did not need to poll, the processor could be performing computation instead.

16.10 Interrupt-Driven I/O

In the 1950s and 1960s, computer architects became aware of the mismatch between the speed of processors and I/O devices. The difference was particularly important when the first generation of computers, which used vacuum tubes, was replaced by a second generation that used solid-state devices. Although the use of solid-state devices (i.e., transistors) increased the speed of processors, the speed of I/O devices remained approximately the same. Thus, architects explored ways to overcome the mismatch between I/O and processor speeds.

One approach emerged as superior, and led to a revolution in computer architecture that produced the third generation of computers. Known as an *interrupt* mechanism, the facility is now standard in processor designs.

The central premise of interrupt-driven I/O is straightforward: instead of wasting time polling, allow a processor to continue to perform computation while an I/O device operates. When the device finishes, arrange for the device to inform the processor so that the processor can handle the device. As the name implies, the hardware temporarily interrupts the computation in progress to handle I/O. Once the device has been serviced, the processor resumes the computation exactly where it was interrupted.

In practice, interrupt-driven I/O requires that all aspects of a computer system be designed to support interrupts, including:

- I/O device hardware
- Bus architecture and functionality
- Processor architecture
- Programming paradigm

I/O Device Hardware. Instead of merely operating under control of a processor, an interrupt-driven I/O device must operate independently once it has started. Later, when it finishes, a device must be able to interrupt the processor.

Bus Architecture And Functionality. A bus must support two-way communication that allows a processor to start an operation on a device and allows the device to interrupt the processor when the operation completes.

Processor Architecture. A processor needs a mechanism that can cause the processor to suspend normal computation temporarily, handle a device, and then resume the computation.

Programming Paradigm. Perhaps the most significant change involves a shift in the programming paradigm. Polling uses a sequential, *synchronous* style of programming in which the programmer specifies each step of the operation an I/O device performs. As we will see in the next chapter, interrupt-driven programming uses an *asynchronous* style of programming in which the programmer writes code to handle events.

16.11 An Interrupt Mechanism And Fetch-Execute

As the term *interrupt* implies, device events are temporary. When a device needs service (e.g., when an operation completes), hardware in the device sends an interrupt signal over the bus to the processor. The processor temporarily stops executing instructions, saves the state information needed to resume execution later, and handles the device. When it finishes handling an interrupt, the processor reloads the saved state and resumes executing exactly at the point the interrupt occurred. That is:

> *An interrupt mechanism temporarily borrows the processor to handle an I/O device. Hardware saves the state of the computation when an interrupt occurs and resumes the computation once interrupt processing finishes.*

From an application programmer's point of view, an interrupt is *transparent*, which means a programmer writes application code as if interrupts do not exist. The hardware is designed so that the result of computation is the same if no interrupts occur, one interrupt occurs, or many interrupts occur during the execution of the instructions.

How does I/O hardware interrupt a processor? In fact, devices only request service. Interrupts are implemented by a modified fetch-execute cycle that allows a processor to respond to a request. As Algorithm 16.1 explains, an interrupt occurs *between* the execution of two instructions.

Algorithm 16.1

```
Repeat forever {

        Test: if any device has requested interrupt, handle the interrupt, and then continue with the next iteration of the loop.

        Fetch: access the next step of the program from the location in which the program has been stored.

        Execute: Perform the step of the program.

}
```

Algorithm 16.1 A Fetch-Execute Cycle That Handles Interrupts.

16.12 Handling An Interrupt

To handle an interrupt, processor hardware takes the five steps that Figure 16.5 lists.

- Save the current execution state
- Determine which device interrupted
- Call the function that handles the device
- Clear the interrupt signal on the bus
- Restore the current execution state

Figure 16.5 Five steps that processor hardware performs to handle an interrupt. The steps are hidden from a programmer.

Saving and restoring state is easiest to understand: the hardware saves information when an interrupt occurs (usually in memory), and a special *return from interrupt* instruction reloads the saved state. In some architectures, the hardware saves complete state information, including the contents of all general-purpose registers. In other architectures, the hardware saves basic information, such as the instruction counter, and requires software to explicitly save and restore values, such as the general-purpose registers. In any case, saving and restoring state are symmetric operations — hardware is designed so the instruction that returns from an interrupt reloads exactly the same state information that the hardware saves when an interrupt occurs. We say that the processor temporarily *switches the execution context* when it handles an interrupt.

16.13 Interrupt Vectors

How does the processor know which device is interrupting? Several mechanisms have been used. For example, some architectures use a special-purpose coprocessor to handle all I/O. To start a device, the processor sends requests to the coprocessor. When a device needs service, the coprocessor detects the situation and interrupts the processor.

Most architectures use control signals on a bus to inform the processor when an interrupt is needed. The processor checks the bus on each iteration of the fetch-execute cycle. When it detects an interrupt request, interrupt hardware in the processor sends a special command over the bus to determine which device needs service. The bus is arranged so that exactly one device can respond at a time. Typically, each device is assigned a unique number, and the device responds by giving its number.

Numbers assigned to devices are not random. Instead, numbers are configured in a way that allows the processor hardware to interpret the number as an index into an array of pointers at a reserved location in memory. An item in the array, which is known as an *interrupt vector*, is a pointer to software that handles the device; we say that the interrupts are *vectored*. The software is known as an *interrupt handler*. Figure 16.6 illustrates the data structure.

The figure shows the simplest interrupt vector arrangement in which each physical device is assigned a unique interrupt vector. In practice, computer systems designed to accommodate many devices often use a variation in which multiple devices share a common interrupt vector. After the interrupt occurs, code in the interrupt handler uses the bus a second time to determine which physical device interrupted. Once it determines the physical device, the handler chooses an interaction that is appropriate for the device. The chief advantage of sharing an interrupt vector among multiple devices arises from scale — a processor with a fixed set of interrupt vectors can accommodate an arbitrary number of devices.

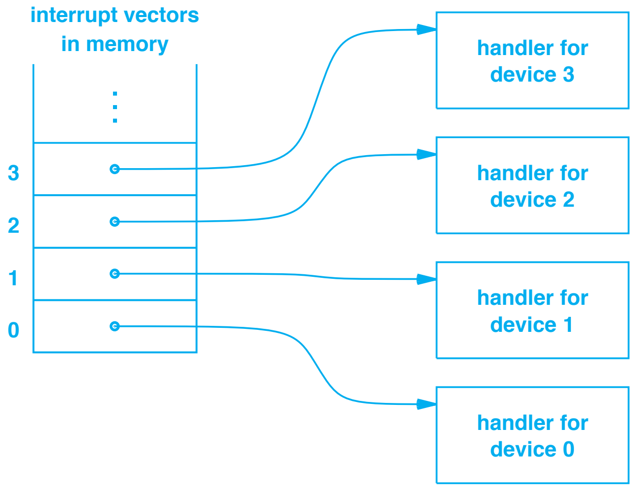


Figure 16.6 Illustration of interrupt vectors in memory. Each vector points to an interrupt handler for the device.

16.14 Interrupt Initialization And Disabled Interrupts

How are values installed in an interrupt vector table? Software must initialize interrupt vectors because neither the processor nor the device hardware enters or modifies the table. Instead, the hardware blindly assumes that the interrupt vector table has been initialized — when an interrupt occurs, the processor saves state, uses the bus to request a vector number, uses the value as an index into the table of vectors, and then branches to the code at that address. No matter what address is found in a vector, the processor will jump to the address and attempt to execute the instruction.

To ensure that no interrupts occur before the table has been initialized, most processors start in a mode that has interrupts *disabled*. That is, the processor continues to run the fetch-execute cycle without checking for interrupts. Later, once the software (usually the operating system) has initialized the interrupt vectors, the software must execute a special instruction that explicitly *enables* interrupts. In many processors, the interrupt status is controlled by the *mode* of the processor; interrupts are automatically enabled when the processor changes from the initial startup mode to a mode suitable for executing programs.

16.15 Interrupting An Interrupt Handler

Once an interrupt occurs and an interrupt handler is running, what happens if another device becomes ready and requests an interrupt? The simplest hardware follows a straightforward policy: once an interrupt occurs, further interrupts are automatically disabled until the current interrupt completes and returns. Thus, there is never any confusion.

The most sophisticated processors offer a *multiple level interrupt mechanism* which is also known as *multiple interrupt priorities*. Each device is assigned an interrupt priority level, typically in the range 1 through 7. At any given time, the processor is said to be operating at one of the priority levels. Priority zero means the processor is not currently handling an interrupt (i.e., is running an application); a priority N greater than zero means the processor is currently handling an interrupt from a device that has been assigned to level N.

The rule is:

> *When operating at priority level K, a processor can only be interrupted by a device that has been assigned to level K+1 or higher.*

Note that when an interrupt happens at priority K, no more interrupts can occur at priority K or lower. The consequence is that at most one interrupt can be in progress at each priority level.

16.16 Configuration Of Interrupts

We said that each device must be assigned an interrupt vector and (possibly) an interrupt priority. Both the hardware in the device and the software running on the processor must agree on the assignments — when a device returns an interrupt vector number, the corresponding interrupt vector must point to the handler for the device.

How are interrupt assignments made? Two approaches have been used:

- Manual assignment only used for small, embedded systems
- Automated assignment used on most computer systems

Manual Assignment. Some small embedded systems still use the method that was used on early computers: a manual approach in which computer owners configure both the hardware and software. For example, some devices are manufactured with physical switches on the circuit board, and the switches are used to enter an interrupt vector address. Of course, the operating system must be configured to match the values chosen for devices.

Automated Assignment. Automated interrupt vector assignment is the most widely used approach because it eliminates manual configuration and allows devices to be installed without requiring the hardware to be modified. When the computer boots, the processor uses the bus to determine which devices are attached. The processor assigns an interrupt vector number to each device, places a copy of the appropriate device handler software in memory, and builds the interrupt vector in memory. Of course, automated assignment means higher delay when booting the computer.

16.17 Dynamic Bus Connections And Pluggable Devices

Our description of buses and interrupt configuration has assumed that devices are attached to a bus while a computer is powered down, that interrupt vectors are assigned at startup, and that all devices remain in place as the computer operates. Early buses were indeed designed as we have described. However, more recent buses have been invented that permit devices to be connected and disconnected while the computer is running. We say that such buses support *pluggable* devices. For example, a *Universal Serial Bus* (*USB*) permits a user to plug in a device at any time.

How does a USB operate? In essence, a USB appears as a single device on the computer's main bus. When the computer boots, the USB is assigned an interrupt vector as usual, and a handler is placed in memory. Later, when a user plugs in a new device, the USB hardware generates an interrupt, and the processor executes the handler. The handler, in turn, sends a request over the USB bus to interrogate devices and determine which device has been attached. Once it identifies the device, the USB handler loads a secondary device-specific handler. When a device needs service, the device requests an interrupt. The USB handler receives the interrupt, determines which device interrupted, and passes control to the device-specific handler.

16.18 Interrupts, Performance, And Smart Devices

Why did the interrupt mechanism cause a revolution in computer architecture? The answer is easy. First, I/O is an important aspect of computing that must be optimized. Second, interrupt-driven I/O automatically overlaps computation and I/O without requiring a programmer to take any special action. That is, interrupts adapt to any speed processor and I/O devices automatically. Because a programmer does not need to estimate how many instructions can be performed during an I/O operation, interrupts never underestimate or overestimate. We can summarize:

> *A computer that uses interrupts is both easier to program and offers better overall performance than a computer that uses polling. In addition, interrupts allow any speed processor to adapt to any speed I/O devices automatically.*

Interestingly, once the basic interrupt mechanism had been invented, architects realized that further improvements are possible. To understand the improvements, consider a disk device. The underlying hardware requires several steps to read data from the disk and place it in memory. Figure 16.7 summarizes the steps.

- If disk is not spinning, bring it to full speed
- Compute the cylinder that contains the requested block and move the disk arm to the cylinder
- Wait for the disk to rotate to the correct sector
- Read bytes of data from a block on the disk and place them in a hardware FIFO
- Transfer bytes of data from the FIFO into memory

Figure 16.7 Example of the steps required to read a block from a disk device.

Early hardware required the processor to handle each step by starting the operation and waiting for an interrupt. For example, the processor had to verify that the disk was spinning. If the disk was idle, the processor had to issue a command that started the motor and wait for an interrupt.

The key insight is that the more digital logic an I/O device contains, the less the device needs to rely on the processor. Informally, architects use the term *dumb device* to refer to a device that requires a processor to handle each step and the term *smart device* to characterize a device that can perform a series of steps on its own. A smart version of a disk device contains sufficient logic (perhaps even an embedded processor) to handle all the steps involved in reading a block. Thus, a smart device does not interrupt as often, and does not require the processor to handle each step. Figure 16.8 lists an example interaction between a processor and a smart disk device.

- The processor uses the bus to send the disk a location in memory and request a *read* operation
- Disk device performs all steps required, including moving bytes into memory, and interrupts only after the operation completes

Figure 16.8 The interaction between a processor and a smart disk device when reading a disk block.

Our discussion of device interaction has omitted many details. For example, most I/O devices detect and report errors (e.g., a disk does not spin or a flaw on a surface prevents the hardware from reading a disk block). Thus, interrupt processing is more complex than described: when an interrupt occurs, the processor must interrogate the CSRs associated with the disk to determine whether the operation was successful or an error occurred. Furthermore, for devices that report *soft errors* (i.e., temporary errors), the processor must retry the operation to determine whether an error was temporary or permanent.

16.19 Direct Memory Access (DMA)

The discussion above implies that a smart I/O device can transfer data into memory without using the CPU. Indeed, such transfers are not only possible but key to high-speed I/O. The technology that allows an I/O device to interact with memory is known as *Direct Memory Access* (*DMA*).

To understand DMA, recall that in most architectures, both memory and I/O devices attach to a central bus. Thus, there is a direct path between an I/O device and memory. If we imagine that a smart I/O device contains an embedded processor, the idea behind DMA should be clear: the embedded processor in the I/O device issues *fetch* or *store* requests to which the memory responds. Of course, the bus design must make it possible for multiple processors (the main processor and an embedded processor in each smart device) to take turns sharing the bus and prevent them from sending multiple requests simultaneously. If the bus supports such a mechanism, an I/O device can transfer data between the memory and the device without using the processor.

To summarize:

> *A technology known as* Direct Memory Access (DMA) *allows a smart I/O device to access memory directly. DMA improves performance by allowing a device to transfer data between the device and memory without using the processor.*

16.20 Extending DMA With Buffer Chaining

It may seem that a smart device using DMA is sufficient to guarantee high performance: data can be transferred between the device and memory without using the processor, and the device does not interrupt for each step of the operation. However, an optimization has been discovered that further improves performance.

To understand how DMA can be improved, consider a high-speed network. Packets tend to arrive from the network in *bursts*, which means a set of packets arrives back-to-back with minimum time between successive packets. If the network interface device uses DMA, the device will interrupt the processor after accepting an incoming packet and placing the packet in memory. The processor must then specify the location of a buffer for the next packet and restart the device. The sequence of events must occur quickly (i.e., before the next packet arrives). Unfortunately, other devices on the system may also be generating interrupts, which means the processor may be delayed slightly. For the highest-speed networks, a processor may not be able to service an interrupt in time to capture the next packet.

To solve the problem of back-to-back arrivals, some smart I/O devices use a technique known as *buffer chaining*. The processor allocates multiple buffers, and creates a linked list in memory. The processor then passes the list to the I/O device, and allows the device to fill each buffer. Because a smart device can use the bus to read values from memory, the device can follow the linked list and place incoming packets in successive buffers. Figure 16.9 illustrates the concept†.

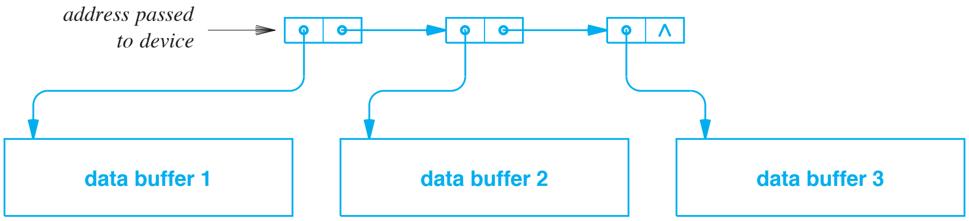


Figure 16.9 Illustration of buffer chaining. A processor passes a list of buffers to a smart I/O device, and the device fills each buffer on the list without waiting for the processor.

The network example given above describes the use of buffer chaining for high-speed input. A buffer chain can also be used with output: a processor places data in a set of buffers, links the buffers on a list, passes the address of the linked list to a smart I/O device, and starts the device. The device moves through the list, taking the data from each buffer in memory and sending the data to the device.

16.21 Scatter Read And Gather Write Operations

Buffer chaining is especially helpful for computer systems in which the buffer size used by software is smaller than the size of a data block used by an I/O device. On input, chained buffers allow a device to divide a large data transfer into a set of smaller buffers. On output, chained buffers allow a device to extract data from a set of small buffers and combine the data into a single block. For example, some operating systems create a network packet by placing the packet header in one buffer and the packet payload in another buffer. Buffer chaining allows the operating system to send the packet without the overhead of copying all the bytes into a single, large buffer.

We use the term *scatter read* to capture the idea of dividing a large block of incoming data into multiple small buffers, and the term *gather write* to capture the idea of combining data from multiple small buffers into a single output block. Of course, to make buffer chaining useful, a linked list of output buffers must specify the size of each buffer (i.e., the number of bytes to write). Similarly, a linked list of input buffers must include a length field that the device can set to specify how many bytes were deposited in the buffer.

†Although the figure shows three buffers, network devices typically use a chain of 32 or 64 buffers.

16.22 Operation Chaining

Although buffer chaining handles situations in which a given operation is repeated over many buffers, further optimization is possible in cases where a device can perform multiple operations. To understand, consider a disk device that offers *read* and *write* operations on individual blocks. To optimize performance, we need to start another operation as soon as the current operation completes. Unfortunately, the operations are a mixture of reads and writes.

The technology used to start a new operation without delay is known as *operation chaining*. Like buffer chaining, a processor that uses operation chaining must create a linked list in memory, and must pass the list to a smart device. Unlike buffer chaining, however, nodes on the linked list specify a complete operation: in addition to a buffer pointer, the node contains an operation and necessary parameters. For example, a node on the list used with a disk might specify a *read* operation and a disk block. Figure 16.10 illustrates operation chaining.

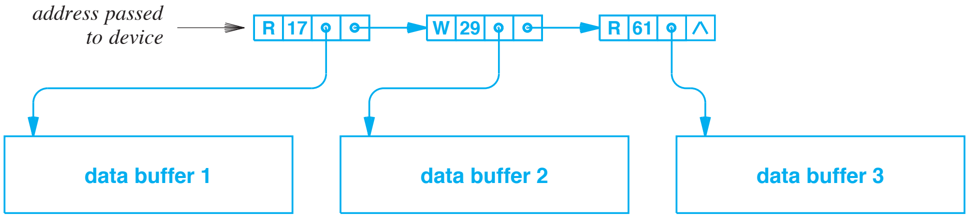


Figure 16.10 Illustration of operation chaining for a smart disk device. Each node specifies an operation (*R* or *W*), a disk block number, and a buffer in memory.

16.23 Summary

Two paradigms can be used to handle I/O devices: programmed I/O and interrupt-driven I/O. Programmed I/O requires a processor to handle each step of an operation by polling the device. Because a processor is much faster than an I/O device, the processor spends many cycles waiting for the device.

Third-generation computers introduced interrupt-driven I/O that allows a device to perform a complete operation before informing the processor. A processor that uses interrupts includes extra hardware that tests once during each execution of a fetch-execute cycle to see whether any device has requested an interrupt.

Interrupts are vectored, which means the interrupting device supplies a unique integer that the processor uses as an index into an array of pointers to handlers. To guarantee that interrupts do not affect a running program, the hardware saves and restores state information during an interrupt. Multilevel interrupts are used to give some devices priority over others.

Smart I/O devices contain additional logic that allows them to perform a series of steps without assistance from the processor. Smart devices use the techniques of buffer chaining and operation chaining to further optimize performance.

EXERCISES

16.1 Assume a RISC processor takes two microseconds to execute each instruction and an I/O device can wait at most 1 millisecond before its interrupt is serviced. What is the maximum number of instructions that can be executed with interrupts disabled?

16.2 List and explain the two I/O paradigms.

16.3 Expand the acronym CSR and explain what it means.

16.4 A software engineer is trying to debug a device driver, and discovers what appears to be an infinite loop:

```
while (*csrptr->tstbusy != 0)
        ;  /* do nothing*/
```

When the software engineer shows you the code, how do you respond?

16.5 Read about devices on a bus and the interrupt priorities assigned to each. Does a disk or mouse have higher priority? Why?

16.6 In most systems, part or all of the device driver code must be written in assembly language. Why?

16.7 Conceptually, what data structure is an interrupt vector, and what does one find in each entry of an interrupt vector?

16.8 What is the most significant advantage of a device that uses chained operations?

16.9 What is the chief advantage of interrupts over polling?

16.10 Suppose a user installs ten devices that all perform DMA into a single computer and attempts to operate the devices simultaneously. What components in the computer might become a bottleneck?

16.11 If a smart disk device uses DMA and blocks on the disk each contain 512 bytes, how many times will the disk interrupt when the processor transfers 2048 bytes (four separate blocks)?

16.12 When a device uses chaining, what is the type of the data structure that a device driver places in memory to give a set of commands to the device?

Chapter Contents

17

A Programmer's View Of Devices, I/O, And Buffering

17.1 Introduction

Previous chapters cover the hardware aspects of I/O. They explain the bus architecture that is used to interconnect devices, processors, and memory, as well as the interrupt mechanism that an external device uses to inform a processor when an operation completes.

This chapter changes the focus to software, and considers I/O from a programmer's perspective. The chapter examines both the software needed to control a device and the application software that uses I/O facilities. We will understand the important concept of a device driver, and see how a driver implements operations like *read* and *write*. We will learn that devices can be divided into two broad types: byte-oriented and block-oriented, and we will understand the interaction used with each.

Although few programmers write device drivers, understanding how a device driver operates and how low-level I/O occurs can help programmers write more efficient applications. Once we have looked at the mechanics of device drivers, we will focus on the concept of buffering, and see why it is essential for programmers to use buffering.

17.2 Definition Of A Device Driver

The previous chapter explains the basic hardware interrupt mechanism. We are now ready to consider how low-level software uses the interrupt mechanism to perform I/O operations. We use the term *device driver* to refer to software that provides an interface between an application program and an external hardware device. In most cases, a computer system has a device driver for each external device, and all applications that access a given device use the same driver. Typically, device drivers are part of the computer's operating system, which means any application running on the computer uses a device driver when communicating with a device.

Because a device driver understands the details of a particular hardware device, we say that a driver contains *low-level code*. The driver interacts with the device over a bus, understands the device's *Control And Status Registers* (*CSRs*), and handles interrupts from the device.

17.3 Device Independence, Encapsulation, And Hiding

The primary purpose of a device driver is *device independence*. That is, the device driver approach removes all hardware details from application programs and relegates them to a driver.

To understand why device independence is important, we need to know how early software was built. Each application program was designed for a specific brand of computer, a specific memory size, and a specific set of I/O devices. An application contained all the code needed to use the bus to communicate with particular devices. Unfortunately, a program written to use a specific set of devices could not be used with any other devices. For example, upgrading a printer to a newer model required all application programs to be rewritten.

A device driver solves the problem by providing a device-independent interface to applications. For example, because all applications that use a printer rely on the printer's device driver, an application does not have detailed knowledge of the hardware built in. Consequently, changing a printer only requires changing the device driver; all applications remain unchanged. We say that the device driver *hides* hardware details from applications or that the device driver *encapsulates* the hardware details.

To summarize:

> *A device driver consists of software that understands and handles all the low-level details of communication with a particular device. Because the device driver provides a high-level interface to applications, an application program does not need to change if a device changes.*

17.4 Conceptual Parts Of A Device Driver

A device driver contains multiple functions that all must work together, including code to communicate over a bus, code to handle device details, and code to interact with an application. Furthermore, a device driver must interact with the computer's operating system. To help manage complexity, programmers think of a device driver as partitioned into three parts:

- A *lower half* comprised of a handler that is invoked when an interrupt occurs
- An *upper half* comprised of functions that are invoked by applications to request I/O operations
- A set of *shared variables* that hold state information used to coordinate the two halves

The names *upper half* and *lower half* reflect the view of a programmer who writes device drivers: hardware is *low level* and application programs are *high level*. Thus, a programmer thinks of applications at the top of a hierarchy and hardware at the bottom. Figure 17.1 illustrates a programmer's view.

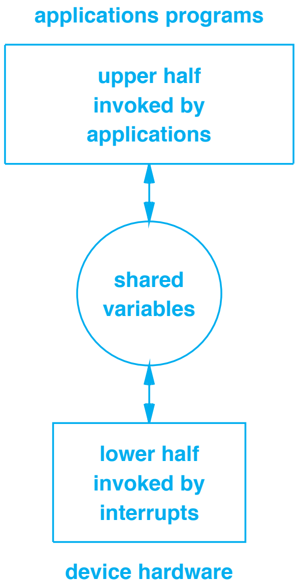

applications programs
upper half invoked by applications
shared variables
lower half invoked by interrupts
device hardware

Figure 17.1 The conceptual division of a device driver into three parts. A device driver provides the interface between applications that operate at a high level and the device hardware that operates at a low level.

17.5 Two Categories Of Devices

Before we can understand more about device drivers, we need to know more about the interface the hardware presents to the driver. Devices can be divided into two broad categories, depending on the style of interface the device uses:

- Character-oriented devices
- Block-oriented devices

A *character-oriented* device transfers a single byte of data at a time. For example, the serial interface used to connect a keyboard to a computer transfers one character (i.e., byte) for each keystroke. From a device driver's point of view, a character-oriented device generates an interrupt each time a character is sent or received — sending or receiving a block of N characters generates N interrupts.

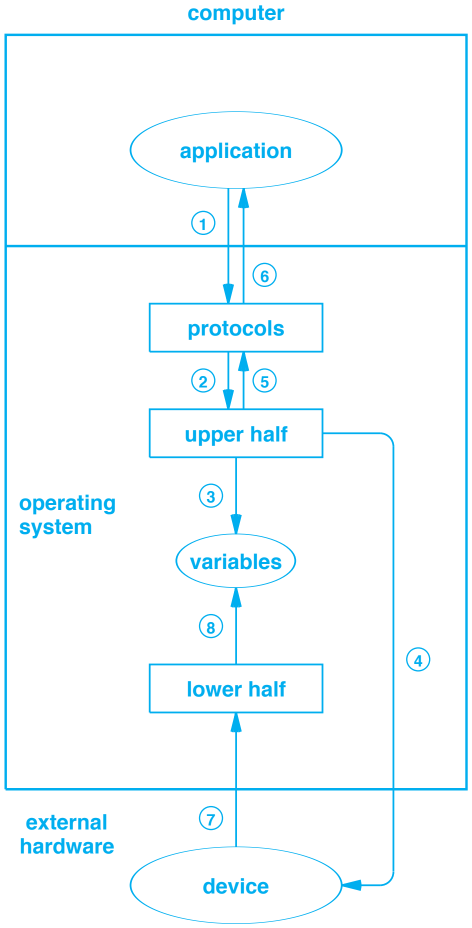
A *block-oriented* device transfers an entire block of data at a time. In some cases, the underlying hardware specifies a block size, B, and all blocks must contain exactly B bytes. For example, a disk device defines a block size equal to the disk's sector size. In other cases, however, blocks are of variable size. For example, a network interface defines a block to be as large as a packet (although it places an upper-bound on packet size, packet switching hardware allows packet sizes to vary from one packet to the next). From a device driver's point of view, a block-oriented device only generates one interrupt each time a block is sent or received.

17.6 Example Flow Through A Device Driver

The details of programming device drivers are beyond the scope of this text. However, to help us understand the concept, we will consider how a device driver might handle basic output. For our example, we will assume that an application sends data over the Internet. The application specifies data to be sent, and the protocol software creates a packet and transfers the packet to the device driver for the network device. Figure 17.2 illustrates the modules involved in a packet transfer, and lists the steps that are taken for output.

As the figure shows, even a straightforward operation requires a complex sequence of steps. When an application sends data, the application process enters the operating system and control passes to protocol software that creates a packet. The protocol software, in turn, passes the outgoing packet to the upper half of the appropriate device driver. The device driver places the packet in the shared variables section, starts the device performing packet transmission, and returns to the protocol software which returns to the application process.

Although control has returned from the operating system, the outgoing packet remains in the shared variables data area where the device can use DMA to access it. Once the device completes sending the packet, the device interrupts and control passes to the lower half. The lower half then removes the packet from the shared area.



Steps Taken

1. The application sends data over the Internet
2. Protocol software passes a packet to the driver
3. The driver stores the outgoing packet in the shared variables
4. The upper half specifies the packet location and starts the device
5. The upper half returns to the protocol module
6. The protocol software returns to the application
7. The device interrupts and the lower half of the driver executes
8. The lower half removes the copy of the packet from the shared variables

Figure 17.2 A simplified example of the steps that occur when an application requests an output operation. A device driver located in the operating system handles all communication with the device.

17.7 Queued Output Operations

Although the design used in our example driver is feasible, the approach is too inefficient to use in a production system. In particular, if our application sends another packet before the device has finished sending the first one, the device driver must poll until the device finishes using the packet. To avoid waiting, device drivers used in production systems implement a *queue of requests*. On output, the upper half does not wait for the device to be ready. Instead, the upper half deposits the data to be written in a queue, ensures that the device will generate an interrupt, and returns to the application. Later, when the device finishes its current operation and generates an interrupt, the lower half extracts the next request from the queue, starts the device, and returns from the interrupt. Figure 17.3 illustrates the conceptual organization.

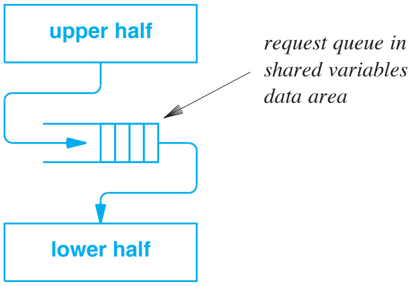


Figure 17.3 The conceptual organization of a device driver that uses a queue of requests. On output, the upper half deposits items in the request queue without waiting for the device, and the lower half controls the device.

A device driver that uses an output queue is elegant — the queue of requests provides coordination between the upper and lower halves of the driver. Figure 17.4 lists the steps that each half of a device driver takes for output.

Initialization (computer system starts)

1. Initialize input queue to empty

Upper half (application performs write)

1. Deposit data item in queue
2. Use the CSR to request an interrupt
3. Return to application

Lower half (interrupt occurs)

1. If the queue is empty, stop the device from interrupting
2. If the queue is nonempty, extract an item and start output
3. Return from interrupt

Figure 17.4 The steps that the upper and lower halves of a device driver take for an output operation when queueing is used. The upper half forces an interrupt, but does not start output on the device.

As the figure indicates, the steps for each half of the device driver are straightforward. Notice that the lower half performs most of the work: in addition to handling interrupts from the device, the lower half checks the queue and, if the queue is not empty, extracts the next item and starts the device. Because the device interrupts each time it completes an operation, the lower half will be invoked once per output operation, which allows it to start the next operation. Thus, the lower half will continue to be invoked until the queue is empty.

What happens after the last item has been removed from the queue? The lower half will be invoked after the last output operation completes, but will find the queue empty. At that point, the device will be idle. To prevent useless interrupts, the lower half controls the device to stop all interrupts. Later, when an application calls the upper half to place a new item in the queue, the upper half will start the device interrupting again, and output will proceed.

17.8 Forcing A Device To Interrupt

Because a request queue is used in so many device drivers, engineers have designed hardware that works well with the programming paradigm outlined in Figure 17.4. In particular, a device often includes a CSR bit that a processor can set to force the device to interrupt. Recall from Chapter 16 that the code required to set a CSR bit is trivial — it consists of a single assignment statement. Software does not need to check the current device status. Instead, the mechanism is designed so that setting the bit has no effect if the device is already active:

- A device has a CSR bit, B, that is used to force the device to interrupt
- If the device is idle, setting bit B causes the device to generate an interrupt
- If the device is currently performing an operation, setting bit B has no effect

In other words, if an interrupt is already destined to occur when the current operation completes, the device waits for the operation to complete and generates an interrupt as usual; if no operation is in progress, the device generates an interrupt immediately. The concept — arranging the hardware so that setting a CSR bit will not affect a busy device until the operation completes — greatly simplifies programming. To see why, look at the steps Figure 17.4 lists. The upper half does not need to test whether the device is busy (i.e., whether an operation is in progress). Instead, the upper half always sets the CSR bit. If an operation is already in progress, the device hardware ignores the bit being set, and waits until the operation completes. If the device is idle, setting the bit causes the device to interrupt immediately, which forces the lower half to select the next request in the queue and start the device.

17.9 Queued Input Operations

A device driver can also use queueing for input. However, additional coordination is required for two reasons. First, a device driver is configured to accept input before an application is ready to read the input (e.g., in case a user types ahead). Therefore, an input queue must be created when the device is initialized. Second, if input does not arrive before an application reads, the device driver must temporarily block the application until input does arrive. Figure 17.5 lists the steps a device driver uses to handle input when a queue is present.

Initialization (computer system starts)

1. Initialize input queue to empty
2. Force the device to interrupt

Upper half (application performs read)

1. If input queue is empty, temporarily stop the application
2. Extract the next item from the input queue
3. Return the item to the application

Lower half (interrupt occurs)

1. If the queue is not full, start another input operation
2. If an application is stopped, allow the application to run
3. Return from interrupt

Figure 17.5 The steps that the upper and lower halves of a device driver take for an input operation when queueing is used. The upper half temporarily stops an application until data becomes available.

Although our description of device drivers omits many details, it gives an accurate picture of the general approach that device drivers use. We can summarize:

> *A production device driver uses input and output queues to store items. The upper half places a request in the queue, and the lower half handles the details of communication with a device.*

17.10 Asynchronous Device Drivers And Mutual Exclusion

In Chapter 16, we said that an interrupt mechanism implies an *asynchronous programming model*. We can now understand why. Like a conventional program, polling is *synchronous* because control passes through the code from beginning to end. A device driver that handles interrupts is *asynchronous* because the programmer writes separate pieces of code that respond to events. One of the upper-half routines is invoked when an application requests I/O. A lower-half routine is invoked when an I/O operation occurs or when an interrupt occurs, and an initialization routine is invoked when a device is started.

Asynchronous programming is more challenging than synchronous programming. Because events can occur in any order, a programmer must use shared variables to encode the current state of the computation (i.e., the events that have occurred in the past and their effect). It can be difficult to test asynchronous programs because a programmer cannot easily control the sequence of events. More important, applications running on the processor and device hardware can generate events simultaneously. Simultaneous events make programming asynchronous device drivers especially difficult. For example, consider a smart device that uses command chaining. The processor creates a linked list of operations in memory, and the device follows the list and performs the operations automatically.

A programmer must coordinate the interaction between a processor and a smart device. To understand why, imagine a smart device extracting items from a list at the same time the upper half of a driver is adding items. A problem can occur if the smart device reaches the end of the list and stops processing just before the device driver adds a new item. Similarly, if two independent pieces of hardware attempt to manipulate pointers in the list simultaneously, links can become invalid.

To avoid errors caused by simultaneous access, a device driver that interacts with a smart device must implement *mutual exclusion*. That is, a device driver must ensure that the smart device will not access the list until changes have been completed, and the smart device must ensure that the device driver will not access the list until changes have been completed. A variety of schemes are used to ensure exclusive access. For example, some devices have special CSR values that the processor can set to temporarily stop the device from accessing the command list. Other systems have a facility that allows the processor to temporarily restrict use of the bus (if it cannot use the bus, a smart device cannot make changes to a list in memory). Finally, some processors offer *test-and-set* instructions that can be used to provide mutual exclusion.

17.11 I/O As Viewed By An Application

The sections above describe how a device driver is programmed. We said earlier that few programmers write device drivers. Thus, the details of CSR addresses, interrupt vectors, and request queues remain hidden from a typical programmer. The motivation for considering device drivers and low-level I/O is background: it helps us understand how to create applications that use low-level services efficiently.

Because they tend to use high-level languages, few programmers invoke low-level I/O facilities directly — to express I/O operations, the programmer uses *abstractions* that the programming language offers. For example, application programs seldom use a disk device. Instead, the programming language or the underlying system presents a programmer with a high-level *file* abstraction. Similarly, instead of exposing a programmer to display hardware, most systems present the programmer with a *window* abstraction.

The point is:

> *In many programming systems, I/O is hidden from the programmer. Instead of manipulating hardware devices, such as disks and display screens, a programmer only uses abstractions such as files and windows.*

Even in embedded systems that do allow application programmers to control I/O devices, the software is usually designed to hide as many details as possible from the programmer. In particular, an application can only specify generic, high-level I/O operations. When a compiler translates the program into a binary form for use on a specific computer, the compiler maps each high-level I/O operation into a sequence of low-level steps.

Interestingly, a typical compiler does not translate each I/O operation directly into a sequence of basic machine instructions. Instead, the compiler generates code that invokes library functions to perform I/O operations. Therefore, before it can be executed, the program must be combined with the appropriate library functions.

We use the term *run-time library* to refer to the set of library functions that accompany a compiled program. Of course, the compiler and run-time library must be designed to work together — the compiler must know which functions are available, the exact arguments used by each function, and the meaning of the function.

> *Application programmers seldom interact with device drivers directly. Instead, they rely on a run-time library to act as an intermediary.*

The chief advantage of using a run-time library as an intermediary arises from the flexibility and ease of change. Only the run-time library functions understand how to use the underlying I/O mechanisms (i.e., the device drivers). If the I/O hardware and/or the device drivers change, only the run-time library needs to be updated — the compiler can remain unchanged. In fact, separating a run-time library from a compiler allows code to be compiled once and then combined with various run-time libraries to produce images for more than one version of an operating system.

17.12 The Library/Operating System Dichotomy

We know that a device driver resides in the operating system and the run-time library functions that an application uses to perform I/O reside outside the operating system (because they are linked with the application). Conceptually, we imagine three layers of software on top of the device hardware as Figure 17.6 illustrates.

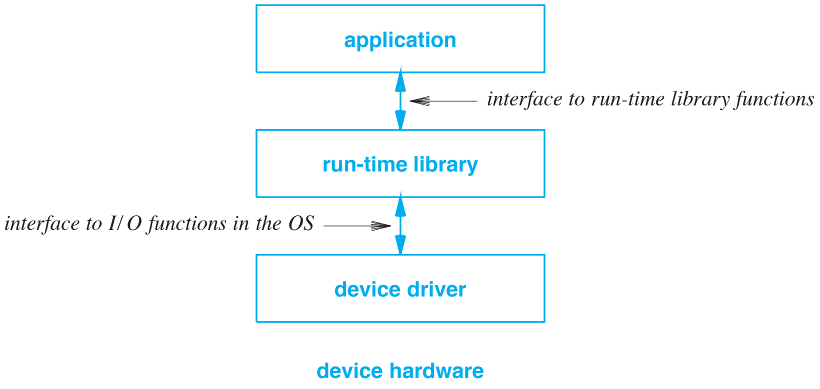

application
interface to run-time library functions
run-time library
interface to I/O functions in the OS
device driver
device hardware

Figure 17.6 The conceptual arrangement of application code, run-time library code, and a device driver with interfaces labeled.

Several questions arise. What services does each layer of software provide? What is the interface between an application and the run-time library, or the interface between the run-time library and the operating system? What are the relative costs of using the two interfaces?

17.13 I/O Operations That The OS Supports

We begin by examining the interface between the run-time library and the operating system. In a low-level programming language such as C, the operating system interface is directly available to applications. Thus, a programmer can choose to use an I/O library or make operating system calls directly†.

Although the exact details of I/O operations depend on the operating system, a general approach has become popular. Known as the *open/read/write/close* paradigm, the approach offers six basic functions. Figure 17.7 lists the functions with the names used by the Unix operating system.

Operation	Meaning
open	Prepare a device for use (e.g., power up)
read	Transfer data from the device to the application
write	Transfer data from the application to the device
close	Terminate use of the device
seek	Move to a new location of data on the device
ioctl	Miscellaneous control functions (e.g., change volume)

Figure 17.7 Six basic I/O functions that comprise the open/read/write/close paradigm. The names are taken from the Unix operating system.

†A later section discusses the standard I/O library used with C.

As an example, consider a device that can read or write a *Digital Video Disk* (*DVD*). The *open* function can be used to start the drive motor and ensure that a disc has been inserted. Once the drive has been started, the *read* function can be used to read data from the disc, and the *write* function can be used to write data onto the disc. The *seek* function can be used to move to a new position (e.g., a specific video segment), and the *close* function can be used to power down the disc. Finally, the *ioctl* function (an abbreviation of I/O control) can be used for all other functions (e.g., the eject function).

Of course, each of the operations takes arguments that specify details. For example, a *write* operation needs arguments that specify the device to use, the location of data, and the amount of data to write. More important, the device driver must understand how to map each operation and arguments to operations on the underlying device. For example, when the driver receives a *control* operation, such as an eject, the driver must know how to implement the operation with the device hardware (e.g., how to assign values to the device’s CSR registers).

17.14 The Cost Of I/O Operations

When an application program invokes a function in the run-time library, the cost is exactly the same as calling a function because a copy of the code for the library function is incorporated into the application when the program is built. Thus, the cost of invoking library functions is relatively low.

When an application program or a run-time library function invokes an I/O operation such as *read* or *write*, however, control must pass through a *system call*† to the appropriate device driver in the operating system. Unfortunately, invoking an operating system function through a system call incurs extremely high overhead. There are three reasons. First, the processor must change privilege mode because the operating system runs with greater privilege than an application. Second, the processor must change the address space from the application’s virtual address space to the operating system’s address space. Third, the processor must copy data between the application’s address space and the operating system’s address space.

We can summarize:

> *The overhead involved in using a system call to communicate with a device driver is high; a system call is much more expensive than a conventional function call, such as the call used to invoke a library function.*

More important, much of the system call overhead is associated with making the call rather than the work performed by the driver. Therefore, to optimize performance, programmers seek ways to minimize the number of system calls.

†Some computer architectures use the term *trap* in place of system call.

17.15 Reducing System Call Overhead

To understand how we can reduce the overhead of system calls, consider a worst-case example. Suppose an application needs to print a document, and suppose printing requires the application to send a total of N bytes of data to the printer. The highest cost occurs if the application makes a separate system call to transfer each byte of data because the application will make a total of N system calls. As an alternative, if the application generates a complete line of text and then makes a system call to transfer the entire line, the overhead is reduced from N system calls to L system calls, where L is the number of lines in the document (i.e., $L < N$).

Can we further reduce the overhead of printing a document? Yes, we can. The application can be redesigned to allocate enough memory to hold an entire page of the document, generate the page, and then make one system call to transfer the entire page to the device driver. The result is an application that only makes P system calls, where P is the number of pages in the document (presumably $P << N$).

A general principle can be stated:

> *To reduce overhead and optimize I/O performance, a programmer must reduce the number of system calls that an application invokes. The key to reducing system calls involves transferring more data per system call.*

Of course, it is not always possible to reduce the number of system calls used for I/O. For example, an application like a text editor or email composer displays characters as the user enters them. The application cannot wait until the user enters an entire line of text or an entire page because each character must appear on the screen immediately. Similarly, input from a keyboard often requires a program to accept one character at a time without waiting for a user to enter an entire line or page. Fortunately, such applications often involve user interaction in which I/O is relatively slow, so optimization is unimportant.

17.16 The Key Concept Of Buffering

The above discussion shows that an application programmer can optimize I/O performance by rewriting code in such a way that the number of systems calls is lower. The optimization is so important for high-speed I/O that it has been incorporated into most computer software. Instead of requiring a programmer to rewrite code, I/O run-time libraries have been designed to handle the optimization automatically.

We use the term *buffering* to describe the concept of accumulating data before an I/O transfer, and the term *buffer* to refer to the area of memory in which the data is placed.

> *The buffering principle: to reduce the number of system calls, accumulate data in a buffer, and transfer a large amount of data each time a system call is made.*

To automate buffering, library routines need a scheme that works for any application. Thus, instead of lines or pages, library functions use a fixed-size buffer. To take advantage of buffering, an application must call library functions instead of the operating system. In the case of a programming language that contains built-in I/O facilities, the run-time library implements buffering, and the compiler generates code that invokes the appropriate library routines; in the case of a programming language that does not have built-in I/O facilities, the programmer must call buffering library routines instead of system calls.

Library routines that implement buffering usually provide the five conceptual operations that Figure 17.8 lists.

Operation	Meaning
setup	Initialize the buffer
input	Perform an input operation
output	Perform an output operation
terminate	Discontinue use of the buffer
flush	Force contents of buffer to be written

Figure 17.8 The conceptual operations provided by a typical library that offers buffered I/O.

The operations listed in the figure are analogous to those that an operating system offers as an interface to a device. In fact, we will see that at least one implementation of a buffered I/O library uses function names that are variants of *open*, *read*, *write*, and *close*. Figure 17.8 uses alternate terminology to help clarify the distinction.

17.17 Implementation of Buffered Output

To understand how buffering works, consider how an application uses the buffered output functions in Figure 17.8. When it begins, the application calls a *setup* function to initialize buffering. Some implementations provide an argument that allows the application to specify a buffer size; in other implementations, the buffer size is a constant†. In any case, we will assume *setup* allocates a buffer, and initializes the buffer to empty. Once the buffer has been initialized, the application can call the *output* function to transfer data. On each call, the application supplies one or more bytes of data. Finally, when it finishes transferring data, the application calls the *terminate* function. (Note: a later section describes the use of function *flush*).

†Typical buffer sizes range from 8 Kbytes to 128 Kbytes, depending on the computer system.

The amount of code required to implement buffered output is trivial. Figure 17.9 describes the steps used to implement each output function. In a language such as C, each step can be implemented with one or two lines of code.

The motivation for a *terminate* function should now be clear: because output is buffered, the buffer may be partially full when the application finishes. Therefore, the application must force the remaining contents of the buffer to be written.

Setup(N)

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to the address of the first byte of the buffer.

Output(D)

1. Place data byte D in the buffer at the position given by pointer p, and move p to the next byte.
2. If the buffer is full, make a system call to write the contents of the entire buffer, and reset pointer p to the start of the buffer.

Terminate

1. If the buffer is not empty, make a system call to write the contents of the buffer prior to pointer p.
2. If the buffer was allocated dynamically, deallocate it.

Figure 17.9 The steps taken to achieve buffered output.

17.18 Flushing A Buffer

It may seem that output buffering cannot be used with some applications. For example, consider an application that allows two users to communicate over a computer network. When it emits a message, an application assumes the message will be transmitted and delivered to the other end. Unfortunately, if buffering is used, the message may wait in the buffer unsent.

Of course, a programmer can rewrite an application to buffer data internally and make system calls directly. However, designers of general-purpose buffering libraries have devised a way to permit applications that use buffered I/O to specify when a system call is needed. The mechanism consists of the *flush* function that an application can call to force data to be sent even if the buffer is not full. Programmers use the phrase *flushing a buffer* to describe the process of forcing output of a partially full buffer. If a buffer is empty when an application calls *flush*, the call has no effect. If the buffer contains data, however, the *flush* function makes a system call to write the data, and then resets the global pointer to indicate that the buffer is empty. Figure 17.10 lists the steps of a *flush* operation.

Flush

1. If the buffer is currently empty, return to the caller without taking any action.
2. If the buffer is not currently empty, make a system call to write the contents of the buffer and set the global pointer p to the address of the first byte of the buffer.

Figure 17.10 The steps required to implement a *flush* function in a buffered I/O library. *Flush* allows an application to force data to be written before the buffer is full.

Look back at the implementation of the *terminate* function given in Figure 17.9. If the library offers a *flush* function, the first step of *terminate* can be replaced by a call to the *flush* function.

To summarize:

> *A programmer uses a* flush *function to specify that outgoing data in a buffer should be sent even if the buffer is not full. A* flush *operation has no effect if a buffer is currently empty.*

17.19 Buffering On Input

The descriptions above explain how buffering can be used with output. In many cases, buffering can also be used to reduce the overhead on input. To understand how, consider reading data sequentially. If an application reads N bytes of data, one byte at a time, the application will make N system calls.

Assuming the underlying device allows transfer of more than one byte of data, buffering can be used to reduce the number of system calls. The application (or the run-time library) allocates a large buffer, makes one system call to fill the buffer, and then satisfies subsequent requests from the buffer. Figure 17.11 lists the steps required. As with output buffering, the implementation is straightforward. In a language such as C, each step can be implemented with a trivial amount of code.

17.20 Effectiveness Of Buffering

Why is buffering so important? Because even a small buffer can have a large effect on I/O performance. To see why, observe that when buffered I/O is used, a system call is only needed once per buffer†. As a result, a buffer of N bytes reduces the number of system calls by a factor of N. Thus, if an application makes S system calls, a buffer of only 8 K bytes reduces the number of system calls to S/8192.

†Our analysis ignores situations in which an application calls *flush* frequently.

Setup(N)

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to indicate that the buffer is empty.

Input(N)

1. If the buffer is empty, make a system call to fill the entire buffer, and set pointer p to the start of the buffer.
2. Extract a byte, D, from the position in the buffer given by pointer p, move p to the next byte, and return D to the caller.

Terminate

1. If the buffer was dynamically allocated, deallocate it.

Figure 17.11 The steps required to achieve buffered input.

Buffering is not limited to run-time libraries. The technique is so important that device drivers often implement buffering. For example, in some disk drivers, the driver maintains a copy of the disk block in memory, and allows an application to read or write data from the block. Of course, buffering in an operating system does not eliminate system calls. However, such buffering does improve performance because external data transfers are slower than system calls. The important point is that buffering can be used to reduce I/O overhead whenever a less expensive operation can be substituted for an expensive operation.

We can summarize the importance of buffering:

> *Using a buffer of N bytes can reduce the number of calls to the underlying system by a factor of N. A large buffer can mean the difference between an I/O mechanism that is fast and one that is intolerably slow.*

17.21 Relationship To Caching

Buffering is closely related to the concept of caching that is described in Chapter 12. The chief difference arises from the way items are accessed: a cache system is optimized to accommodate random access, and a buffering system is optimized for sequential access.

In essence, a cache stores items that *have been* referenced, and a buffer stores items that *will be* referenced (assuming sequential references). Thus, in a virtual memory system, a cache stores entire pages of memory — when any byte on the page is referenced, the entire page is placed in the cache. In contrast, a buffer stores sequential bytes. Thus, when a byte is referenced, a buffering system preloads the next bytes — if the referenced byte lies at the end of a page, the buffering system preloads bytes from the next page.

17.22 An Example: The C Standard I/O Library

One of the best-known examples of a buffering I/O library was created for the C programming language and the Unix operating system. Known as the *standard I/O library* (*stdio*), the library supports both input and output buffering. Figure 17.12 lists a few of the functions found in the Unix standard I/O library along with their purpose.

Function	Meaning
fopen	Set up a buffer
fgetc	Buffered input of one byte
fread	Buffered input of multiple bytes
fwrite	Buffered output of multiple bytes
fprintf	Buffered output of formatted data
fflush	Flush operation for buffered output
fclose	Terminate use of a buffer

Figure 17.12 Examples of functions included in the standard I/O library used with the Unix operating system. The library includes additional functions not listed here.

17.23 Summary

Two aspects of I/O are pertinent to programmers. A systems programmer who writes device driver code must understand the low-level details of the device, and an application programmer who uses I/O facilities must understand the relative costs.

A device driver is divided into three parts: an upper half that interacts with application programs, a lower half that interacts with the device itself, and a set of shared variables. A function in the upper half receives control when an application reads or writes data; the lower half receives control when the device generates an input or output interrupt.

The fundamental technique programmers use to optimize sequential I/O performance is known as *buffering*. Buffering can be used for both input and output, and is often implemented in a run-time library. Because it gives an application control over when data is transferred, a *flush* operation allows buffering to be used with arbitrary applications.

Buffering reduces system call overhead by transferring more data per system call. Buffering provides significant performance improvement because a buffer of N bytes reduces the number of system calls that an application makes by a factor of N.

EXERCISES

17.1 What does a device driver provide, and how do device drivers make it easier to write applications?

17.2 Name the three conceptual parts of a device driver and state how each is used.

17.3 Explain the use of an output queue in a device driver by describing how and when items are inserted in the queue, as well as how and when they are removed.

17.4 A user invokes an app that writes a file. The app displays a progress bar that shows how much of the file has been written. Just as the progress bar reaches 50%, the battery fails and the device crashes. When the user reboots the device, he or she discovers that less than 20% of the file has actually been written. Explain why the app reported writing 50%.

17.5 When a program calls *fputc*, what does the program invoke?

17.6 What is a *flush* operation, and why is it needed?

17.7 To increase the performance of an app, a programmer rewrites the app so that instead of reading one byte at a time, the app reads eight thousand bytes and then processes them. What technique is the programmer using?

17.8 Compare the time needed to copy a large file using *write* and *fwrite*.

17.9 The standard I/O function *fseek* allows random access. Measure the difference in the time required to use *fseek* within a small region of a file and within a large region.

17.10 Build an output buffering routine, *bufputc*, that accepts as an argument a character to be printed. On each call to *bufputc*, store the character in a buffer, and call *write* once for the entire buffer. Compare the performance of your buffered routine to a program that uses *write* for each character.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Part V

Advanced Topics

The Fundamental Concepts
Of
Parallelism And Pipelining

Chapter Contents

18

Parallelism

18.1 Introduction

Previous chapters cover the three key components of computer architecture: processors, memory systems, and I/O. This chapter begins a discussion of fundamental concepts that cross the boundaries among architectural components.

The chapter focuses on the use of parallel hardware, and shows that parallelism can be used throughout computer systems to increase speed. The chapter introduces terminology and concepts, presents a taxonomy of parallel architectures, and examines computer systems in which parallelism is the fundamental paradigm around which the entire system is designed. Finally, the chapter discusses limitations and problems with parallel architectures.

The next chapter extends the discussion by examining a second fundamental technique: pipelining. We will see that both parallelism and pipelining are important in high-speed designs.

18.2 Parallel And Pipelined Architectures

Some computer architects assert that there are only two fundamental techniques used to increase hardware speed: *parallelism* and *pipelining*. We have already encountered examples of each technique, and have seen how they can be used.

Other architects take a broader view of parallelism and pipelining, using the techniques as the fundamental basis around which a system is designed. In many cases, the architecture is so completely dominated by one of the two techniques that the resulting system is informally called a *parallel computer* or a *pipelined computer*.

18.3 Characterizations Of Parallelism

Rather than classify an architecture as *parallel* or *nonparallel*, computer architects use a variety of terms to characterize the type and amount of parallelism that is present in a given design. In many cases, the terminology describes the possible extremes for a type of parallelism. We can classify an architecture by stating where the architecture lies between the two extremes. Figure 18.1 lists the key characterizations using nomenclature proposed by Michael J. Flynn in a classic paper†. Later sections explain each of the terms and give examples.

- Microscopic vs. macroscopic
- Symmetric vs. asymmetric
- Fine-grain vs. coarse-grain
- Explicit vs. implicit

Figure 18.1 Terminology used to characterize the amount and type of parallelism present in a computer architecture.

18.4 Microscopic Vs. Macroscopic

Parallelism is fundamental; an architect cannot design a computer without thinking about parallel hardware. Interestingly, the pervasiveness of parallelism means that unless a computer uses an unusual amount of parallel hardware, we typically do not discuss the parallel aspects. To capture the idea that much of the parallelism in a computer remains hidden inside subcomponents, we use the term *microscopic parallelism*. Like microbes in the world around us, microscopic parallelism is present, but does not stand out without closer inspection.

The point is:

> *Parallelism is so fundamental that virtually all computer systems contain some form of parallel hardware. We use the term* microscopic parallelism *to characterize parallel facilities that are present, but not especially visible.*

To be more precise, we say that *microscopic parallelism* refers to the use of parallel hardware within a specific component (e.g., inside a processor or inside an ALU), whereas *macroscopic parallelism* refers to the use of parallelism as a basic premise around which a system is designed.

†M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, C-21(9):948--960, September 1972.

18.5 Examples Of Microscopic Parallelism

In earlier chapters, we have seen examples of using *microscopic parallelism* within processors, memory systems, and I/O subsystems. The following paragraphs highlight a few examples.

ALU. An Arithmetic Logic Unit handles logical and arithmetic operations. Most ALUs perform integer arithmetic by processing multiple bits in parallel. Thus, an ALU that is designed to operate on integers contains parallel hardware that allows the ALU to compute a Boolean function on a pair of thirty-two bit values in a single operation. The alternative consists of an ALU that processes one bit at a time, an approach that is known as *bit serial processing*. It should be easy to see that bit serial processing takes much longer than computing bits in parallel. Therefore, bit serial arithmetic is reserved for special cases.

Registers. The general-purpose registers in a CPU make heavy use of microscopic parallelism. Each bit in a register is implemented by a separate digital circuit (specifically, a latch). Furthermore, to guarantee the highest-speed computation, parallel data paths are used to move data between general-purpose registers and the ALU.

Physical Memory. As another example of microscopic parallelism, recall that a physical memory system uses parallel hardware to implement *fetch* and *store* operations — the hardware is designed to transfer an entire word on each operation. As in an ALU, microscopic parallelism increases memory speed dramatically. For example, a memory system that implements sixty-four bit words can access or store approximately sixty-four times as much data in the same time as a memory system that accesses a single bit at a time.

Parallel Bus Architecture. As we have seen, the central bus in a computer usually uses parallel hardware to achieve high-speed transfers among the processor, memory, and I/O devices. A typical modern computer has a bus that is either thirty-two- or sixty-four-bits wide, which means that either thirty-two or sixty-four bits of data can be transferred across the bus in a single step.

18.6 Examples Of Macroscopic Parallelism

As the examples in the previous section demonstrate, microscopic parallelism is essential for high-speed performance — without parallel hardware, various components of a computer system cannot operate at high speed. Computer architects are aware that the global architecture often has a greater impact on overall system performance than the performance of any single subsystem. That is, adding more parallelism to a single subsystem may not improve the overall system performance†.

To achieve the greatest impact, parallelism must span multiple components of a system — instead of merely using parallelism to improve the performance of a single component, the system must allow multiple components to work together. We use the

†Chapter 21 discusses performance in more detail.

term *macroscopic parallelism* to characterize the use of parallelism across multiple, large-scale components of a computer system. A few examples will clarify the concept.

Multiple, Identical Processors. Systems that employ macroscopic parallelism usually employ multiple processors in one form or another. For example, some PCs are advertised as *dual core* or *quad core* computers, meaning that the PC contains two or four copies of the processor on a single chip. The chip is arranged to allow both processors to operate at the same time. The hardware does not control exactly how the cores are used. Instead, the operating system assigns code to each core. For example, the operating system can assign one core the task of handling I/O (i.e., running device drivers), and assign other cores application programs to run.

Multiple, Dissimilar Processors. Another example of macroscopic parallelism arises in systems that make extensive use of special-purpose coprocessors. For example, a computer optimized for high-speed graphics might have four displays attached, with a special graphics processor running each display. A graphics processor, typically found on an interface card, does not use the same architecture as a CPU because the graphics processor needs instructions optimized for graphics operations.

18.7 Symmetric Vs. Asymmetric

We use the term *symmetric parallelism* to characterize a design that uses replications of identical elements, usually processors or cores, that can operate simultaneously. For example, the multicore processors mentioned above are said to be symmetric because all cores are identical.

The alternative to a symmetric parallel design is a parallel design that is *asymmetric*. As the name implies, an asymmetric design contains multiple elements that function at the same time, but differ from one another. For example, a PC with a CPU, a graphics coprocessor, a math coprocessor, and an I/O coprocessor is classified as using asymmetric parallelism because the four processors can operate simultaneously, but differ from one another internally†.

18.8 Fine-grain Vs. Coarse-grain Parallelism

We use the term *fine-grain parallelism* to refer to computers that provide parallelism on the level of individual instructions or individual data elements, and the term *coarse-grain parallelism* to refer to computers that provide parallelism on the level of programs or large blocks of data. For example, a graphics processor that uses sixteen parallel hardware units to update sixteen bytes of an image at the same time is said to use fine-grain parallelism. In contrast, a dual core PC that uses one core to print a document while another core composes an email message is described as using coarse-grain parallelism.

†Some architects also apply the term *asymmetric* to a multicore design if the cores do not have the same access to memory and I/O devices.

18.9 Explicit Vs. Implicit Parallelism

An architecture in which the hardware handles parallelism automatically without requiring a programmer to initiate or control parallel execution is said to offer *implicit parallelism*, and an architecture in which a programmer must control each parallel unit is said to offer *explicit parallelism*. We will consider the advantages and disadvantages of explicit and implicit parallelism later.

18.10 Types Of Parallel Architectures (Flynn Classification)

Although many systems contain multiple processors of one type or another, the term *parallel architecture* is usually reserved for designs that permit arbitrary *scaling*. That is, when they refer to a parallel architecture, architects usually mean a design in which the number of processors can be arbitrarily large (or at least reasonably large). As an example, consider a computer that can have either one or two processors. Although adding a second processor increases parallelism, such an architecture is usually classified as a *dual-processor computer* rather than a parallel architecture. Similarly, a PC with four cores is classified as a *quad-core PC*. However, a cluster of thirty-two interconnected PCs that can scale to one thousand twenty-four PCs is classified as a parallel architecture.

The easiest way to understand parallel architectures is to divide the architectures into broad groups, where each group represents a type of parallelism. Of course, no division is absolute — most practical computer systems are hybrids that contain facilities from more than one group. Nevertheless, we use the classification to define basic concepts and nomenclature that allow us to discuss and characterize the systems.

A popular way to describe parallelism that is attributed to Flynn considers whether processing or data is replicated. Known as the *Flynn classification*, the system focuses on whether the computer has multiple, independent processors each running a separate program or a single program being applied to multiple data items. Figure 18.2 lists terms used by the Flynn classification to define types of parallelism; the next sections explain the terminology and give examples.

Name	Meaning
SISD	Single Instruction stream Single Data stream
SIMD	Single Instruction stream Multiple Data streams
MISD	Multiple Instruction streams Single Data stream
MIMD	Multiple Instruction streams Multiple Data streams

Figure 18.2 Terminology used by the Flynn classification to characterize parallel computers†.

†MISD is a specialized category that is reserved for unusual hardware, such as the pipeline architecture shown in Figure 19.5 on page 387 that executes multiple instructions on a single piece of data or a redundant processor used to increase reliability.

18.11 Single Instruction Single Data (SISD)

The phrase *Single Instruction Single Data stream* (*SISD*) is used to describe an architecture that does not support macroscopic parallelism. The term *sequential architecture* or *uniprocessor architecture* is often used in place of SISD to emphasize that the architecture is not parallel. In essence, SISD refers to a conventional (i.e., Von Neumann) architecture — the processor runs a standard fetch-execute cycle and performs one operation at a time. The term refers to the idea that a single, conventional processor is executing instructions that each operate on a single data item. That is, unlike a parallel architecture, a conventional processor can only execute one instruction at any time, and each instruction refers to a single computation.

Of course, we have seen that an SISD computer can use parallelism internally. For example, the ALU may be able to perform operations on multiple bits in parallel, the CPU may invoke a coprocessor, or the CPU may have mechanisms that allow it to fetch operands from two banks of memory at the same time. However, the overall effect of an SISD architecture is sequential execution of instructions that each operate on one data item.

18.12 Single Instruction Multiple Data (SIMD)

The phrase *Single Instruction Multiple Data streams* (*SIMD*) is used to describe a parallel architecture in which each instruction specifies a single operation (e.g., integer addition), but the instruction is applied to many data items at the same time. Typically, an SIMD computer has sufficient hardware to handle sixty-four simultaneous operations (e.g., sixty-four simultaneous additions).

Vector Processors. An SIMD architecture is not useful for applications such as word processing or email. Instead, SIMD is used with applications that apply the same operation to a set of values. For example, graphics applications and some scientific applications work well on an SIMD architecture that can apply an operation to a large set of values. The architecture is sometimes called a *vector processor* or an *array processor* after the mathematical concept of vectors and the computing concept of arrays.

As an example of how an SIMD machine works, consider normalizing the values in a vector, *V*, that contains *N* elements. Normalization requires that each item in the vector be multiplied by a floating point number, *Q*. On a sequential architecture (i.e., an SISD architecture), the algorithm required to normalize the vector consists of a loop as Figure 18.3 shows.

```
for i from 1 to N {
        V[i] ← V[i] × Q;
}
```

Figure 18.3 A sequential algorithm for vector normalization.

On an SIMD architecture, the underlying hardware can apply an arithmetic operation to all the values in an array simultaneously (assuming the size of the array does not exceed the parallelism in the hardware). For example, in a single step, hardware that has sixty-four parallel units can multiply each value in an array of sixty-four elements by a constant. Thus, the algorithm to perform normalization of an array on an SIMD computer takes one step:

$$V \leftarrow V \times Q;$$

Of course, if vector V is larger than the hardware capacity, multiple steps will be required. The important point is that a vector instruction on an SIMD architecture is not merely a shorthand for a loop. Instead, the underlying system contains multiple hardware units that operate in parallel to provide substantial speedup; the performance improvement can be significant, especially for computations that use large matrices.

Of course, not all instructions in an SIMD architecture can be applied to an array of values. Instead, an architect identifies a subset of operations to be used with vectors, and defines a special *vector instruction* for each. For example, normalization of an entire array is only possible if the architect chooses to include a vector multiplication instruction that multiplies each value in the vector by a constant.

In addition to operations that use a constant and a vector, SIMD computers usually provide instructions that use two vectors. That is, a vector instruction takes one or more operands that each specify a vector. For example, SIMD architectures are used for problems involving matrix multiplication. On most SIMD machines, an operand that specifies a vector gives two pieces of information: the location of the vector in memory and an integer that specifies the size of the vector (i.e., number of items in the vector). On some machines, vector instructions are controlled by special-purpose registers — the address and size of each vector are loaded into registers before a vector instruction is invoked. In any case, software determines the number of items in a vector up to the maximum size supported by the hardware†.

Graphics Processors. SIMD architectures are also popular for use with graphics. To understand why, it is important to know that typical graphics hardware uses sequential bytes in memory to store values for pixels on a screen. For example, consider a video game in which foreground figures move while a background scene stays in place. Game software must copy the bytes that correspond to the foreground figure from one location in memory to another. A sequential architecture requires a programmer to specify a loop that copies one byte at a time. On an SIMD architecture, however, a programmer can specify a vector size, and then issue a single *copy* command. The underlying SIMD hardware then copies multiple bytes simultaneously.

†An exercise considers speedup in cases where vectors exceed the capacity of the hardware; a definition of *speedup* can be found in Section 18.15.

18.13 Multiple Instructions Multiple Data (MIMD)

The phrase *Multiple Instructions Multiple Data streams* (*MIMD*) is used to describe a parallel architecture in which each of the processors performs independent computations at the same time. Although many computers contain multiple internal processing units, the MIMD designation is reserved for computers in which the processors are visible to a programmer. That is, an MIMD computer can run multiple, independent programs at the same time.

Symmetric Multiprocessor (SMP). The most well-known example of an MIMD architecture consists of a computer known as a *Symmetric Multiprocessor* (*SMP*). An SMP contains a set of *N* processors (or *N* cores) that can each be used to run programs. In a typical SMP design, the processors are identical: they each have the same instruction set, operate at the same clock rate, have access to the same memory modules, and have access to the same external devices. Thus, any processor can perform exactly the same computation as any other processor. Figure 18.4 illustrates the concept.

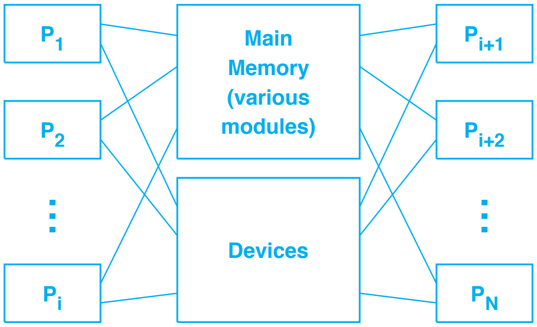


Figure 18.4 The conceptual organization of a symmetric multiprocessor with N identical processors that each have access to memory and I/O devices.

While some researchers explored ways to increase the speed and power of silicon chips, other researchers investigated the symmetric multiprocessor form of MIMD as an alternate way to provide more powerful computers. One of the most well-known projects, which was conducted at Carnegie Mellon University, produced a prototype known as the *Carnegie multiminiprocessor* (*C.mmp*). During the 1980s, vendors first created commercial products, informally called *multiprocessors*, that used the SMP approach. Sequent Corporation (currently owned by IBM) created a symmetric multiprocessor that runs the Unix operating system, and Encore Corporation created a symmetric multiprocessor named *Multimax*.

Asymmetric Multiprocessor (AMP). Although SMPs are popular, other forms of MIMD architectures are possible. The chief alternative to an SMP design is an *Asymmetric Multiprocessor* (*AMP*). An AMP contains a set of *N* programmable processors that can operate at the same time, but does not require all processors to have identical capabilities. For example, an AMP design can choose a processor that is appropriate to a given task (i.e., one processor can be optimized for management of high-speed disk storage devices and another processor can be optimized for graphics display).

In most cases, AMP architectures follow a *master-slave* approach in which one processor (or in some cases a set of processors) controls the overall execution and invokes other processors as needed. The processor that controls execution is known as the *master*, and other processors are known as *slaves*.

In theory, an AMP architecture that has *N* processors can have many distinct types of processors. In practice, however, most AMP designs have between two and four types of processors. Typically, a general-purpose AMP architecture includes at least one processor optimized for overall control (the master), and others optimized for subsidiary functions such as arithmetic computation or I/O.

Math And Graphics Coprocessors. Commercial computer systems have been created that use an asymmetric architecture. One of the most widely known AMP designs became popular in the late 1980s and early 1990s when PC manufacturers began selling *math coprocessors*. The idea of a math coprocessor is straightforward: the coprocessor is a special-purpose chip that the CPU can invoke to perform floating point computation. Because it is optimized for one task, a coprocessor can perform the task faster than the CPU.

CDC Peripheral Processors. Control Data Corporation helped pioneer the idea of using an AMP architecture in mainframes when they created the 6000 series of mainframe computers. The CDC architecture used ten *peripheral processors* to handle I/O. Figure 18.5 illustrates the conceptual organization with peripheral processors between the CPU and I/O devices. Interestingly, CDC's peripheral processors were not limited to I/O — a peripheral processor resembled a minicomputer with a general-purpose instruction set that could be used however a programmer chose. The peripheral processors had access to memory, which meant a peripheral processor could read or store values in any location. Although they were much slower than the CPU, all ten peripheral processors on the CDC could execute simultaneously. Thus, it was possible to optimize program performance by dividing tasks among the peripheral processors as well as the CPU.

Although CDC computers are no longer manufactured, the basic idea of programmable I/O processors continues to be used. Surprisingly, multicore chips have made the general approach feasible again because many cores make it possible to dedicate one or more cores to I/O.

I/O Processors. Most mainframe computers use an AMP architecture to handle I/O at high speed without slowing down the CPU. Each external I/O connection is equipped with a dedicated, programmable processor. Instead of manipulating a bus or handling interrupts, the CPU merely downloads a program into the programmable processor. The processor then handles all the details of I/O. For example, the mainframe computers sold by IBM Corporation use programmable I/O processors called *channels*.

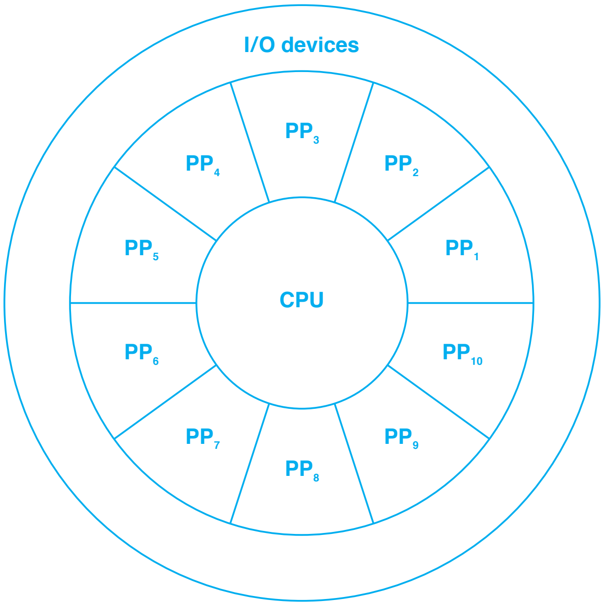


Figure 18.5 Illustration of the asymmetric architecture used in the CDC 6000 mainframe computers.

18.14 Communication, Coordination, And Contention

It may seem obvious that a multiprocessor architecture will always have better performance than a uniprocessor architecture. Consider, for example, a symmetric multiprocessor, M. Intuitively, computer M can outperform a uniprocessor because M can perform *N* times as many operations at any time. Moreover, if a chip vendor finds a way to make a single processor run faster than M, the vendor who sells M merely replaces each of the processors in M with the new chip to have a faster multiprocessor. Indeed, many companies that created multiprocessors made these statements to attract customers.

Unfortunately, our intuition about computer performance can be misleading. Architects have found three main challenges in designing a high-performance parallel architecture:

- Communication
- Coordination
- Contention

Communication. Although it may seem trivial to envision a computer that has dozens of independent processors, the computer must also provide a mechanism that allows the processors to communicate with each other, with memory, and with I/O devices. More important, the communication mechanism must be able to scale to handle a large number of processors. An architect must spend a significant amount of effort to create a parallel computer system that does not have severe communication bottlenecks.

Coordination. In a parallel architecture, processors must work together to perform computation. Therefore, a coordination mechanism is needed that allows processing to be controlled. We said that asymmetric designs usually designate one of the processors to act as a master that controls and coordinates all processing; some symmetric designs also use the master-slave approach. Other architectures use a distributed coordination mechanism in which the processors must be programmed to coordinate among themselves without a master.

Contention. When two or more processors attempt to access a resource at the same time, we say that the processors *contend* for the resource. Resource *contention* creates one of the greatest challenges in designing a parallel architecture because contention increases as the number of processors increases.

To understand why contention is a problem, consider memory. If a set of N processors all have access to a given memory, a mechanism is needed that only permits one processor to access the memory at any time. When multiple processors attempt to use the memory simultaneously, the hardware contention mechanism blocks all except one of them. That is, N − 1 of the processors are idle during the memory access. In the next round, N − 2 processors remain idle. It should be obvious that:

> *In a parallel architecture, contention for shared resources lowers performance dramatically because only one processor can use a given resource at any time; the hardware contention mechanism forces other processors to remain idle while they wait for access.*

18.15 Performance Of Multiprocessors

Multiprocessor architectures have not fulfilled the promise of scalable, high-performance computing. There are several reasons: operating system bottlenecks, contention for memory, and I/O. In a modern computer system, the operating system controls all processing, including allocating tasks to processors and handling I/O. Only one copy of an operating system can run because a device cannot take orders from multiple processors simultaneously. Thus, in a multiprocessor, at most one processor can run operating system software at any time, which means the operating system is a shared resource for which processors must contend. As a consequence, the operating system quickly becomes a bottleneck that processors access serially — if *K* processors need access, *K − 1* of them must wait.

Contention for memory has proven to be an especially difficult problem. First, hardware for a multiported memory is extremely expensive. Second, one of the more important optimizations used in memory systems, caching, causes problems when used with a multiprocessor. If the cache is shared, processors contend for access. If each processor has a private cache, all caches must be coordinated so that any update is propagated to all caches. Unfortunately, such coordination introduces overhead.

Many multiprocessor architectures suffer from another weakness: the architecture only outperforms a uniprocessor when performing intensive computation. Surprisingly, most applications are not limited by the amount of computation they perform. Instead, most applications are *I/O bound*, which means the application spends more time waiting for I/O than performing computation. For example, most of the delay in common applications, such as word spreadsheets, video games, and Web browsing, arises when the application waits for I/O from a file or the network. Therefore, adding additional computational power to the underlying computer does not lower the time required to perform the computation — the extra processors sit idle waiting for I/O.

To assess the performance of an N-processor system, we define the notion of *speedup* to be the ratio of the performance of a single processor to the performance of a multiprocessor. Specifically, we define speedup as:

$$Speedup \ = \ \frac{\tau_1}{\tau_N}$$

where τ_1 denotes the execution time taken on a single processor, and τ_N denotes the execution time taken on a multiprocessor†. In each case, we assume performance is measured using the best algorithm available (i.e., we allow the program to be rewritten to take advantage of parallel hardware).

When multiprocessors are measured performing general-purpose computing tasks, an interesting result emerges. In an ideal situation, we would expect performance to increase linearly as more processors are added to a multiprocessor system. Experience has shown, however, that problems like memory contention, inter-processor communication, and operating system bottlenecks mean that multiprocessors do not achieve linear speedup. Instead, performance often reaches a limit as Figure 18.6 illustrates.

Surprisingly, the performance illustrated in the figure may not be achievable in practice. In some multiprocessor designs, communication overhead and memory contention dominate the running time: as more and more processors are added, the performance starts to decrease. For example, a particular symmetric multiprocessor design exhibited a small speedup with a few processors. However, when sixty-four processors were used, communication overhead made the performance worse than a single processor system. We can summarize:

> *When used for general-purpose computing, a multiprocessor may not perform well. In some cases, added overhead means performance decreases as more processors are added.*

†Because we expect the processing time on a single processor to be greater than the processing time on a multiprocessor, we expect the speedup to be greater than one.

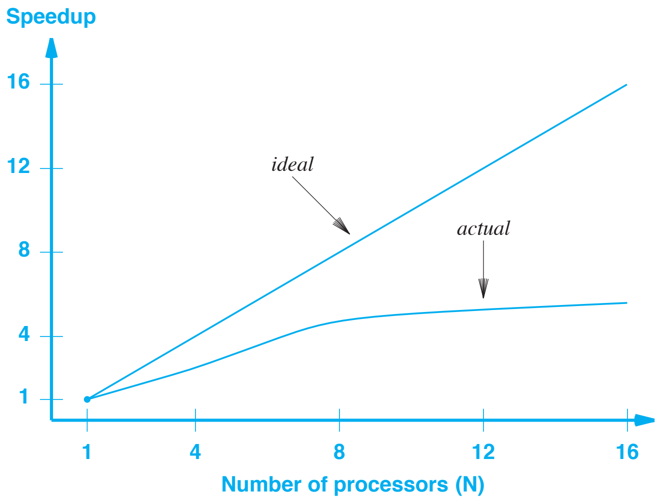


Figure 18.6 Illustration of the ideal and typical performance of a multiprocessor as the number of processors is increased. Values on the y-axis list the relative speedup compared to a single processor.

18.16 Consequences For Programmers

Parallelism usually makes programming more complex. A programmer must be aware of parallel execution, and must prevent one parallel activity from interfering with another. The following sections describe some of the mechanisms and facilities that programmers use.

18.16.1 Locks And Mutual Exclusion

Writing code that uses multiple processors is inherently more complex than writing code for a single processor. To understand the complexity, consider using a shared variable. For example, suppose two processors use a variable *x* to store a count. A programmer writes a statement such as:

```
x = x + 1;
```

A compiler translates the statement into a sequence of machine instructions, such as the sequence in Figure 18.7.

```
load    x, R5    # Load variable x into R5
incr    R5       # Increment the value in R5
store   R5, x    # Store R5 back into x
```

Figure 18.7 An example sequence of machine instructions used to increment a variable in memory. In most architectures, increment entails a *load* and a *store* operation.

Unfortunately, if two processors attempt to increment *x* at nearly the same time, the value of *x* might be incremented once instead of twice. The error arises because each of the two processors operates independently and competes for access to memory. Thus, the operations might be performed in the order given in Figure 18.8.

- Processor 1 loads x into its register 5
- Processor 1 increments its register 5
- Processor 2 loads x into its register 5
- Processor 1 stores its register 5 into x
- Processor 2 increments its register 5
- Processor 2 stores its register 5 into x

Figure 18.8 A sequence of steps that can occur when two independent processors or cores access variable *x* in shared memory.

To prevent problems like the one illustrated in Figure 18.8, multiprocessor hardware provides *hardware locks*. A programmer must associate a lock with each shared item, and use the lock to ensure that no other processors can change the item while an update is in progress. For example, if lock 17 is associated with variable *x*, a programmer must obtain lock 17 before updating *x*. The idea is called *mutual exclusion*, and we say that a processor must gain *exclusive use* of a variable before updating the value. Figure 18.9 illustrates the sequence of instructions.

```
lock     17       # wait for lock 17
load     x, R5    # Load variable x into R5
incr     R5       # Increment the value in R5
store    R5, x    # Store R5 back into x
release 17        # release lock 17
```

Figure 18.9 Illustration of the instructions used to guarantee exclusive access to a variable. A separate lock is assigned to each shared item.

The underlying hardware guarantees that only one processor will be granted a lock at any time. Thus, if two or more processors both attempt to obtain a given lock at the same time, one obtains access (i.e., continues to execute) and the other is blocked. In fact, an arbitrary number of processors can be blocked while one processor holds the lock. Once the processor that holds the lock releases it, the hardware selects a blocked processor, grants the processor the lock, and allows the processor to proceed. Thus, the hardware ensures that at most one processor can hold a given lock at any time.

Locking adds a nontrivial amount of complexity to programs for several reasons. First, because locking is unusual, a programmer not accustomed to programming multiprocessors can easily forget to lock a shared variable, and because unprotected access may not always result in an error, the problem can be difficult to detect. Second, locking can severely reduce performance — if K processors attempt to access a shared variable at the same time, the hardware will keep $K-1$ of them idle while they wait for access. Third, because separate instructions are used to obtain and release a lock, locking adds overhead. Thus, a programmer must decide whether to obtain a lock for each individual operation or whether to obtain a lock, hold the lock while performing a series of operations on the variable, and then release the lock.

18.16.2 Programming Explicit And Implicit Parallel Computers

The most important aspect of parallelism for a programmer concerns whether software or hardware is responsible for managing parallelism: a system that uses implicit parallelism is significantly easier to program than a system that uses explicit parallelism. For example, consider a processor designed to handle packets arriving from a computer network. In an implicit design, a programmer writes code to handle a single packet, and the hardware automatically applies the same program to N packets in parallel. In an explicit design, the programmer must plan to read N packets, send each to a different core, wait for the cores to complete processing, and extract the resulting packets. In many cases, the code required to control parallel cores and determine when they each finish is more complex than the code to perform the desired computation. More important, code to control parallel hardware units must allow hardware to operate in arbitrary order. For example, because the time required to process a packet depends on the packet's contents, a controller must be ready for the hardware units to complete processing in arbitrary order. The point is:

> *From a programmer's point of view, a system that uses explicit parallelism is significantly more complex to program than a system that uses implicit parallelism.*

18.16.3 Programming Symmetric And Asymmetric Multiprocessors

One of the most important advantages of symmetry arises from the positive consequences it has for programmers: a symmetric multiprocessor can be substantially easier to program than an asymmetric multiprocessor. First, if all processors are identical, a programmer only needs one compiler and one language. Second, symmetry means a programmer does not need to consider which tasks are best suited for which type of processor. Third, because identical processors usually operate at the same speed, a programmer does not need to worry about the time required to perform a task on a given processor. Fourth, because all processors use the same encoding for instructions and data, a binary program or a data value can be moved from one processor to another.

Of course, any form of multiprocessor introduces a complication: in addition to everything else, a programmer must consider how coding decisions will influence performance. For example, consider a computation that processes packets arriving over a network. A conventional program keeps a global counter in memory, and updates the counter when a packet arrives. On a shared memory architecture, however, updating a value in memory is more expensive because a processor must obtain a lock before updating a shared value in memory. Thus, a programmer needs to consider the effect of minor details, such as updating a shared counter in memory.

18.17 Redundant Parallel Architectures

Our discussion has focused on the use of parallel hardware to improve performance or increase functionality. However, it is also possible to use parallel hardware to improve reliability and prevent failure. That is, multiple copies of hardware can be used to verify each computation.

The term *redundant hardware* usually refers to multiple copies of a hardware unit that operate in parallel to perform an operation. The basic difference between redundant hardware and the parallel architectures described above arises from the data items being used: a parallel architecture arranges for each copy of the hardware to operate on a separate data item; a redundant architecture arranges for all copies to perform exactly the same operation.

The point of using redundant hardware is verification that a computation is correct. What happens when redundant copies of the hardware disagree? The answer depends on the details and purpose of the underlying system. One possibility uses votes: K copies of a hardware unit each perform the computation and produce a value. A special hardware unit then compares the output, and selects the value that appears most often. Another possibility uses redundant hardware to detect hardware failures: if two copies of the hardware disagree, the system displays an error message, and then halts until the defective unit can be repaired or replaced.

18.18 Distributed And Cluster Computers

The parallel architectures discussed in this chapter are called *tightly coupled* because the parallel hardware units are located inside the same computer system. The alternative, which is known as a *loosely coupled* architecture uses multiple computer systems that are interconnected by a communication mechanism that spans longer distances. For example, we use the term *distributed architecture* to refer to a set of computers that are connected by a computer network or an internet. In a distributed architecture, each computer operates independently, but the computers can communicate by sending messages across a network.

A special form of distributed computing system is known as a *network cluster* or a *cluster computer*. In essence, a cluster computer consists of a set of independent computers, such as commodity PCs, connected by a high-speed computer network. Scientists use cluster computers to run computations on extremely large sets of data, Internet search companies use clusters to respond to users' search terms, and cloud providers use the cluster approach to build cloud data centers. The general idea is that for a cluster of *N* computers, computation can be divided many ways. The computers in a cluster are flexible — they can be dedicated to solving a single problem or separate problems. Computers in the cluster run independently. If they are working on a single problem, the results may be collected to produce the final output.

A special case of cluster computing is used to construct a high-capacity Web site that handles many small requests. Each computer in the cluster runs a copy of the same Web server. A special-purpose system known as a *Web load balancer* disperses incoming requests among computers in the cluster. Each time a request arrives, the load balancer chooses the least-loaded computer in the cluster and forwards the request. Thus, a Web site with *N* computers in a cluster can respond to approximately *N* times as many requests per second as a single computer.

Another form of loosely coupled distributed computing is known as *grid computing*. Grid computing uses the global Internet as a communication mechanism among a large set of computers. The computers (typically personal computers owned by individuals) agree to provide spare CPU cycles for the grid. Each computer runs software that repeatedly accepts a request, performs the requested computation, and returns the result. To use the grid, a problem must be divided into many small pieces. Each piece of the problem is sent to a computer, and all computers can execute simultaneously.

18.19 A Modern Supercomputer

Informally, the term *supercomputer* is used to denote an advanced computing system that has significantly more processing power than mainframe computers. Because they are often used for scientific calculations, supercomputers are typically assessed by the number of floating point operations per second the computer can perform.

Parallelism has always played an important role in supercomputers. Early supercomputers had 16 or 64 processors. A modern supercomputer consists of a cluster of many PCs that are interconnected by a high-speed Local Area Network. Furthermore, the processor in each PC has multiple cores. Modern supercomputers carry parallelism to a surprising extreme. For example, the *Tianhe-2* supercomputer in China consists of a cluster of 16,000 Intel nodes. Each node has its own memory and a set of processors, each of which has multiple cores. The resulting system has a total of 3,120,000 cores. The computational power of a computer with over 3 million cores is difficult to imagine.

18.20 Summary

Parallelism is a fundamental optimization technique used to increase hardware performance. Most components of a computer system contain parallel hardware; an architecture is only classified as parallel if the architecture includes parallel processors. Explicit parallelism gives a programmer control over the use of parallel facilities; implicit parallelism handles parallelism automatically.

A uniprocessor computer is classified as a Single Instruction Single Data (SISD) architecture because a single instruction operates on a single data item at any given time. A Single Instruction Multiple Data (SIMD) architecture allows an instruction to operate on an array of values. Typical SIMD machines include vector processors and graphics processors. A Multiple Instructions Multiple Data (MIMD) architecture employs multiple, independent processors that operate simultaneously and can each execute a separate program. Typical MIMD machines include symmetric and asymmetric multiprocessors. Alternatives to SIMD and MIMD architectures include redundant, distributed, cluster, and grid architectures.

In theory, a general-purpose multiprocessor with *N* processors should perform *N* times faster than a single processor. In practice, however, memory contention, communication overhead, and coordination mean that the performance of a multiprocessor does not increase linearly as the number of processors increases. In the extreme case, overhead means that performance can decrease as additional processors are added.

Programming a computer with multiple processors can be a challenge. In addition to other considerations, a programmer must use locks to guarantee exclusive access to shared items.

A modern supercomputer consists of a large cluster of processors. If a problem can be partitioned into subparts, the processors in a supercomputer cluster can work on subparts in parallel.

EXERCISES

18.1 Define macroscopic parallelism and give an example.

18.2 If a computer has four cores plus two GPU cores, does the system have symmetric parallelism, asymmetric parallelism, or some of both? Explain?

18.3 Use the Flynn classification scheme to classify a dual-core smart phone.

18.4 What is contention, and how does it affect performance?

18.5 A C programmer is writing code that will run on multiple cores, and must increment a shared variable *x*. Instead of writing:

```
x = x + 1;
```

the C programmer writes:

```
x++;
```

Does the second form guarantee that two cores can execute the increment without interfering with one another? Explain.

18.6 You receive two job offers for the same salary, one writing code for a system that uses explicit parallelism and another writing code for a system that uses implicit parallelism. Which do you choose, and why?

18.7 Consider multiplying two 10 x 20 matrices on a computer that has vector capability but limits each vector to sixteen items. How is matrix multiplication handled on such a computer, and how many vector multiplications are required?

18.8 In the previous exercise, how many scalar multiplications are needed on a uniprocessor (i.e., an SISD architecture)? If we ignore addition and only measure multiplication, what is the speedup? Does the speedup change when multiplying 100 x 100 matrices?

18.9 If you have access to single-processor and dual-processor computers that use the same clock rate, write a program that consumes large amounts of CPU time, run multiple copies on both computers, and record the running times. What is the effective speedup?

18.10 In the previous question, change the program to reference large amounts of memory (e.g., repeatedly set a large array to a value *x*, then set the array to value *y*, and so on). How do memory references affect the speedup?

18.11 Can a multiprocessor ever achieve speedup that is *better* than linear? To find out, consider an encryption breaking algorithm that must try twenty-four (four factorial) possible encryption keys and must perform up to 1024 operations to test each key (stopping early only if an answer is found). If we assume a multiprocessor requires K milliseconds to perform 1024 operations, on average how much time will the processor spend solving the entire problem? How much time will a 32-processor MIMD machine spend solving the problem? What is the resulting speedup?

18.12 Search the Web to find a list of the top 10 supercomputers. How many cores does each have?

Chapter Contents

19

Data Pipelining

19.1 Introduction

Earlier chapters present processors, memory systems, and I/O as the fundamental aspects of computer architecture. The previous chapter shows how parallelism can be used to increase performance, and explains a variety of parallel architectures.

This chapter focuses on the second major technique used to increase performance: data pipelining. The chapter discusses the motivation for pipelining, explains the variety of ways pipelining is used, and shows why pipelining can increase hardware performance.

19.2 The Concept Of Pipelining

The term *pipelining* refers broadly to any architecture in which digital information flows through a series of stations (e.g., processing components) that each inspect, interpret, or modify the information as Figure 19.1 illustrates.

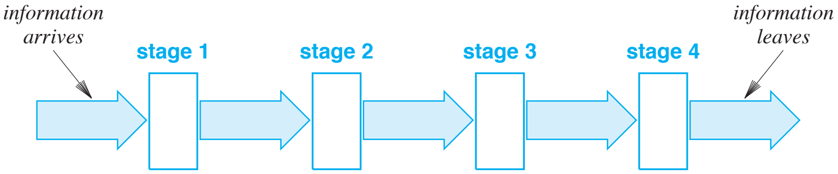


Figure 19.1 Illustration of the pipeline concept. The example pipeline has four stages, and information flows through each stage.

Although we are primarily interested in hardware architectures and the use of pipelining within a single computer system, the concept itself is not limited to hardware. Pipelining is not restricted to a single computer, a particular type or size of digital information, or a specific length of pipeline (i.e., a particular number of stages). Instead, pipelining is a fundamental concept in computing that is used in a variety of situations.

To help us understand the concept, we will consider a set of characteristics. Figure 19.2 lists ways to characterize pipelines, and succeeding paragraphs explain each of the characteristics.

- Hardware or software implementation
- Large or small scale
- Synchronous or asynchronous flow
- Buffered or unbuffered flow
- Finite chunks or continuous bit streams
- Automatic data feed or manual data feed
- Serial or parallel path
- Homogeneous or heterogeneous stages

Figure 19.2 The variety of ways a pipeline can be used in digital systems.

Hardware Or Software Implementation. Pipelining can be implemented in either software or hardware. For example, the Unix operating system provides a *pipe* mechanism that can be used to form a software pipeline — a set of processes creates pipes that connect the output of one process to the input of the next process. We will examine hardware pipelines in later sections. However, it should be noted that software and hardware pipelines are independent: a software pipeline can be created on a computer that does not use a pipeline hardware architecture, and pipeline hardware is not necessarily visible to programmers.

Large Or Small Scale. Stages in a pipeline can range from simplistic to powerful, and a pipeline can range in length from short to long. At one extreme, a hardware pipeline can be contained entirely within a small functional unit on a chip. At the other extreme, a software pipeline can be created by passing data through a series of programs that each run on a separate computer and use the Internet to communicate. Similarly, a short pipeline can be formed of two stages, one that generates information and one that absorbs it, and a long pipeline can contain hundreds of stages.

Synchronous Or Asynchronous Flow. A *synchronous pipeline* operates like an assembly line: at a given time, each stage is processing some amount of information (e.g., a byte). A global clock controls movement, which means that all stages simultaneously forward their data (i.e., the results of processing) to the next stage. The alternative, an *asynchronous pipeline*, allows a station to forward information at any time. Asynchronous communication is especially attractive for situations where the amount of time a given stage spends on processing depends on the data the stage receives. However, asynchronous communication can mean that if one stage delays for a long time, later stages must wait.

Buffered Or Unbuffered Flow. Our conceptual diagram in Figure 19.1 implies that one stage of a pipeline sends data directly to another stage. It is also possible to construct a pipeline in which a *buffer* is placed between each pair of stages. Buffering is useful with asynchronous pipelines in which information is processed in *bursts* (i.e., a pipeline in which a stage repeatedly emits steady output, then ceases emitting output, and then begins emitting steady output again).

Finite Chunks Or Continuous Bit Streams. The digital information that passes through a pipeline can consist of a sequence of small data items (e.g., packets from a computer network) or an arbitrarily long bit stream (e.g., a continuous video feed). Furthermore, a pipeline that operates on individual data items can be designed such that all data items are the same size (e.g., disk blocks that are each four Kbytes) or the size of data items is not fixed (e.g., a series of Ethernet packets that vary in length).

Automatic Data Feed Or Manual Data Feed. Some implementations of pipelines use a separate mechanism to move information, and other implementations require each stage to participate in moving information. For example, a synchronous hardware pipeline typically relies on an auxiliary mechanism to move information from one stage to another. However, a software pipeline usually requires each stage to *write* outgoing data and *read* incoming data explicitly.

Serial Or Parallel Path. The large arrows in Figure 19.1 imply that a parallel path is used to move information from one stage to another. Although some hardware pipelines do use a parallel path, many use serial communication. Furthermore, communication between stages need not consist of conventional communication (e.g., stages can use a computer network or shared memory to communicate).

Homogeneous Or Heterogeneous Stages. Although Figure 19.1 uses the same size and shape for each stage of a pipeline, homogeneity is not required. Some implementations of pipelines choose a type of hardware that is appropriate for each stage.

19.3 Software Pipelining

From a programmer's point of view, a software pipeline is attractive for two reasons. First, a software pipeline provides a way to handle complexity. Second, a software pipeline allows programs to be re-used. In essence, both goals are achieved because a software pipeline allows a programmer to divide a large, complex task into smaller, more generic pieces.

As an example of software pipelining, consider the pipeline facilities provided by the Unix shell (i.e., the command interpreter). To create a software pipeline, a user enters a list of command names separated by the vertical bar character to specify that the programs should be run as a pipeline. The shell arranges the programs so the output from one program becomes the input of the next. Each program can have zero or more arguments that control processing. For example, the following input to the shell specifies that three programs, *cat*, *sed*, and *more* are to be connected in a pipeline:

```
cat x | sed ’s/friend/partner/g’ | more
```

In the example, the *cat* program writes a copy of file *x* (presumably a text file) to its output, which becomes the input of the *sed* program. The *sed* program, in the middle of the pipeline, receives input from *cat* and sends output to *more*. *Sed* has an argument that specifies translating every occurrence of the word *friend* to *partner*. The final program in the pipeline, *more*, receives input from *sed* and displays the input on the user’s screen.

Although the example above is trivial, it illustrates how a software pipeline helps programmers. Decomposing a program into a series of smaller, less complex programs makes it easier to create and debug software. Furthermore, if the division is chosen carefully, some pieces can be re-used among programs. In particular, programmers often find that using a pipeline to separate input and output processing from computation allows the code that performs computation to be re-used with various forms of input and output.

19.4 Software Pipeline Performance And Overhead

It may seem that software pipelining results in lower performance than a single program. The operating system must run multiple application programs concurrently, and must pass data between pairs of programs. Inefficiency can be especially high if early stages of a pipeline pass large volumes of data that are later discarded. For example, consider the following software pipeline that contains one more stage than the example above: an additional invocation of *sed* that deletes any line containing the character *W*.

```
cat x | sed ’s/friend/partner/g’ | sed ’/W/d’ | more
```

If we expect ninety-nine percent of all lines to contain the character *W*, the first two stages of the pipeline will perform unnecessary work (i.e., processing lines of text that will be discarded in a later stage of the pipeline). In the example, the pipeline can be optimized by moving the deletion to an earlier stage. However, the overhead of using a software pipeline appears to remain: copying data from one program to another is less efficient than performing all computation in a single program.

Surprisingly, a software pipeline can perform better than a large, monolithic program, even if the underlying hardware does not use multiple cores. To understand why, consider the underlying architecture: processing, memory, and I/O are constructed from independent hardware. An operating system takes advantage of the independence by automatically switching the processor among application programs (i.e., processes): when one application is waiting for I/O, another application runs. Thus, if a pipeline is composed of many small applications, the operating system may be able to improve overall performance by running one of the applications in a pipeline, while another application waits for I/O.

19.5 Hardware Pipelining

Like software pipelining, hardware pipelining can help a designer manage complexity — a complex task can be divided into smaller, more manageable pieces. However, the most important reason architects choose a hardware pipeline is increased performance. There are two distinct uses of hardware pipelines that each provide high performance:

- Instruction pipeline
- Data pipeline

Instruction Pipeline. Chapter 5 explains how the fetch-execute cycle in a processor can use a pipeline to decode and execute instructions. To be precise, we use the term *instruction pipeline* to describe a pipeline in which the information consists of machine instructions and the stages of the pipeline decode and execute the instructions. Because the instruction set and operand types vary among processors, there is no overall agreement on the number of stages in an instruction pipeline or the exact operations performed at a given stage†.

Data Pipeline. The alternative to an instruction pipeline is known as a *data pipeline*. That is, instead of passing instructions, a data pipeline is designed to pass data from stage to stage. For example, if a data pipeline is used to handle packets that arrive from a computer network, each packet passes sequentially through the stages of the pipeline. Data pipelining provides some of the most unusual and most interesting uses of pipelining. As we will see, data pipelining also has the potential for the greatest overall improvement in performance.

19.6 How Hardware Pipelining Increases Performance

To understand why pipelining is fundamental in hardware design, we need to examine a key point: pipelining can dramatically increase performance. To see how, compare a data pipeline to a monolithic design. For example, consider the design of an Internet router that is used by an Internet Service Provider (ISP) to forward packets between customers and Web sites. A router connects to multiple networks, some of which lead to customers and at least one leads to the Internet. Network packets can arrive over any network, and the router's job is to send each packet on toward its destination. For purposes of this example, we will assume the router performs six basic operations on each packet as listed in Figure 19.3. It is not important to understand each of the operations, only to appreciate that the example is realistic.

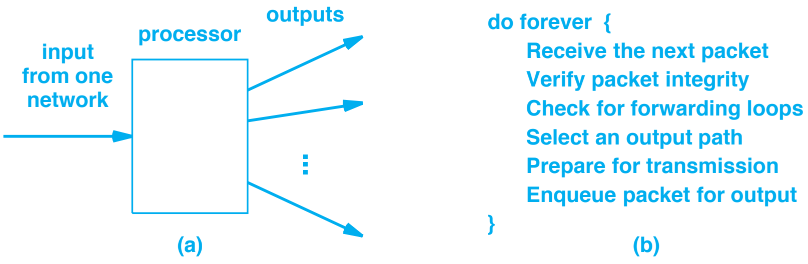
†The definition of *superpipeline*, given later in this chapter, also relates to an instruction pipeline.

1. Receive a packet (i.e., read the packet from the network device and transfer the bytes into a buffer in memory)
2. Verify packet integrity (e.g., use a checksum to verify that no changes occurred between transmission and reception)
3. Check for forwarding loops (i.e., decrement a value in the header, and reform the header with the new value)
4. Select a path (i.e., use the destination address field in the packet to select one of the possible output networks and a destination on that network)
5. Prepare for transmission (i.e., compute information that will be sent with the packet and used by the receiver to verify integrity)
6. Transmit the packet (i.e., transfer the packet to the output device)

Figure 19.3 An example series of steps that hardware in an Internet router performs to forward a packet.

Consider the design of hardware that implements the steps in the figure. Because the steps involve complex computation, it may seem that a processor should be used to perform packet forwarding. However, a single processor is not fast enough for high-speed networks. Thus, most designs employ two optimizations described in earlier chapters: smart I/O devices and parallelism. A smart I/O device can transfer a packet to or from memory without using a processor, and a parallel design uses a separate processor to handle each input.

A parallel router design with a smart I/O interface means that each processor implements a loop that repeatedly executes the six basic steps. Figure 19.4 illustrates how a processor connects to an input, and shows the algorithm the processor runs.



```
do forever {
    Receive the next packet
    Verify packet integrity
    Check for forwarding loops
    Select an output path
    Prepare for transmission
    Enqueue packet for output
}
```

Figure 19.4 (a) Illustration of the connections on a processor used in a parallel implementation of an Internet router, and (b) the algorithm the processor executes. Each processor handles input from one network.

Suppose a parallel architecture, like the one in the figure, is still too slow. That is, suppose the processor cannot execute all the steps of the algorithm before the next packet arrives over the interface and no faster processor is available. How can we achieve higher performance? One possibility for higher speed lies in a data pipeline: use a pipeline of several processors in place of a single processor as Figure 19.5 illustrates†.

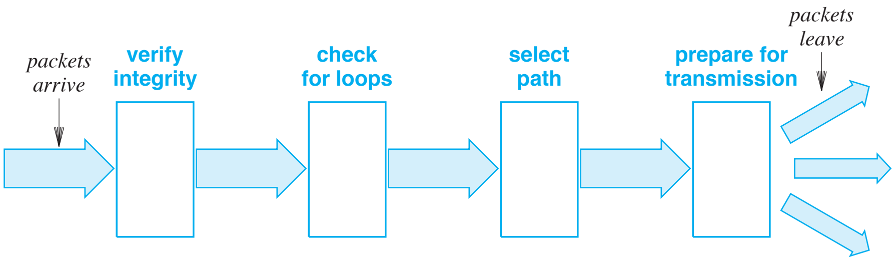


Figure 19.5 Illustration of a pipeline used in place of a single processor in an Internet router.

It may seem that the pipeline in the figure is no faster than the single processor in Figure 19.4. After all, the pipeline architecture performs exactly the same operations on each packet as the single processor. Furthermore, if the processors in Figure 19.5 are each the same speed as the processor in Figure 19.4, the time to perform a given operation will be the same. For example, the step labeled *verify integrity* will take the same amount of time on both architectures, the step labeled *check for loops* will take the same amount of time on both architectures, and so on. Thus, if we ignore the delay introduced by passing packets among stages of the pipeline, the total time taken to process a packet is exactly the same as in a single processor architecture. That is:

> *A data pipeline passes data through a series of stages that each examine or modify the data. If it uses the same speed processors as a non-pipeline architecture, a data pipeline will not improve the overall time needed to process a given data item.*

If the total processing time required for an item is the same in the pipelined and non-pipelined architectures, what is the advantage of a data pipeline? Surprisingly, even if the individual processors in Figure 19.5 are each exactly the same speed as the processor in Figure 19.4, the pipeline architecture can process more packets per second. To see why, observe that an individual processor executes fewer instructions per packet. Furthermore, after operating on one data item, a processor moves on to the next data item. Thus, a data pipeline architecture allows a given processor to move on to the next data item more quickly than a nonpipeline architecture. As a result, data can enter (and leave) a pipeline at a higher rate.

†A pipeline provides an example of the Flynn MISD type of parallel architecture mentioned in the previous chapter.

We can summarize:

> *Even if a data pipeline uses the same speed processors as a nonpipeline architecture, a data pipeline has higher overall throughput (i.e., number of data items processed per second).*

19.7 When Pipelining Can Be Used

A pipeline will not yield higher performance in all cases. Figure 19.6 lists conditions that must be met for a pipeline to perform faster than a single processor.

- Partionable problem
- Equivalent processor speed
- Low overhead data movement

Figure 19.6 The three key conditions that must be met for a data pipeline to perform better than the same computation on a single processor.

Partionable Problem. It must be possible to partition processing into stages that can be computed independent of one another. Computations that employ a sequence of steps work well in a pipeline, but computations that involve iteration often do not.

Equivalent processor speed. It should be obvious that if the processors used in a data pipeline are slow enough, the overall time required to perform a computation will be much higher than on a single processor. Processors in the pipeline do not need to be faster than the single processor. We merely require that each processor in the pipeline is approximately as fast as the single processor. That is, the time required to perform a given computation on a pipeline processor must not exceed the time required to perform the same computation on the single processor.

Low overhead data movement. In addition to the time required to perform computation, a data pipeline has an additional overhead: the time required to move a data item from one stage of the pipeline to the next. If moving the data incurs extremely high latency, pipelining will not increase performance.

The requirements arise because of an important principle:

> *The throughput of a pipeline is limited by the stage that takes the most time.*

As an example, consider the data pipeline in Figure 19.5. Suppose that all processors in the pipeline are identical, and assume that a pipeline processor takes exactly the same time to execute an instruction as the single processor. To make the example concrete, assume that a processor can execute ten instructions each microsecond. Further suppose the four stages in the figure take fifty, one hundred, two hundred, and one hundred fifty instructions, respectively, to process a packet. The slowest stage requires two hundred instructions, which means the total time the slowest stage takes to process a packet is:

$$\textit{total time} \;=\; \frac{200\ \textit{inst}}{10\ \textit{inst}\ /\ \mu \textit{sec}} \;=\; 20\ \mu\textit{sec} \tag{19.1}$$

Looking at this another way, we can see that the maximum number of packets that can be processed per second is the inverse of the time per packet of the slowest stage. Thus, the overall throughput of the example pipeline, T_p, is given by:

$$T_p \;=\; \frac{1\ \textit{packet}}{20\ \mu\textit{sec}} \;=\; \frac{1\ \textit{packet} \times 10^6}{20\ \textit{sec}} \;=\; 50{,}000\ \textit{packets per second} \tag{19.2}$$

In contrast, a non-pipelined architecture must execute all 500 instructions for each packet, which means that the total time required for a packet is 50 μ*sec*. Thus, the throughput of the non-pipelined architecture is limited to:

$$T_{np} \;=\; \frac{1\ \textit{packet}}{50\ \mu\textit{sec}} \;=\; \frac{1\ \textit{packet} \times 10^6}{50\ \textit{sec}} \;=\; 20{,}000\ \textit{packets per second} \tag{19.3}$$

19.8 The Conceptual Division Of Processing

The reason data pipelining improves performance arises because pipelining provides a special form of parallelism. By dividing a series of sequential operations into groups that are each handled by a separate stage of the pipeline, pipelining allows stages to operate in parallel. Of course, a pipeline architecture differs from a conventional parallel architecture in a significant way: although the stages operate in parallel, a given data item must pass through all stages. Figure 19.7 illustrates the concept.

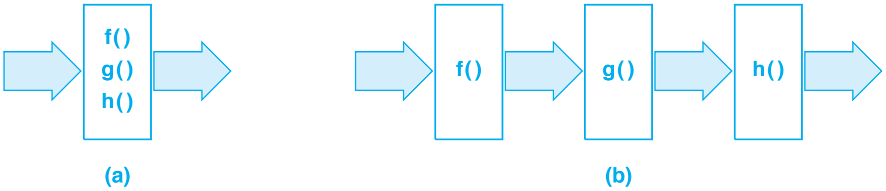


Figure 19.7 (a) Processing on a conventional processor, and (b) equivalent processing in a data pipeline. The functions performed in sequence are divided among stages of the pipeline.

The point of the figure is that the three stages operate in parallel. Stage three performs function *h* on one data item at the same time stage two performs function *g* on a second data item and stage one performs function *f* on a third data item. As long as the pipeline is full (i.e., there are no delays between items), the overall system benefits from N stages all running in parallel.

19.9 Pipeline Architectures

Recall from the previous chapter that we distinguish between hardware architectures that merely use parallelism and architectures in which parallelism forms the central paradigm around which the entire architecture is designed. We make an analogous distinction between hardware architectures that use pipelining and architectures in which pipelining forms the central paradigm around which the entire system is designed. We reserve the name *pipeline architectures* for the latter. Thus, one might hear an architect say that the processor in a given system uses instruction pipelining, but the architect will not characterize the system as a pipeline architecture unless the overall design centers around a pipeline.

Most hardware systems that follow a pipeline architecture are dedicated to special-purpose functions. For instance, the example above describes how pipelining can be used to improve performance of a packet processing system. Pipelining is especially important in network systems because the high data rates used when sending data over optical fibers exceeds the capacity of conventional processors.

Pipeline architectures are less relevant to general-purpose computers for two reasons. First, few applications can be decomposed into a set of independent operations that can be applied sequentially. Instead, a typical application accesses items randomly and keeps large volumes of additional state information. Second, even in situations where the functions to be performed on data can be decomposed into a pipeline, the number of stages in the pipeline and the hardware needed to implement each stage is not usually known in advance. As a result, general-purpose computers usually restrict pipeline hardware to an instruction pipeline in the processor or a special-purpose pipeline in an I/O device.

19.10 Pipeline Setup, Stall, And Flush Times

Our description of pipelines overlooks many of the practical details. For example, many pipeline implementations have overhead associated with starting and stopping the pipeline. We use the term *setup time* to describe the amount of time required to start a pipeline after an idle period. Setup may involve synchronizing processing among stages or passing a special control token through the pipeline to restart each stage. For a software pipeline, setup can be especially expensive because connections among various stages are created dynamically.

Unlike other architectures, a pipeline can require significant time to terminate processing. We use the term *flush time* to refer to the amount of time that elapses between the input being unavailable and the pipeline finishing its current processing. We say that items currently in the pipeline must be *flushed* before the pipeline can be shut down.

The need to flush items through a pipeline can arise for two reasons. First, a pipeline becomes idle when no input is available for the first stage. Second, as we have seen, later stages of a pipeline become idle when one stage *stalls* (i.e., the stage delays because it cannot complete processing). In a high-speed hardware pipeline, mundane operations such as a memory reference or an I/O operation can cause a stage to stall. Thus, high flush (or setup) times can reduce pipeline performance significantly.

19.11 Definition Of Superpipeline Architecture

A final concept completes our description of pipelines. Architects use the term *superpipeline* to describe an extension of the pipeline approach in which a given stage of the pipeline is subdivided into a set of partial stages. Superpipelining is most often used with an instruction pipeline, but the concept applies to data pipelines as well. The general idea is: if dividing processing into N stages can increase overall throughput, adding more stages can increase throughput further.

A traditional instruction pipeline might have five stages that correspond to: instruction fetch, instruction decode, operand fetch, ALU operation, and memory write. A superpipeline architecture subdivides one or more stages into multiple pieces. For example, a superpipeline might subdivide the operand fetch stage into four steps: decode an operand, fetch immediate values or values from registers, fetch values from memory, and fetch indirect operand values. As with standard pipelining, the point of the subdivision is higher throughput — because each substage takes less time, throughput of a superpipeline is higher than the throughput of a standard pipeline.

19.12 Summary

Pipelining is a broad, fundamental concept that is used with both hardware and software. A software pipeline, which arranges a set of programs in a series with data passing through them, can be used on hardware that does not provide pipelining.

A hardware pipeline is either classified as an instruction pipeline, which is used inside a processor to handle machine instructions, or a data pipeline, in which arbitrary data is transferred through the pipeline. The superpipeline technique, in which a stage of a pipeline is further subdivided into partial stages, is often used with an instruction pipeline.

A data pipeline does not decrease the overall time required to process a single data item. However, using a pipeline does increase the overall throughput (items processed per second). The stage of a pipeline that requires the most time to process an item limits the throughput of the pipeline.

EXERCISES

19.1 A scientist uses a cluster of PCs and arranges to have software on each processor perform one step of a computation. The processor reads up to 1 MB of data (whatever is available), processes the data, and then passes its output to the next processor over a 32-bit bus. What characteristics from Figure 19.2 does the arrangement have?

19.2 Your team has been given the task of moving a video processing program from an old single-core processor to a new quad-core processor that has a high-speed interconnect among cores. Conventional parallel approaches will not work because the frames of video must be processed in order. What technique can you suggest that offers a possible way to use the new hardware to improve performance?

19.3 An engineer builds a data pipeline with eight processors. To measure performance, the engineer runs the software on one processor and measures the time taken to process a single data item. The engineer then divides the software into eight stages, and measures the time taken to process a single data item. What do the measurements show?

19.4 Most data pipeline hardware is devoted to specialized tasks (e.g., graphics processing). Would installing a data pipeline in all computers increase the performance of all programs? Why or why not?

19.5 A manager notices that the company has a few idle computers in each of ten data centers. The data centers are spread across the country, with low-speed Internet connections used to communicate among the data centers. The manager proposes that rather than using a computer in the local data center, a "giant data pipeline" be set up across all ten data centers to increase performance. What do you tell the manager about the idea?

19.6 You are given a program that runs on one core, and are asked to divide the program into pieces that will use up to eight cores in a data pipeline. You can divide the program two ways. In one, the cores each perform 680, 2000, 1300, 1400, 800, 1900, 1200, and 200 instructions. In the other, the cores perform 680, 1400, 1300, 1400, 1400, 1000, 1200, and 1100 instructions. Which division do you choose, and why?

19.7 What is the maximum throughput of a homogeneous pipeline in which four processors each handle one million instructions per second and processing a data item requires 50, 60, 40, and 30 instructions, respectively? Assume a constant execution time for all types of instructions.

19.8 In the previous exercise, what is the relative gain in throughput compared to an architecture without pipelining? What is the maximum speedup?

19.9 Extend the previous exercise by considering heterogeneous processors that have speeds of 1.0, 1.2, 0.9, and 1.0 million instructions per second, respectively.

19.10 If you are asked to apply superpipelining to subdivide one of the stages of an existing pipeline, which stage should you choose? Why?

Chapter Contents

20

Power And Energy

20.1 Introduction

The related topics of power consumption and total energy consumption have become increasingly important in the design of computing systems. For portable devices, designs strive for a balance between maximizing battery life and maximizing features that users desire. For large data centers, the power consumed and the consequent cooling required are now critical factors in the design and scale.

This brief chapter introduces the topic without going into much detail. It defines terminology, explains the types of power that digital circuits consume, and describes the relationship between power and energy. Most important, the chapter describes how software systems can be used to shut down parts of a system to reduce power consumption.

20.2 Definition Of Power

We define *power* to be the rate at which energy is consumed (e.g., transferred or transformed). For an electronic circuit, power is the product of voltage and current. Taking the definitions from physics, power is measured in units of *watts*, where a watt is defined as one joule per second (J/s). The higher the wattage of an electronic device, the more power it consumes; some devices use *kilowatts* (10^3 watts) of power. For a large data center cluster, the aggregate power consumed by all the computers in the cluster is so large that it is measured in megawatts (10^6 watts). For small hand-held devices, such as cell phones, the power requirements are so minimal that they are measured in *milliwatts* (10^{-3} watts).

It is important to note that the amount of power a system uses can vary over time. For example a smart phone uses less power when the display is turned off than when the screen is on. Therefore, to be precise, we define the *instantaneous power* at time t, *P(t)*, to be the product of the voltage at time t, *V(t)*, and the current at time t, *I(t)*:

$$P(t) \quad = \quad V(t) \times I(t) \tag{20.1}$$

We will see that the ability of a system to vary its power usage over time can be important for both extremely large and extremely small computing systems (e.g., powerful computers in a data center and small battery powered devices).

The maximum power that a system uses is especially important for large systems, such as a cluster of computers in a data center. We use the term *peak instantaneous power* to specify the maximum power a system will need. Peak power is especially important when constructing a large computing system because the designer must arrange to meet the peak power requirements. For example, when planning a data center, a designer must guarantee that an electric utility can supply sufficient power to meet the peak instantaneous power demand.

20.3 Definition Of Energy

From the above, the total *energy* that a system uses is computed as the power consumed over a given time, measured in joules. Electrical energy is usually reported in multiples of watts multiplied by a unit of time. Typically, the time unit is an hour, and the multiples of watts are kilowatts, megawatts, or milliwatts. Thus, the energy consumed by a data center during a week might be reported in *kilowatt hours* (*kWh*) or *megawatt hours* (*MWh*), and the energy consumed by a battery during a week might be reported in *milliwatt hours* (*mWh*).

If power utilization is constant, the energy consumed can be computed easily by multiplying power utilization, *P*, by the time the power is used. For example, during the time period from t_0 to t_1, the energy used is given by:

$$E \quad = \quad P \times (\, t_1 - t_0 \,) \tag{20.2}$$

A system that uses exactly 6 kilowatts during an hour has an energy consumption of 6 kWh as does a system that has an energy consumption of 3 kilowatts for a period of two hours.

As we described above, most systems do not consume power at a constant rate. Instead, the power consumption varies over time. To capture the idea that power varies continuously, we define energy to be the integral of instantaneous power over time:

$$E \quad = \quad \int_{t=t_0}^{t_1} P(t)\, dt \tag{20.3}$$

Although power is defined to be an instantaneous measure that can change over time, some electronic systems specify a value known as the *average power*. Recall that power is the rate at which energy is used, which means the average power over a time interval can be computed by taking the amount of energy used during the interval and dividing by the time:

$$P_{avg} = \frac{E}{(t_1 - t_0)} \tag{20.4}$$

20.4 Power Consumption By A Digital Circuit

Recall that a digital circuit is created from logic gates. At the lowest level, all logic gates are composed of transistors, and transistors consume power in two ways†:

- Switching or dynamic power (denoted P_s or P_d)
- Leakage power (denoted P_{leak})

Switching Power. The term *switching* refers to a change in the output in response to an input. When one or more inputs of a gate change, the output may change. A change in output can only occur because electrons flow through transistors. Individual transistors consume more power during switching, which means that the total power for the system increases.

Leakage Power. Although we think of a digital circuit as having a binary value (on or off), solid state physicists realize that transistors are imperfect switches. That is, even when a transistor is off, a few electrons can penetrate the semiconductor boundary. Therefore, whenever power is supplied to a digital circuit, some amount of current will always flow, even if the outputs are not switching. We use the term *leakage* to refer to current that flows when a circuit is not operating.

For a given transistor, the amount of leakage current is insignificant. However, a single processor can have a billion transistors, meaning that the aggregate leakage current can be quite high. In fact, for some digital systems, the leakage current accounts for more than half of the power utilization. The point can be summarized:

> *In a typical computing system, 40 to 60 percent of the power the system uses is leakage power.*

A further point is important in the discussion of power management. The basic principle is that leakage always occurs when power is present:

> *Leakage current can only be eliminated by removing power from a circuit.*

†In addition to the two major sources, a minor amount of *short-circuit power* is consumed because CMOS transistors form a brief connection between the power source and ground when switching.

20.5 Switching Power Consumed By A CMOS Digital Circuit

Our focus is using software to manage the power use of a digital circuit. To understand power management techniques, we need a few basic concepts. First, we will consider the total energy consumed by switching. The energy, required for a single change of a gate is denoted E_d, and is given by:

$$E_d \quad = \quad \frac{1}{2} \; C \;\; V_{dd}^2 \tag{20.5}$$

where C is a value of capacitance that depends on the underlying CMOS technology, and V_{dd} is the voltage at which the circuit operates†.

To understand the power consequences of Equation 20.5, consider a clock. The clock generates a square wave at a fixed frequency. Suppose the clock signal is connected to an inverter. The inverter output will change twice during a clock cycle, once when the clock goes from zero to one and once when the clock goes from one back to zero. Therefore, if the clock has period T_{clock}, the average power used is:

$$P_{avg} \quad = \quad \frac{C \;\; V_{dd}^2}{T_{clock}} \tag{20.6}$$

The frequency of the clock is the inverse of the period:

$$F_{clock} \quad = \quad \frac{1}{T_{clock}} \tag{20.7}$$

which means we can rewrite Equation 20.6 in terms of clock frequency:

$$P_{avg} \quad = \quad C \;\; V_{dd}^2 \;\; F_{clock} \tag{20.8}$$

One additional term is used to compute the average power: a fraction of the circuit whose outputs are switching. We use α to denote the fraction, $0 \leq \alpha \leq 1$, which makes the final form of Equation 20.8 for average power:

$$P_{avg} \quad = \alpha \quad C \;\; V_{dd}^2 \;\; F_{clock} \tag{20.9}$$

Equation 20.9 captures the three main components of power that are pertinent to the following discussion. Constant C is a property of the underlying technology and cannot be changed easily. Thus, the three components that can be controlled are:

- The fraction of the circuit that is active, α
- The clock frequency, F_{clock}
- The voltage in the circuit, V_{dd}

†The notation V_{dd} is used to specify the voltage used to operate a CMOS circuit; the notation V (voltage) can be used if the context is understood.

20.6 Cooling, Power Density, And The Power Wall

Recall that instantaneous power use is often associated with data centers or other large installations where a key aspect is peak power utilization. In addition to the question of whether an electric utility is able to deliver the megawatts needed during peak use, designers focus on two other aspects of power use: cooling and power density.

Cooling. When a digital device operates, it generates heat. A huge power load means many devices are operating, and each device is generating heat. Consequently, the heat being produced is related to the power being consumed. All electronic circuits must be cooled or circuits will overheat and burn out. For the smallest devices, enough heat escapes to the surrounding air that no further cooling is needed. For medium-size devices, cooling requires a fan that blows cold air across the circuits constantly; the air must be brought in through a *Heating, Ventilation, and Air Conditioning* (*HVAC*) system. In the most extreme cases, air cooling is insufficient, and a form of liquid cooling is required.

Power Density. Although the total amount of heat a circuit produces dictates the total cooling capacity required, another aspect of heat is important: the concentration of heat in a small area. In a data center, for example, if many computers are placed adjacent to one another, they can overheat. Thus, spacing is added between computers and between racks of computers to permit cool air to flow through the racks and remove heat.

Power density is also important on an individual integrated circuit, where power density refers to the amount of power that is dissipated per a given area of silicon. For many years, the semiconductor industry followed Moore's Law. The size of an individual transistor continued to shrink, and every eighteen months, the number of transistors that fit on a single chip doubled. However, following Moore's Law had a negative aspect: power density also increased. As power density increases, the amount of heat generated per unit area increases, which means that a modern processor produces much more heat per square centimeter than earlier processors.

Consequently, packing transistors closer together has led to a major problem: we are reaching the limits of the rate at which heat can be removed from a chip. Engineers refer to the limit as the *power wall* because it means power cannot be increased. With current cooling technologies, the limit can be approximated:

$$PowerWall \quad \approx \quad 100 \ \frac{watts}{cm^2} \tag{20.10}$$

20.7 Energy Use

Unlike power, which measures instantaneous flow of current, energy measures the total power consumed over a given time interval. A focus on energy is especially pertinent to portable devices that use batteries. We can think of a battery as a bucket of energy, and imagine the device extracting energy as needed. The total time a battery can power a device (measured in milliwatt hours) is derived from the amount of energy in the battery.

Modeling a battery as a bucket of energy (analogous to a bucket of water) is overly simplistic. However, three aspects of water buckets apply to batteries. First, like water in a bucket, the energy stored in a battery can evaporate. In the case of a battery, chemical and physical processes are imperfect — internal resistance allows a trivial amount of current to flow inside the battery. Although the flow is almost imperceptible, allowing a battery to sit for a long time (e.g., a year) will result in loss of charge. Second, just as some of the water poured from a bucket is likely to spill when extracting energy from a battery, some of the energy is lost. Third, energy can be removed from a battery at various rates, just as water can be extracted from a bucket at various rates. The important idea behind the third property is a battery becomes more efficient at lower current levels (i.e., lower power levels). Thus, designers look for ways to minimize power that a battery operated device consumes.

20.8 Power Management

The above discussion shows that reducing power consumption is desirable in all cases. In a large data center, reducing power consumption reduces the heat generated. For a small portable device, reducing power consumption extends the battery life. Two questions arise: what methods can be used to reduce power consumption, and which of the power reduction techniques can be controlled by software?

Recall from Equation 20.9†, three primary factors contribute to power consumption: α, the fraction of a circuit that is active, F_{clock}, the clock frequency, and V_{dd}, the voltage used to operate a circuit. The next sections describe how voltage and frequency can be used to reduce power consumption; a later section considers the fraction of a circuit that is active.

20.8.1 Voltage And Delay

Because power utilization depends on the square of the voltage, lowering voltage will produce the largest reduction in power. However, voltage is not an independent variable. First, decreasing voltage increases *gate delay*, the time a gate takes to change its outputs after inputs change. A processor is designed carefully so that all hardware units operate according to the clock. If the delay for a single gate becomes sufficiently large, the delay across an entire hardware unit (many gates) will exceed the design specification.

For current technology, the delay can be estimated by:

$$Delay \quad = \beta \; \frac{K \;\; V_{dd}}{(\; V_{dd} - V_{TH} \;)} \tag{20.11}$$

†Equation 20.9 can be found on page 398.

where V_{dd} is the voltage used, V_{TH} is a *threshold voltage* determined by the underlying CMOS technology, K is a constant that depends on the technology, and β is a constant (approximately 1.3 for current technology).

A second aspect of power is related to voltage: leakage current. The leakage current depends on the temperature of a circuit and the threshold voltage of the CMOS technology. Lowering voltage decreases leakage current, but has an interesting consequence: lower voltage means increased delay, which results in more total energy being consumed. To understand why increasing leakage can be significant, recall that leakage can account for 40% to 60% of the power a circuit uses. The point is:

> *Although power depends on the square of voltage, reducing voltage increases delay which increases total energy usage.*

Despite the problems, voltage is the most significant factor in power reduction. Therefore, researchers who work on solid state physics and silicon technologies have devised transistors that operate correctly at much lower voltages. For example, although early digital circuits operated at 5 volts, current technologies used in cell phones operate at lower voltages. A fully charged cell phone battery provides about 4 volts, and the circuits continue to operate as the battery discharges. In fact, some cell phones that use NiMH battery technology can still receive calls with a battery that provides only 1.2 volts, and the phone only declares a battery dead when the voltage falls below 0.8 volts. (Lithium-based batteries tend to die at approximately 3.65 volts.)

20.8.2 Decreasing Clock Frequency

Clock frequency forms a second factor in power utilization. In theory, power is proportional to clock frequency, so slowing the clock will save power. In practice, reducing the clock frequency lowers performance, which may be critical in systems that have real-time requirements (e.g., a system that displays video or plays music).

Interestingly, adjusting the clock frequency can be used in conjunction with a reduction in voltage. That is, a slower clock can accommodate the increased delays that a lower voltage causes. Thus, if a designer decreases the clock frequency as voltage is decreased, performance will suffer but the circuit will operate correctly.

When both clock frequency and voltage are reduced, the resulting reduction in power can be dramatic. In one specific case, reducing the frequency to one-half the original rate allowed the voltage to be divided by 1.7. Because voltage is squared in the power equation (Equation 20.9), reducing the voltage allows the resulting power to be reduced dramatically. For the example, the resulting power was approximately 15% of the original power. Although the savings depend on the technology being used, the general idea can be summarized:

If a circuit can deliver adequate performance with a reduced clock frequency, power can be cut dramatically because reducing the clock frequency also allows voltage to be reduced.

Intel has invented an interesting twist on reduced clock frequency by permitting dynamic changes. The idea is straightforward. When the processor is busy, the operating system sets the clock frequency high. If the processor exceeds a preset thermal limit (i.e., overheats) or a power limit (e.g., would drain a battery quickly), the operating system reduces the clock frequency until the processor operates within the prescribed limits. For example, clock frequency might be increased or decreased dynamically by multiples of 100 MHz. If the processor is idle, the clock frequency can also be reduced to save energy. Instead of advertising the capability as dynamic speed reduction, Intel marketing turns the situation around and advertises the feature as *Turbo Boost*.

20.8.3 Slower Clock Frequency And Multicore Processors

In the early 2000s, at the same time power utilization was becoming a problem, chip vendors introduced *multicore processors*. On the surface, a shift to multicore architectures seems counterproductive because two cores will require twice as much power as a single core. Of course, the cores may share some of the circuitry (e.g., a memory or bus interface), which means the power consumption of a dual-core chip will not be exactly double the power consumption of a single core chip. However, a second core adds substantial additional power requirements.

Why would vendors introduce more cores if reducing power consumption is important? To understand, look carefully at clock frequency. Before multicore chips appeared, clock frequency increased every few years as new processors appeared. We know from the above discussion that slowing down a clock to one-half of its original speed allows voltage to be lowered and cuts power consumption significantly. Now consider a dual-core chip. Suppose that each core runs at one-half the clock frequency of a single-core chip. The computational power of the dual-core version is still approximately the same as a single core that runs twice as fast. In terms of power utilization, however, the voltage can be reduced, which means that each of the two cores takes a fraction, F, of the power required by the single-core version. As a result, the multicore chip takes approximately $2F$ as much power as the single core version. Provided F is less than 50%, the slower dual-core chip consumes less power. In the example above, F is 15%, which means a dual-core chip will provide equivalent computational power at only 30% of the original power requirements. We can summarize:

A multicore chip in which each core runs at a slower clock frequency and lower voltage can deliver approximately the same computational capability as a single core chip while incurring significantly lower power utilization.

Of course, the discussion above makes an important assumption about multicore processing. Namely, it assumes that computation can be divided among multiple cores. Unfortunately, Chapter 18 points out that experience with parallelism has not been promising. For computations where a parallel approach is not feasible, a slow clock can make the system unusable. Even in cases where some parallelism is feasible, memory contention and other inefficiencies can result in disappointing performance. When parallel processing is used to handle multiple input items at the same time, overall throughput from two cores can be the same as that of a single, faster core. However, latency (i.e., the time required to process a given item) is higher. Finally, one should remember that the discussion has focused on switching power — leakage can still be a significant problem.

20.9 Software Control Of Energy Use

Software on a system usually has little or no ability to make minor increases or decreases in the voltage used. Instead, software is often restricted to two basic operations:

- Clock gating
- Power gating

Clock Gating. The term refers to reducing the clock frequency to zero which effectively stops a processor. Before a processor can be stopped, a programmer must arrange for a way to restart it. Typically, the code image is kept in memory, and the memory retains power. Thus, the image remains ready whenever the processor restarts.

Power Gating. The term refers to cutting off power from the processor. A special solid state device that has extremely low leakage current is used to cut off power. As with clock gating, a programmer must arrange for a restart, either by saving and then restoring a copy of the memory image or by ensuring that the memory remains powered on so the image is retained.

Systems that offer power gating capabilities do not apply gating across the entire system. Instead, the system is divided into *islands*, and gating is applied to some islands while others continue to operate normally. Memory cache forms a particularly important power island — if power is removed from a memory cache, all cached data will be lost. We know from Chapter 12 that caching is important for performance. Therefore, a memory cache can be placed in a power island that is not shut down when power is removed from other parts of the processor.

Some processors extend the idea to provide a set of *low power modes* that software can use to reduce power consumption. Vendors use a variety of names to describe the modes, such as *sleep*, *deep sleep*, and *hibernation*. We will use the generic names *LPM0*, *LPM1*, *LPM2*, *LPM3*, and *LPM4*. In general, low power modes are arranged in a hierarchy. LPM0 turns off the least amount of circuitry and has the fastest recovery. LPM4, the deepest sleep mode, turns off almost the entire processor. As a consequence, restarting from LPM4 takes much longer than other low power modes.

20.10 Choosing When To Sleep And When To Awaken

Two questions must be answered: when should a system enter a sleep mode, and when should it awaken? Choosing when to awaken from sleep mode is usually straightforward: wake up *on demand*. That is, the hardware waits until an event occurs that requires the processor, and the hardware then moves the processor out of sleep mode. For example, a screen saver restarts the display whenever a user moves a mouse, touches a touch-sensitive screen, or presses a key on a keyboard.

The question of when to enter a low power mode is more complex. The motivation is to reduce power utilization. Therefore, we want to *gate* power to a subsystem (i.e., turn it off) if the subsystem will not be needed for a reasonably long time. Because we usually cannot know future requirements, most systems employ a heuristic to estimate when a subsystem will be needed: if a sufficiently long period of inactivity occurs, assume the subsystem will remain inactive for a while longer. Typically, if a processor or a device remains inactive for N seconds, the processor or device enters a sleep mode. The heuristic can also be applied to cause deeper sleep — if a processor remains in a light sleep state for K seconds, the hardware moves the processor to a deeper sleep state (i.e., additional parts of the processor are turned off).

What value of N should be used as a timeout for sleep mode? Subsystems that provide interaction with a human user typically allow the user to choose a timeout. For example, a screen saver allows a user to specify how long the input devices should remain idle before the screen saver runs. Allowing users to specify a timeout means that each user can tailor the system to their needs.

Choosing a timeout for a system that does not involve human preference requires a more careful analysis. A simplified model will help illustrate the calculation. For the model, we will assume two states: a *RUN* state in which the processor runs with full power and an *OFF* state in which all power is removed. When the processor makes a transition, some time elapses, which we denote $T_{shutdown}$ and T_{wakeup}. Figure 20.1 illustrates the simplified model.

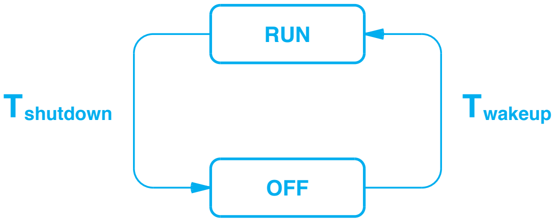


Figure 20.1 A simplified model of transitions among low power modes.

Power is used for each transition (i.e., to save state information or prepare I/O devices for the transition). To make calculations easier, we will assume the power used during a transition is constant. Therefore, the energy required for a transition can be calculated by multipling the power used by the time that elapses:

$$E_{shutdown} \quad = \quad E_s \quad = \quad P_{shutdown} \; \times \; T_{shutdown} \tag{20.12}$$

and

$$E_{wakeup} \quad = \quad E_w \quad = \quad P_{wakeup} \; \times \; T_{wakeup} \tag{20.13}$$

Understanding the energy required for transitions and the energy used when the system runs and when it is shut down allows us to assess potential energy savings. In essence, shutting down is beneficial if shutdown, sleep, and later wakeup consume less energy than continuing to run over the same time interval.

Let t be the time interval being considered. If we assume the power used by the running system is constant, the energy consumed when the system remains running for time t is:

$$E_{run} \quad = \quad P_{run} \; \times \; t \tag{20.14}$$

The energy consumed if the system is put into sleep mode for time t consists of the energy required for each of the transitions plus P_{off}, the energy used (if any) while the processor is shut down:

$$E_{sleep} \quad = \quad E_s \; + \; E_w \; + \; P_{off} \; (\; t \; - \; T_{shutdown} \; - \; T_{wakeup} \;) \tag{20.15}$$

Shutting down the system will be beneficial if:

$$E_{sleep} \quad < \quad E_{\,run} \tag{20.16}$$

By using Equations 20.12 through 20.15, the inequality can be expressed in terms of a single free variable, the time interval t. Therefore, it is possible to compute a *break-even point* that specifies the minimum value of t for which shutting down saves energy.

Of course, the analysis above is based on a simplified model. Power usage may not remain constant; the time and power required for transitions may depend on the state of the system. More important, the analysis focuses on energy consumed by switching and ignores leakage. However, the analysis does illustrate a basic point:

> *Even for a simplified model with only one low power state, details such as the energy used during state transitions complicate the decision about when to move to low power mode.*

20.11 Sleep Modes And Network Devices

Many devices have a low power mode that is used to save energy. For example, a printer usually sleeps after *N* minutes of inactivity. Similarly, wireless network adapters can enter a sleep mode to reduce power consumption. For a network adapter, handling output (transmission) is trivial because the adapter can be awakened whenever an application generates an outgoing packet. However, input (reception) poses a difficult challenge for low power mode because a computer cannot know when another computer will send a packet.

As an example, the Wi-Fi (802.11) standard includes a *Power Saving Polling* (*PSP*) mode. To save power, laptops and other devices using Wi-Fi shut down and only wake up periodically. We use the term *duty-cycle* to characterize the repeated cycle of a device running and then being shut down. A radio must be up when an access point transmits. A Wi-Fi base station periodically sends a beacon that includes a list of recipients for which the base station has undelivered packets. The beacon is frequent enough so a device is guaranteed to receive the beacon during the part of the duty cycle when they are awake. If a device finds itself on the recipient list, the device remains awake to receive the packet.

Two basic approaches have been used to allow a network adapter to sleep without missing packets indefinitely. In one approach, each device synchronizes its sleep cycles with the base station. In the other approach, a base station transmits each packet many times until the receiver wakes up and receives it.

20.12 Summary

Power is an instantaneous measure of the rate at which energy is used; energy is the total amount of power used over a given time. A digital circuit uses dynamic or switching power (i.e., an output changes in response to the change of an input) and leakage power. Leakage can account for 40 to 60 percent of the power a circuit consumes.

Power consumption can be reduced by making parts of a circuit inactive, reducing the clock frequency, and reducing the voltage. Reducing the voltage has the largest effect, but also increases delay. Power density refers to the concentration of power in a given space; power density is related to heat. The *power wall* refers to the limit of approximately 100 watts per cm^2 that gives the maximum power density for which heat can be removed from a silicon chip using current cooling technologies.

Clock gating and power gating can be used to turn off a circuit (or part of a circuit). For devices that use battery power, the overall goal of power management systems is a reduction in total energy use. Because moving into and out of a low power (sleep) mode consumes energy, sleeping is only justified if the energy required for sleep mode is less than the energy required to remain running. A simplified model shows that the computation involves the cost to shut down and the cost to wake up.

Devices can also use low-power modes. Network interfaces pose a challenge because the interface must be awake to receive packets and a computer does not always know when packets will arrive. The Wi-Fi standard includes a Power Saving Polling mode.

EXERCISES

20.1 Estimate the amount of power required for the *Tianhe-2* supercomputer described on page 376. Hint: start by finding an estimate of the number of watts used by a single processor.

20.2 Suppose the frequency of a clock is reduced by 10% and all other parameters remain the same. How much is the power reduced?

20.3 Suppose the voltage, V_{dd}, is reduced by 10% and all other parameters remain the same. How much is the power reduced?

20.4 Use Equation 20.16 to find a break-even value for t.

20.5 Extend the model in Figure 20.1 to a three-state system in which the processor has both a *sleep* mode and a *deep sleep* mode.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter Contents

21

Assessing Performance

21.1 Introduction

Earlier parts of the text cover the three fundamental mechanisms that computer architects use to construct computer systems: processors, memories, and I/O devices. They characterize each mechanism, and explain the salient features. Previous chapters consider two techniques used to increase computational performance: parallelism and pipelining.

This chapter takes a broader view of performance. It examines how performance can be measured, and discusses how an architect evaluates an instruction set. More important, the chapter presents Amdahl's law, and explains consequences for computer architecture.

21.2 Measuring Computational Power And Performance

How can we measure computational power? What makes one computer system perform better than another? These questions have engendered research in the scientific community, caused heated debate among representatives from the sales and marketing departments of commercial computer vendors, and resulted in a variety of answers.

The chief problem that underlies performance assessment arises from the flexibility of a general-purpose computer system: a computer is designed to perform a variety of tasks. More important, because optimization involves choosing among alternatives, optimizing the architecture for a given task means that the architecture will be less than optimal for other tasks. Consequently, the performance of a computer system depends on how the system is used.

We can summarize:

Because a computer is designed to perform a wide variety of tasks and no architecture is optimal for all tasks, the performance of a system depends on the task being performed.

The dependency between performance and the task being performed has two important consequences. First, it means that many computer vendors can each claim that they have the most powerful computer. For example, a vendor whose computer performs matrix multiplication at high speed uses matrix multiplication examples when measuring performance, while a vendor whose computer performs integer operations at high speed uses integer examples when measuring performance. Both vendors can claim that their computer performs best. Second, from a scientific point of view, we can see that no single measure of computer system performance suffices for all cases. The point is fundamental to understanding performance assessment:

A variety of performance measures exist because no single measure suffices for all situations.

21.3 Measures Of Computational Power

Recall that early computer systems consisted of a central processor with little or no I/O capability. As a consequence, early measures of computer performance focused on the execution speed of the CPU. Even when performance measures are restricted to a CPU, however, multiple measures apply. The most important distinction arises between computer systems optimized for:

- Integer computation
- Floating point computation

Because scientific and engineering calculations rely heavily on floating point, applications that employ floating point are often called *scientific applications*, and the resulting computation is known as *scientific computation*. When assessing how a computer performs on scientific applications, engineers focus entirely on the performance of floating point operations. They ignore the speed of integer operations, and measure the speed of floating point operations (specifically, floating point addition, subtraction, multiplication and division). Of course, addition and subtraction are generally faster than multiplication and division, and a program contains other instructions (e.g., instructions to call functions and control iteration). On many computers, however, a floating point operation takes so much longer than a typical integer instruction that floating point computation dominates the overall performance of a program.

Rather than reporting the time required to perform a floating point operation, engineers report the number of floating point operations that can be performed per unit time. In particular, the primary measure is given as the average number of floating point operations the hardware can execute per second (*FLOPS*).

Of course, floating point speed is only pertinent for scientific computation; the speed of floating point hardware is irrelevant to programs that use integers. More important, a measure of FLOPS does not make sense for a RISC processor that does not offer floating point instructions. Thus, as an alternative to measuring floating point performance, a vendor may choose to exclude floating point and report the average number of other instructions that a processor can execute per unit time. Typically, such vendors measure *millions of instructions per second* (*MIPS*).

Simplistic measures of performance such as MIPS or FLOPS only provide a rough estimate of performance. To see why, consider the time required to execute an instruction. For example, consider a processor on which floating point multiplication or division takes twice as long as floating point addition or subtraction. If we assume that an addition or subtraction instruction takes Q nanoseconds and weight each of the four instruction types equally, the average time the computer takes to perform a floating point instruction, T_{avg} is:

$$T_{avg} = \frac{Q + Q + 2 \times Q + 2 \times Q}{4} = 1.5\,Q \quad ns\ per\ instr. \tag{21.1}$$

However, when the computer performs addition and subtraction, the time required is only Q nanoseconds per instruction (i.e., 33% less than the average). Similarly, when performing multiplication or division, the computer requires $2 \times Q$ nanoseconds per instruction (i.e., 33% more than the average). In practice, the times required for addition and division can differ by more than a factor of two, which means that actual performance can vary by more than 33%. An exercise considers one possible ratio.

The point is:

> *Because some instructions take substantially longer to execute than others, the average time required to execute an instruction only provides a crude approximation of performance. The actual time required depends on which instructions are executed.*

21.4 Application Specific Instruction Counts

How can we produce a more accurate assessment of performance? One answer lies in assessing performance for a specific application. For example, suppose we need to know how a floating point hardware unit will perform when multiplying two $N \times N$ matrices. By examining the program, it is possible to derive a set of expressions that give the number of floating point additions, subtractions, multiplications, and divisions that will be performed as a function of N. For example, assume that multiplying a pair of

$N \times N$ matrices requires N^3 floating point multiplications and $N^3 - N^2$ floating point additions. If each addition requires Q nanoseconds and each multiplication requires $2 \times Q$ nanoseconds, multiplying two matrices will require a total of:

$$T_{total} = 2 \times Q \times N^3 + Q \times (N^3 - N^2) \tag{21.2}$$

As an alternative to precise analysis, engineers use a weighted average. That is, instead of calculating the exact number of times each instruction is executed, an approximate percentage is used. For example, suppose a graphics program is run on many input data sets, the number of floating point operations is counted to obtain the list in Figure 21.1.

Instruction Type	Count	Percentage
Add	8513508	72
Subtract	1537162	13
Multiply	1064188	9
Divide	709458	6

Figure 21.1 Example of instruction counts for a graphics application run on many input values. The third column shows the relative percentage of each instruction type.

Once a set of instruction counts has been obtained, the performance of hardware can be assessed by using a weighted average. When the graphics application is run on the hardware described above, we expect the average time for each floating point instruction to be:

$$T_{avg'} = .72\ Q + .13\ Q + .09 \times 2\ Q + .06 \times 2\ Q \ = \ 1.16\ Q \ \ \textit{ns per instr.} \tag{21.3}$$

As the example shows, a weighted average can differ significantly from a uniform average. In this case, the weighted average is 23% less than the average in Equation 21.1 that was obtained using uniform instruction weights†.

21.5 Instruction Mix

Although it provides a more accurate measurement of performance, the weighted average example above only applies to one specific application and only assesses floating point performance. Can we give a more general assessment? One approach has become popular: use a large set of programs to obtain relative weights for each type of instruction, and then use the relative weights to assess the performance of a given architecture. That is, instead of focusing on floating point, keep a counter for each instruction type (e.g., integer arithmetic instructions, bit shift instructions, subroutine calls,

†Equation 21.1 can be found on page 413.

conditional branches), and use the counts and relative weights to compute a weighted average performance.

Of course, the weights depend on the specific programs chosen. Therefore, to be as accurate as possible, we must choose programs that represent a typical workload. Architects choose an *instruction mix* that represents typical programs.

In addition to helping assess performance of a computer, an instruction mix helps an architect design an efficient instruction set. The architect drafts a tentative instruction set, assigns an expected cost to each instruction, and uses weights from the instruction mix to see how the proposed instruction set will perform. In essence, the architect uses the instruction mix to evaluate how the proposed architecture will perform on typical programs. If the performance is unsatisfactory, the architect can change the design.

We can summarize:

> *An instruction mix consists of a set of instructions along with relative weights that have been obtained by counting instruction execution in a set of example programs. An architect can use an instruction mix to assess how a proposed architecture will perform.*

21.6 Standardized Benchmarks

What instruction mix should be used to compare the performance of two architectures? To answer the question, we need to know how the computers will be used: the programs the computers are intended to run, and the type of input the programs will receive. In essence, we need to find a set of applications that are typical. Engineers and architects use the term *benchmark* to refer to such programs — a benchmark provides a standard workload against which a computer can be measured.

Of course, devising a benchmark is difficult, and the community does not benefit if each vendor creates a separate benchmark. To solve the problem, an independent not-for-profit corporation was formed in the 1980s. Named *Standard Performance Evaluation Corporation* (*SPEC*), the corporation was created to “establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers”†. SPEC has devised a series of standard benchmarks that are used to compare performance. For example, the *SPEC cint2006* benchmark is used to evaluate integer performance, and the *SPEC cfp2006* benchmark is used to evaluate floating point performance.

The benchmarks produced by SPEC are primarily used for measurement, not design. That is, each benchmark consists of a set of programs that are run and measured. The score that results from running a SPEC benchmark, known as a *SPECmark*, is often quoted in the industry as a vendor-independent measure of computer performance.

Interestingly, SPEC has produced many benchmarks that each test one aspect of performance. For example, SPEC offers six separate benchmarks that focus on integer

†The description is taken from the SPEC bylaws (see http://www.spec.org).

arithmetic and another fourteen benchmarks that focus on various aspects of floating point performance. In addition, SPEC provides benchmarks to assess the power computers consume, performance of a Java environment, and performance of Unix systems running the *Network File System* (*NFS*) for remote file access during software development tasks.

21.7 I/O And Memory Bottlenecks

CPU performance only accounts for part of the overall performance of a computer system. As users of personal computers have realized, a faster CPU or more cores does not guarantee faster response for all computing tasks. A colleague of the author complains that although CPU power increases by an order of magnitude every ten years, the time required to launch an application seems to increase.

What prevents a faster CPU from increasing the overall speed? We have already seen one answer: the Von Neumann bottleneck (i.e., memory access). Recall that the speed of memory can affect the rate at which instructions can be fetched as well as the rate at which data can be accessed. Thus, rather than merely measuring CPU performance, some benchmarks are designed to measure memory performance. The memory benchmark consists of a program that repeatedly accesses memory. Some memory benchmarks are designed to test sequential access (i.e., access to contiguous bytes), while others are designed to test random access. More important, memory benchmarks also make repeated references to a memory location to test memory caching.

As the chapters on I/O point out, peripheral devices and the buses over which peripheral devices communicate can also form a bottleneck. Thus, some benchmarks are designed to test the performance of I/O devices. For example, a benchmark to test a disk will repeatedly execute *write* and *read* operations that each transfer a block of data to the disk and then read the data back. As with memory, some disk benchmarks focus on measuring performance when accessing sequential data blocks, and other benchmarks focus on measuring performance when accessing random blocks.

21.8 Moving The Boundary Between Hardware And Software

One of the fundamental principles that underlies computer performance arises from the relative speed of hardware and software: hardware (especially hardware designed for a special purpose) is faster than software. As a consequence, moving a given function to hardware will result in higher performance than executing the function in software. In other words, an architect can increase overall performance by adding special-purpose hardware units.

A corollary arises from an equally important principle: software provides much more flexibility than hardware. The consequence is that functionality implemented with hardware cannot be changed. Therefore, an architect can increase overall flexibility and generality by allowing software to handle more functions. The recent use of FPGAs is an example of hardware functions moving to software — instead of building a chip with fixed gates, an FPGA allows functions in the design to be programmed.

The point is that hardware and software represent a tradeoff:

> *Performance can be increased by moving functionality from software to hardware; flexibility can be increased by moving functionality from hardware to software.*

21.9 Choosing Items To Optimize, Amdahl's Law

When an architect needs to increase performance, the architect must choose which items to optimize. Adding hardware to a design increases cost; special-purpose, high-speed hardware is especially expensive. Therefore, an architect cannot merely specify that arbitrary amounts of high-speed hardware be used. Instead, a careful choice must be made to select functions that will be optimized with high-speed hardware and functions that will be handled with conventional hardware.

How should the choice be made? A computer architect, Gene Amdahl, observed that it is a waste of resources to optimize functions that are seldom used. For example, consider the hardware used to handle division by zero or the circuitry used to power down a computer system. There is little point in optimizing such hardware because it is seldom used.

Amdahl suggested that the greatest gains in performance are made by optimizing functions that account for the most time. His principle, which is known as *Amdahl's law*, focuses on operations that each require extensive computation or operations that are performed most frequently. Usually, the principle is stated in a form that refers to the potential for speedup:

> Amdahl's Law: *the performance improvement that can be realized from faster hardware technology is limited to the fraction of time the faster technology can be used.*

Amdahl's law can be expressed quantitatively by giving the overall speedup in terms of the fraction of time-enhanced hardware is used and the speedup that the enhancement delivers. Equation 21.4 gives the overall speedup:

$$Speedup_{overall} \;=\; \frac{1}{1 - Fraction_{enhanced} + \dfrac{Fraction_{enhanced}}{Speedup_{enhanced}}} \qquad (21.4)$$

The equation works for two extremes. If the enhanced hardware is never used (i.e., the fraction is 0), there is no speedup, and Equation 21.4 results in a ratio of 1. If the enhanced hardware is used 100% of the time (i.e., the fraction is 1), the overall speedup equals the speedup of the enhanced hardware. At fractional values between 0 and 1, the overall speedup is weighted according to how much the enhanced hardware is used.

21.10 Amdahl's Law And Parallel Systems

Chapter 18 discusses parallel architectures, and explains that performance has been disappointing. In particular, overhead from communication among processors and contention for shared resources such as memory and I/O buses limit the effective speed of the system. As a result, parallel systems that contain *N* processors do not achieve *N* times the performance of a single processor.

Interestingly, Amdahl's Law applies directly to parallel systems and explains why adding more processors does not help. The speedup that can be achieved by optimizing the processing power (i.e., adding additional processors) is limited to the amount of time the processors are being used. Because a parallel system spends most of the time waiting for communication or bus access rather than using the processors, adding additional processors does not produce a significant increase in performance.

21.11 Summary

A variety of performance measures exist. Simplistic measures of processor performance include the average number of floating point operations a computer can perform per second (FLOPS) or the average number of instructions the computer can execute per second (MIPS). More sophisticated measures use a weighted average in which an instruction that is used more often is weighted more heavily. Weights can be derived by counting the instructions in a program or a set of programs; such weights are specific to the application(s) used. We say that weights, which are useful in assessing an instruction set, correspond to an instruction mix.

A benchmark refers to a standardized program or set of programs used to assess performance; each benchmark is chosen to represent a typical computation. Some of the best-known benchmarks have been produced by the SPEC Corporation, and are known as SPECmarks. In addition to measuring performance of various aspects of integer and floating point performance, SPEC benchmarks are available to measure such mechanisms as remote file access.

Amdahl's Law helps architects select functions to be optimized (e.g., moved from software to hardware or moved from conventional hardware to high-speed hardware). The law states that functions to be optimized should account for the most time. Amdahl's Law explains why parallel computer systems do not always benefit from a large number of processors.

EXERCISES

21.1 Write a C program that measures the performance of integer addition and subtraction operators. Perform at least 10,000 operations and calculate the average time per operation.

21.2 Write a computer program that measures the difference in execution times between integer addition and integer division. Execute each operation 100,000 times, and compare the difference in running times. Repeat the experiment, and verify that no other activities on the computer interfere with the measurement.

21.3 Extend the measurement in the previous exercise to compare the performance of sixteen-bit, thirty-two-bit, and (if your computer supports it) sixty-four-bit integer addition. That is, use *short, int, long,* or *long long* variables as needed. Explain the results.

21.4 Computer professionals commonly use addition, subtraction, multiplication, and division as ways to measure performance of a processor. However, many programs also use logical operations, such as *logical and*, *logical or*, *bit complement*, *right shift*, *left shift*, and so on. Measure such operations, and compare the performance to integer addition.

21.5 If floating point addition and subtraction each take Q microseconds and floating point multiplication and division each take 3Q microseconds, what is the average time required for all four operations?

21.6 Extend the previous exercise and compute the percentage difference between the time for addition and the average time, and the percentage difference between the time for multiplication and the average time.

21.7 In the previous problem, repeat the measurement with compiler optimization enabled and determine the relative speedup.

21.8 Write a program that compares the average time required to perform integer arithmetic operations and the average time required to reference memory. Calculate the ratio of memory cost to integer arithmetic cost.

21.9 Write a program that compares the average times required to perform floating point operations and integer operations. For example, compare the average time required to perform 10,000 floating point additions and the average time required to perform 10,000 integer additions.

21.10 A programmer decides to measure the performance of a memory system. The programmer finds that according to the DRAM chip manufacturer, the time needed to access an integer in the physical memory is 80 nanoseconds. The programmer writes an assembly language program that stores a value into a memory location four billion times, measures the time taken, and computes the average performance. Surprisingly, it only takes an average of 52 nanoseconds per store operation. How is such a result possible?

21.11 Turn the previous exercise around, and state why accurate measurement of a physical memory is difficult.

21.12 A hashing function places values in random loctions in an array called a *hash table*. A programmer finds that even when memory caching is turned off, storing and then looking up 50,000 values in an extremely large hash table (16 megabytes) has worse performance than using the same data in a smaller hash table (16 kilobytes). Explain why.

Chapter Contents

22

Architecture Examples And Hierarchy

22.1 Introduction

Earlier chapters explain the concepts and terminology that are essential to an understanding of computer architecture. The chapters discuss the fundamental aspects of processors, memory, and I/O, and explain the role of each. Previous chapters discuss how parallelism and pipelining are used to improve performance.

This chapter considers a few architecture examples. Instead of introducing new ideas, the chapter shows how the ideas in previous chapters can be used to describe and explain various aspects of digital systems. The examples have been chosen to show a range of possibilities.

22.2 Architectural Levels

Recall from earlier chapters that architecture can be presented at multiple levels of abstraction. To help us appreciate how broadly architectural concepts apply to digital systems, we will explore a hierarchy of architectural specifications. The hierarchy ranges in size from a complete computer system to a small functional unit on a single integrated circuit. We use the terms *system-level architecture* (sometimes called *macroscopic architecture*), *board-level architecture*, and *chip-level architecture* (sometimes called *microscopic architecture*) to characterize the range. For each level, we will see that the concepts from earlier chapters allow us to understand both the basic components and their interconnection. Furthermore, we will see that at a given level, it is

Level	Description
System	A complete computer with processor(s), memory, and I/O devices. A typical system architecture describes the interconnection of components with buses.
Board	An individual circuit board that forms part of a computer system. A typical board architecture describes the interconnection of chips and the interface to a bus.
Chip	An individual integrated circuit that is used on a circuit board. A typical chip architecture describes the interconnection of functional units and gates.

Figure 22.1 Conceptual levels of architecture and the purpose of each.

possible to specify a logical (i.e., conceptual) architecture or to specify a more detailed implementation. Figure 22.1 summarizes the levels we will consider.

22.3 System-level Architecture: A Personal Computer

Conceptually, a personal computer consists of a processor, memory, and a set of I/O devices that all attach to a single bus. In practice, however, even a personal computer contains a complex assortment of buses and interconnection mechanisms that are each designed to fill a specific role.

Some of the variety and complexity in underlying hardware arises from special performance requirements and cost. For example, a video card needs much higher data throughput than a floppy disk, and a high-resolution screen requires more throughput than a low-resolution screen. Unfortunately, the hardware that interconnects a device to a high-speed bus costs significantly more than the hardware that interconnects a device to a low-speed bus, which means that using multiple buses can lower the overall cost of the system.

A second motivation for multiple I/O buses arises from a vendor's desire to provide a low-cost migration path to newer, more powerful systems. That is, a vendor strives to create a processor that offers the advantages of higher performance and more capabilities, while simultaneously retaining the ability to use existing peripheral devices. We use the term *backward compatibility* to characterize the ability to use existing pieces of hardware.

Backward compatibility is especially important for bus architectures because a bus forms the interconnection between an I/O device and a processor. How can a computer vendor devise a new, higher-speed bus while still retaining the ability to attach older peripheral devices? One possibility consists of creating a processor with multiple bus interfaces. A much less expensive answer lies in the use of *bridging*.

22.4 Bus Interconnection And Bridging

The use of bridging for backward compatibility is easy to understand through a historical example. At one point in history, all personal computers used an *Industry Standard Architecture* (*ISA*) bus that was developed by IBM Corporation. Peripheral devices for PCs were designed with an interface for the ISA bus. Later, a higher-speed bus architecture was developed: a *Peripheral Component Interconnect* (*PCI*) bus. The two standards for PC buses are incompatible — an interface that plugs into an ISA bus cannot be connected to a PCI bus. Thus, if a user owns ISA devices, the user is less likely to purchase a computer that only accepts PCI devices.

To entice computer owners to upgrade their computers to a computer with a PCI bus, vendors created a *bridge* to interconnect the new PCI bus and the older ISA bus. Logically, the bridge provides the interconnection that Figure 22.2 illustrates.

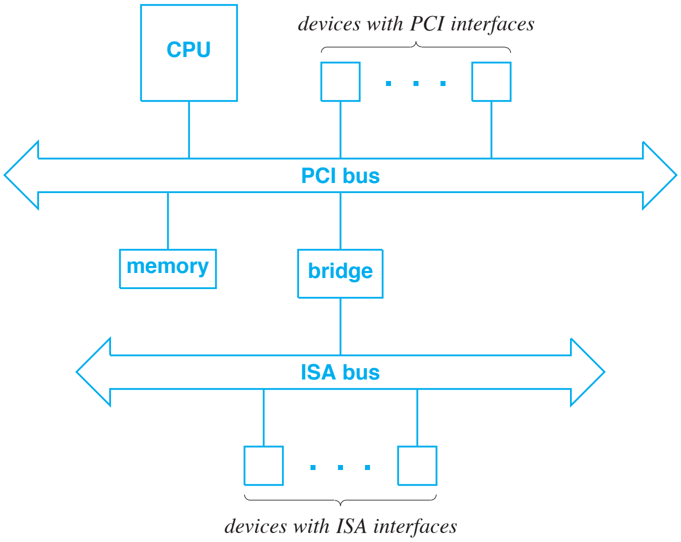


Figure 22.2 Conceptual view of a PC architecture that uses a bridge to interconnect an ISA bus and a PCI bus. The bridge makes it possible to use older ISA devices with a newer processor.

In the figure, the CPU and any I/O devices that have a PCI interface connect directly to a PCI bus. The bridge provides a connection to an ISA bus that is used by I/O devices that have an ISA interface. In the best case, the interconnection provided by a bridge is *transparent*. That is, each side uses a local bus protocol to communicate without knowing about the interconnection — the CPU addresses ISA devices as if they are connected to the PCI bus, and an ISA device responds as if the CPU is connected to the ISA bus.

22.5 Controller Chips And Physical Architecture

Although the architecture illustrated in Figure 22.2 provides a conceptual explanation of a PC architecture, an implementation is much more complex than the figure indicates. First, although a PC provides slots that external devices use to connect to each bus, the PC does not use the same technology internally. Instead, a PC usually contains two special-purpose *controller chips* that provide all the bus and memory interconnections. Second, controller chips are configured to give the illusion of multiple buses.

To understand the need for controller chips, consider the functionality required in a PC. An architect needs to connect the processor, memory, and I/O bus (or buses). In addition to providing electrically compatible interconnections, the architect must design a mechanism that allows one component to communicate with another. For example, both the CPU and I/O devices need to access memory.

Unfortunately, replicating hardware interfaces is expensive. In particular, an architect cannot afford to build a system in which each component has multiple interface units that each handle communication with one other component. For example, although the processor and most I/O devices need to access memory, the cost prohibits an architect from providing a memory interface for each device.

To save effort and expense, architects often adopt the approach of using a centralized *controller chip*. A controller chip contains a set of K hardware interfaces, one for each type of hardware, and forwards requests among them. When a hardware unit needs to access another hardware unit, the request always goes to the controller. The controller translates each incoming request into the appropriate form, and then forwards the request to the destination hardware unit. Similarly, the controller translates each reply.

The key idea is:

> *Architects use a controller chip to provide interconnection among components in a computer because doing so is less expensive than equipping each unit with a set of interfaces or building a set of discrete bridges to interconnect buses.*

22.6 Virtual Buses

A controller chip introduces an interesting possibility. Because a bus is used to communicate, we expect two or more devices to be attached to each bus (e.g., a processor and a disk). In a computer that uses a controller chip, however, it is reasonable to create a bus that contains exactly one connected device. For example, if only one device needs an ISA bus and all others use a PCI bus, a controller chip can be created that uses the ISA protocol to communicate with the ISA device and uses the PCI protocol to communicate with other devices. Even if the controller chip uses the ISA protocol to communicate with the ISA device, the computer will not need slots for ISA devices and will not have a physical ISA bus in the usual sense. That is:

> *A controller chip can provide the illusion of a bus over a direct connection; there is no need for the physical sockets and wiring that is normally used with a bus.*

The concept of a controller chip that can provide the illusion of a bus over a direct connection allows architects to generalize the notion of a bus. Instead of separate physical entities with parallel wires, a silicon chip can be used to create the appearance of a bus. We use the term *virtual bus* to describe the technology. For example, a controller can be created that presents the illusion of one virtual bus per attached device. As an alternative, a controller can be created that combines one or more virtual buses with connections to one or more physical buses. Later sections show examples.

Typically, PC architectures use two controller chips instead of one. The controllers are known informally as the *Northbridge* and *Southbridge* chips; the Northbridge is sometimes called a *system controller*. The Northbridge connects high-speed components, such as the CPU, memories, streaming communications controllers, and an *Advanced Graphics Port* (*AGP*) interface that is used to operate a high-speed graphics display. The Southbridge, which attaches to the Northbridge, provides connectivity for lower-speed components, such as a PCI bus, a Wi-Fi network interface†, an audio device, keyboard, mouse, and similar devices. Figure 22.3 illustrates the physical interconnections in a PC architecture that uses two controller chips.

As the figure shows, a controller chip must accommodate heterogeneity because a controller can connect to multiple bus technologies. In the figure, for example, the Southbridge provides connections for a PCI bus, a USB bus, and an ISA bus. Of course, the controller must follow the rules for each bus. That is, the controller must adhere to the electrical specifications, ensure that all addresses lie within the bus address space, and obey the protocol that defines how devices access and use the bus.

Vendors who manufacture CPUs usually offer a set of controller chips that are designed to interconnect a CPU with standard buses. For example, Intel Corporation offers an *82865PE* chip that provides the functionality of a Northbridge and an *ICH5* chip that provides the functionality of a Southbridge. More important, the Intel processor chip and Intel controller chips are designed to work together: each chip contains an interface that allows the chips to be directly interconnected, and each chip performs the translation necessary to connect heterogeneous devices.

†Networks that operate at gigabit speeds connect to the Northbridge.

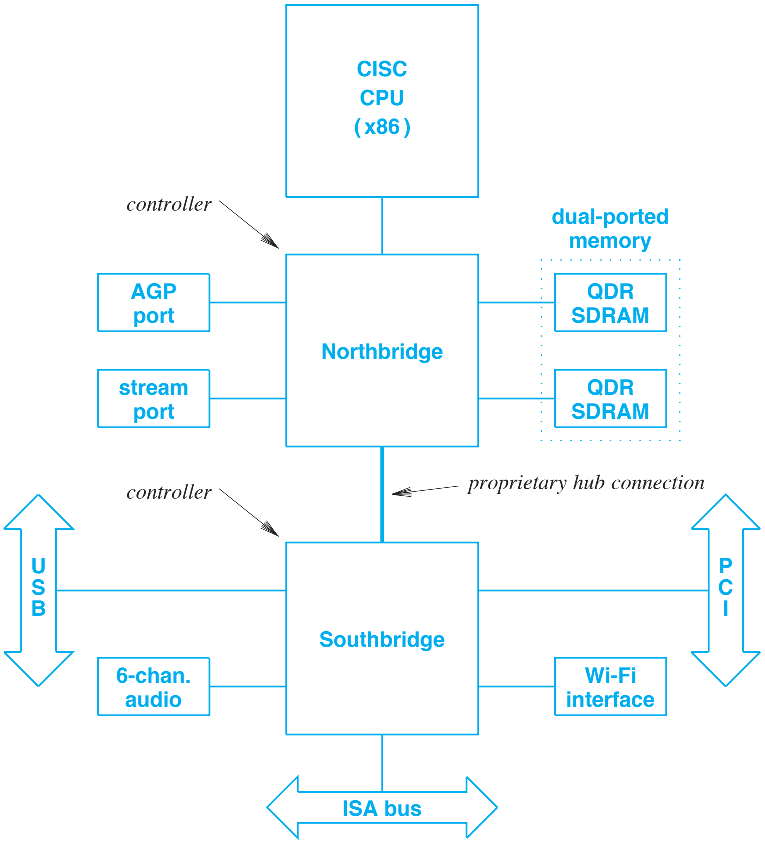


Figure 22.3 Example of a system-level architecture that shows the physical interconnections in a PC that uses two controller chips. Components that require the highest speeds attach to the *Northbridge* controller.

22.7 Connection Speeds

The connections illustrated in Figure 22.3 typically use a parallel hardware interface that has a fixed width and is engineered to operate at a fixed clock rate to deliver a specified throughput. Figure 22.4 lists typical values for the clock rate, width, and throughput of major connections.

Connection	Clock Rate	Width	Throughput†
USB 1.0	33 MHz	32 bits	1.5 MB/s
FCC broadband	–	–	3.1 MB/s
AGP	100–200 MHz	64–128 bits	2.0 GB/s
USB 3.0	up to 500 MHz	32 bits	5.0 GB/s
Memory	200–800 MHz	64–128 bits	6.4 GB/s
PCI 3.0	33 MHz	32 bits	126.0 GB/s
Registers	1000–2000 MHz	64–128 bits	672.0 GB/s

Figure 22.4 Example clock rates, data widths, and throughput for connections in the architecture that Figure 22.3 illustrates.

For comparison purposes, the figure includes the FCC's definition of an Internet connection (25 megabits per second downstream, which is 3.1 megabytes per second) and a register file in a modern processor. Note that transfers in a computer can occur much faster than a broadband Internet connection, and the sustained throughput to registers dwarfs all other throughputs listed in the figure.

22.8 Bridging Functionality And Virtual Buses

As the names *Northbridge* and *Southbridge* imply, the two controllers provide bridging functionality. For example, the Northbridge chip bridges memory, high-speed devices, and the Southbridge chip. The Northbridge presents the CPU with a single, unified address space that includes all of the above. Similarly, the Southbridge combines the PCI bus, ISA bus, and USB bus into a single, unified address space, which becomes part of the address space that the Northbridge presents to the processor.

Interestingly, a set of controllers does not need to bridge all devices into a single address space. Instead, the controller can present the CPU with the illusion of multiple virtual buses. For example, a controller might allow the CPU to access two separate PCI buses: bus number zero contains the CPU and memory, while bus number one contains I/O devices. As an alternative, a controller might present the illusion of three virtual buses: one that contains the CPU and memory, another that contains a high-speed graphics device, and a third that corresponds to the external PCI slots for arbitrary devices. Although it is not particularly interesting to a programmer, the separation is crucial to a hardware designer interested in performance because the controller chip can contain parallel circuitry that allows all virtual buses to operate at the same time.

†Throughput is reported in *Megabytes per second* (*MB/s) and Gigabytes per second* (*GB/s*), where the uppercase *B* emphasizes bytes instead of bits.

22.9 Board-level Architecture

The architecture in Figure 22.3 includes a Wi-Fi interface as one of the units in a personal computer. The role of the interface is straightforward: provide the physical connection between the PC and the Wi-Fi radio, and transfer data that the PC sends over the network as well as data that arrives over the network. Physically, a Wi-Fi interface can be integrated onto the *motherboard* in a laptop or reside on a circuit board in a desktop system. In either case, the logical interconnection remains the same.

A network interface card contains a surprising amount of computational power. In particular, an interface usually contains an embedded processor, instructions in ROM, a buffer memory, an external host interface (e.g., a PCI bus interface), and a connection to the radio transmitter and receiver. Some interface cards use a conventional RISC processor; others use a specialized *network processor* that is optimized for handling network packets. Figure 22.5 illustrates a possible architecture for a LAN interface that uses a network processor.

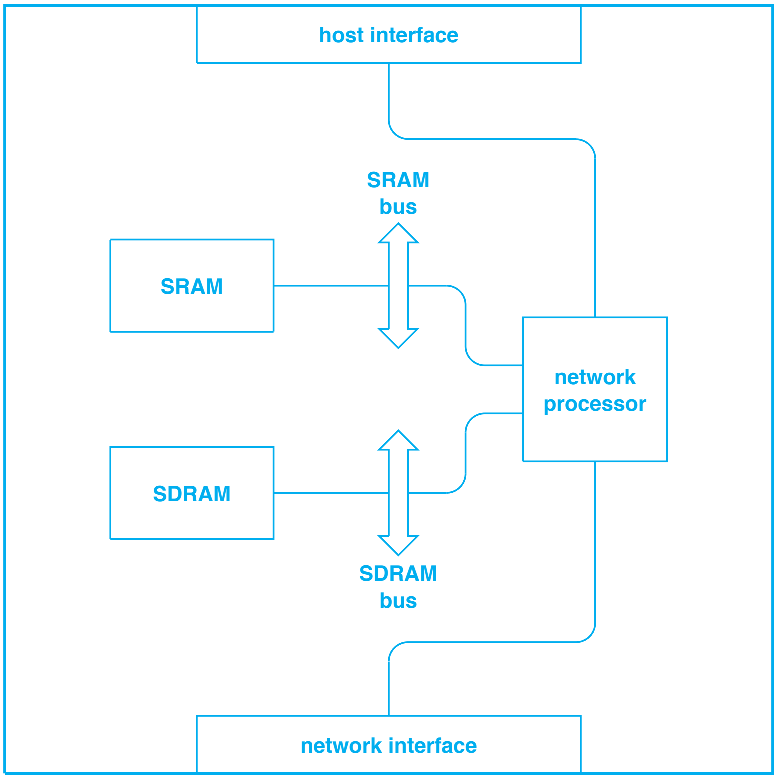


Figure 22.5 Example architecture of a network interface card used for a Wi-Fi device.

Why might a Wi-Fi interface need two types of memory? The primary motivation is cost: although it is faster, SRAM costs more than SDRAM. Thus, a large SDRAM can be used to hold packets, and a small SRAM can be used for values that must be accessed or updated frequently (e.g., instructions for the network processor to execute). In this particular example, the two memory connections are chosen because the network processor described in the next section uses both SRAM and SDRAM.

22.10 Chip-level Architecture

We said that a chip-level architecture describes the internal structure of a single integrated circuit. As an example, consider the network processor in the board-level architecture illustrated in Figure 22.5; the figure uses a rectangle to depict a network processor. If we move to a chip-level architecture, we can examine the internal structure of the chip. Figure 22.6 shows the chip-level architecture of a Netronome network processor†.

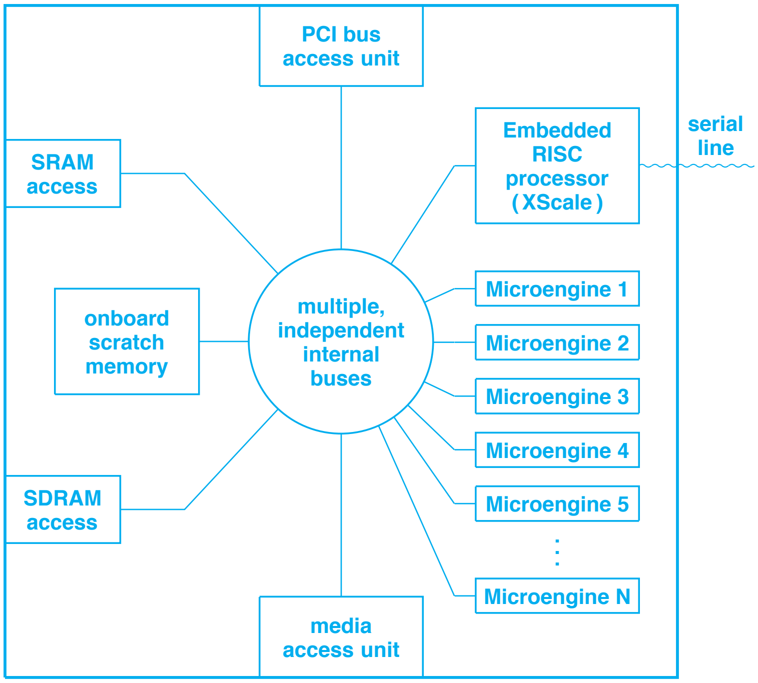


Figure 22.6 Example of a chip-level architecture that shows the major internal components of a Netronome network processor. Access units provide connections outside the chip.

†Intel Corporation designed the network processor, and later sold the design to Netronome.

It is important to remember that the entire figure refers to a single integrated circuit. As the figure shows, the network processor chip contains many items, including various external interfaces, an onboard *scratch* memory that provides high-speed storage, and multiple, independent processors. In particular, the chip contains a set of programmable RISC processors, known as *microengines*†, that operate in parallel as well as an *XScale* RISC processor. The XScale provides a general-purpose processor that manages other processors and provides a management interface. When the network processor operates, the XScale runs a conventional operating system, such as *Linux*. To indicate that processors are part of an integrated circuit, we say they are *embedded*.

Details of the network processor and each of its processors are irrelevant. The important point is to understand that more detail is revealed at each architectural level. In this case, we have seen that although a single integrated circuit can contain many functional units, the structure of the circuit is only revealed in a chip-level diagram; the chip structure remains hidden in a board-level diagram. We can summarize:

> *Each level of an architecture reveals details that are hidden by higher levels. A chip-level architecture specifies the internal structure of an integrated circuit that is hidden in a board-level architecture.*

22.11 Structure Of Functional Units On A Chip

As a final example of architectural levels, we will examine how it is possible to describe the architecture of one component on a chip. Figure 22.7 shows the SRAM access unit from Figure 22.6. The internal structure of the memory access unit is quite complex.

22.12 Summary

The architecture of a digital system can be viewed at several levels of abstraction. A system-level architecture shows the structure of an entire computer system, a board-level architecture shows the structure of each board, and a chip-level architecture shows the internal structure of an integrated circuit. At each successive level, details are revealed that remain hidden in previous levels.

As an example, the chapter presents a hierarchy of architectures that shows the structure of a personal computer, a Wi-Fi network interface board in the computer, and a network processor on the interface board. Finally, we saw that a chip-level architecture can be further refined by looking at the architecture of each embedded unit.

†A more advanced version of the chip provides sixteen microengines.

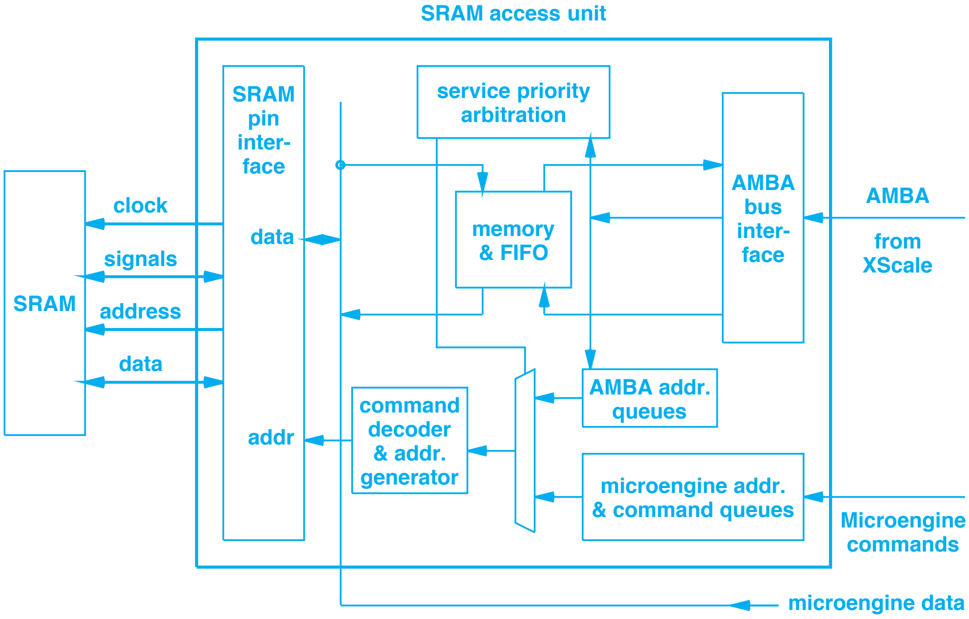


Figure 22.7 The internal structure of the SRAM access unit that remains hidden in Figure 22.6. Each successive level in the architectural hierarchy reveals further details and structure.

EXERCISES

22.1 If an engineer is offered a job as a system architect, what will the job entail?

22.2 What is the motivation for a computer that offers two buses?

22.3 A computer with a USB port contains hardware known as a *USB hub* that usually connects the external ports to a PCI bus. Modify the diagram in Figure 22.2 to show a USB hub.

22.4 If a computer contains two buses connected by a *transparent bridge*, and the memory connects to one bus while the devices connect to the other, will the devices be able to communicate with memory? Explain.

22.5 What is the purpose of a controller chip in modern bus architectures?

22.6 A computer has one device that uses an old bus, but does not have the normal sockets or wires for the bus. How is such a situation possible?

22.7 In a PC, would a super high-speed video system connect to the Northbridge chip or the Southbridge chip? Explain.

22.8 If it takes 40 seconds to transfer a video over a USB 3.0 port, approximately how long will it take to transfer the same video over a Wi-Fi network that operates at 20 megabits per second?

22.9 A network processor, such as the one shown in Figure 22.6, is classified as a System on Chip (SoC). Explain why.

22.10 In many hardware design documents, rectangular boxes are used to represent a subsystem. Can one tell by looking at the diagram approximately how many gates will be needed to implement the function the box represents? Explain.

Chapter Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

23

Hardware Modularity

23.1 Introduction

Earlier chapters give an overview of hardware architectures without discussing design or implementation details. This brief chapter considers designs that employ modularity. In particular, the chapter contrasts hardware modularity with software modularity, and considers why common programming abstractions do not apply to hardware. It then uses an example to illustrate how a basic hardware module can be designed that is flexible, and how replication of a basic module allows a designer to form a scalable hardware design.

23.2 Motivations For Modularity

Modular construction has two motivations: intellectual and economic. From an intellectual perspective, a modular approach allows a designer to break a large complex problem into smaller pieces. A small piece is easier to understand than the complete solution. Consequently, it is easier for a designer to ensure that the piece is correct, and easier for a designer to optimize an individual piece.

The economic motivation for modularity arises from the cost of designing and testing products. In many cases, a company does not produce one isolated product. Instead, the company creates a set of related products. One common reason for multiple products arises from size — a company might sell a set of related products that range in size from small to large. For example, a company that sells network equipment might offer four models of a network switch, where the models connect four computers, twenty-four computers, forty-eight computers, or ninety-six computers. Alternatively, a company may offer a series of products that supply the same basic functionality, but where each product has special features. For example, a company that sells network equipment may offer one model that connects to a wireless Wi-Fi network and another model that connects to a wired Ethernet.

Because designing a product is expensive, a company can save money if a basic module can be designed once and then re-used in multiple products. Further savings arise because once a basic module has been tested thoroughly, successive designs that use the module can assume it works correctly.

23.3 Software Modularity

Modularity has played a key role in software design since early computers. The principal abstraction consists of a *subroutine* (also called a *procedure*, *subprogram* or *function*). The early motivation for using subroutines arose from limited memory size — instead of repeating sections of code at multiple places throughout the program, a single copy of the code could be placed in memory, and then used (i.e., *called*) at several places in the program.

As software became more complex, subprograms became an important tool for handling complexity. In particular, the use of a subprogram abstraction made it possible to have an expert build a piece of software that other programmers could use without understanding the details. For example, an expert who understands numerical mathematics can create a set of trigonometric functions, such as *sin(x)* and *cos(x)*, that are both efficient and accurate. Other programmers can invoke the functions without writing the code themselves and without needing to understand the algorithms being used. By raising the level of abstraction and hiding details, subprograms allow programmers to work at a higher level, meaning that they can be much more productive and the resulting software will contain fewer errors.

23.4 Parameterized Invocation Of Subprograms

How can a basic building block be used for multiple purposes? The answer for software is well known. When creating a subprogram, a programmer specifies a set of *formal parameters*. Then, when writing code that invokes the subprogram, the programmer specifies actual arguments that are substituted in place of formal parameters. The key point is:

> *When building modularized software, a single copy of each subprogram exists. The only change among invocations consists of the actual arguments supplied when the subprogram is invoked.*

23.5 Hardware Scaling And Parallelism

Although it works well with software, the paradigm of parameterized function invocation cannot be used with hardware. The reason is that software can invoke a function iteratively, but hardware requires separate physical instantiations that can be controlled in parallel. For example, consider controlling a set of *N* items. In software, the items can be stored in an array, a function can be written to perform an operation on one item, and the program can iterate through the array, calling the function for each element. The program can scale to a larger array merely by changing the bound on the iteration.

When hardware is created to control a set of items, each item requires some hardware dedicated to the item. If additional elements are added to the set, additional hardware must be added to the design. In other words, scaling a hardware design always requires adding additional pieces of hardware. As a consequence:

> *When hardware designers think about a modular design, they look for ways to make it possible to add additional hardware to the design, not for ways to invoke a given piece of hardware iteratively.*

23.6 Basic Block Replication

The fundamental technique used to make it possible to scale hardware consists of defining a basic building block that can be replicated as needed. We have already seen trivial examples. For instance, a latch circuit can be replicated *N* times to form an *N*-bit register, and a *full adder* is replicated *N–1* times and combined with a *half adder* to build a circuit to compute the sum of two *N*-bit integers.

In the trivial cases described above, replication involves a small circuit (i.e., a few gates), and the number of replications is fixed. Although replication of a small circuit is an important aspect of design, the approach can be applied to significantly larger circuits and used to scale a design. For example, a chip manufacturer may use a multicore architecture to produce a series of products that have two cores, four cores, eight cores, and so on. Replication is especially important in designs where the number of inputs or outputs visible to a user varies across a series of products.

23.7 An Example Design (Rebooter)

An example will clarify the idea. Rather than choose a hypothetical design, we will consider a piece of hardware used in the author's lab. The lab, which is used for operating system and networking research, has a large set of *backend* computers that are available for researchers and students in classes. The lab facilities allow a user to create an operating system, allocate a backend computer, download the operating system into the backend computer's memory, and start the computer running. The user then can interact with the backend computer.

Unfortunately, experimental work on operating systems often results in crashes or leaves the computer hardware in a state that cannot respond to further input. In such situations, the backend computer must be power-cycled to regain control. Therefore, we have created a special-purpose hardware system that can power-cycle individual backend computers as needed. We call the system a *rebooter*. Several generations of rebooter hardware have been used in the lab; we will review one design.

23.8 High-level Rebooter Design

In principle, the rebooter hardware follows a straightforward approach. A rebooter has a set of outputs that each supply power to a backend computer. The inputs to the rebooter consist of a binary value that specifies one of the outputs to reboot plus an *enable input* that tells the rebooter to act. To use the rebooter, a binary value is placed on the input lines (to specify one of the outputs) and the enable input is set to 1, which causes the rebooter to power-cycle the specified output†. Figure 23.1 illustrates the inputs and outputs.

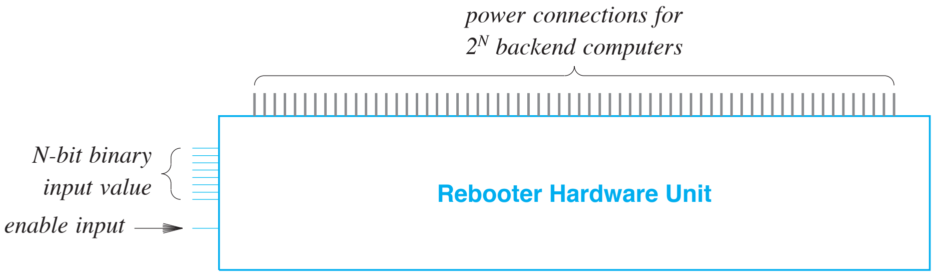


Figure 23.1 The conceptual organization of rebooter hardware.

How many outputs should a rebooter have? The question is important because the rebooter needs a physical connection for each output. Initially, the lab had only one backend, but the size evolved quickly to two and then eight. To plan for the future, we needed a rebooter circuit to accommodate at least 40 backends, and perhaps 100. The situation illustrates a standard hardware dilemma:

- A design with too few outputs will not accommodate future needs
- A design with too many outputs is wasteful

†The exact details of how the rebooter circuit is used are irrelevant to the discussion that follows; it is only important to understand the basics.

23.9 A Building Block To Accommodate A Range Of Sizes

Rather than choose a specific size, we used a modular approach. That is, we chose a basic building block and devised a way to interconnect basic blocks to form a larger rebooter. The modular approach allowed us to construct a small rebooter, and then add additional outputs as needed.

Our basic building block consists of a sixteen-output rebooter as Figure 23.2 illustrates.

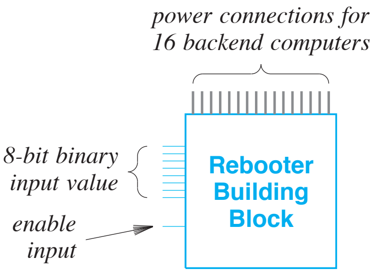


Figure 23.2 Illustration of the basic building block used for the rebooter.

Look carefully at the figure. The binary input value comprises eight bits, but there are only sixteen outputs. Thus, only four bits are needed to select one of the outputs. Why are extra input bits present? They are used to allow multiple copies of the building block to be combined to form a larger rebooter.

23.10 Parallel Interconnection

Our design uses a parallel approach common to many hardware systems. That is, the inputs connect to all modules in parallel. Conceptually, each building block passes a copy of its inputs (including the enable input) on to the next building block. Figure 23.3 illustrates the idea.

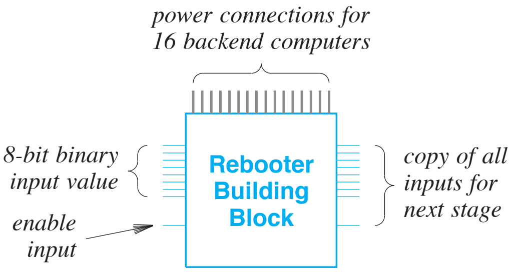


Figure 23.3 Illustration of a basic building block passing all inputs to the next stage of the rebooter.

23.11 An Example Interconnection

Figure 23.4 illustrates how the building blocks can be connected.

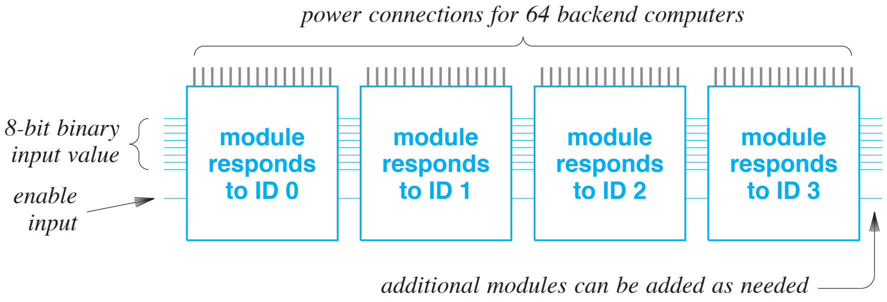


Figure 23.4 An example interconnection of four copies of the basic building block that provides 64 outputs.

23.12 Module Selection

As Figure 23.4 indicates, the inputs are passed in parallel to all four modules. A question arises: if the input specifies power-cycling computer number 5, does each module power-cycle its fifth output? The answer is no. Only the fifth output on module 1 is affected.

To understand how modules respond to inputs, it is necessary to know that each module is assigned a unique ID (0, 1, 2, and 3 in our example). A module includes hardware that checks the four high-order bits of the input to see if they match the assigned ID. If the input does not match the ID, the input is ignored. In other words, the hardware interprets the four high-order bits as a *module selection* and the four low-order bits as an *output selection*.

As an example, Figure 23.5 illustrates how the hardware interprets the input value 5 as module 0 and output 5.

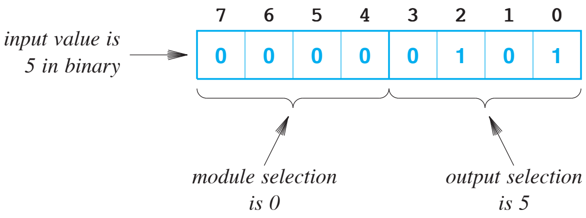


Figure 23.5 The interpretation of input 5 by the rebooter in Figure 23.4.

As Figure 23.5 shows, input 5 means the four high-order bits contain 0000 and the four low-order bits contain 0101. The high order bits match the ID assigned to module 0, but none of the other modules. Therefore, only module 0 responds to the input.

Using the high-order bits of the input to select a module makes the hardware extremely efficient. The module selection bits can be passed to a *comparator chip* along with the ID of the module. As the name implies, a comparator compares two sets of inputs, and sets an output line high if the two are equal. Thus, very little additional hardware is needed to perform module selection.

23.13 Summary

Both hardware and software engineers use modularity. In software, the fundamental abstraction for modularity is a subprogram. In hardware, the fundamental abstraction is the replication of a basic building block.

One method used to accommodate a range of hardware sizes consists of structuring a module (i.e., a building block) to accept a set of N input lines that control a set of 2^N outputs. When building blocks are replicated, each is assigned a unique ID. Additional input lines are added to the design, which means the high-order bits of the input can be used to select one of the modules, and the low-order bits can be used to select an output on the module.

EXERCISES

23.1 In enginnering, what is the relationship between *modularity* and *re-use*?

23.2 How does the ability to pass arguments to functions help programmers control the complexity of software?

23.3 When a software engineer and a hardware engineer think about the design of a crypto system that processes 128-bit integers, they each start with a bias. A software engineer might imagine an algorithm that iterates through the integer, working on 32 bits at a time. What will a hardware engineer envision?

23.4 Mathematically, one can have an arbitrary number of outputs from a module and use arithmetic to extract a module number and an input for the module (e.g., for seven outputs per module divide the input value by 7 to get a module number and use the remainder to select an ouput within the module). However, hardware engineers always choose to make outputs a power of two. Explain.

23.5 What are the tradeoffs to consider when choosing how many outputs a piece of hardware should have?

23.6 Suppose a basic building block contains 4 outputs, and a design must scale to 64 outputs. How many building blocks will be used?

23.7 If each building block contains 8 outputs and the input has 16 bits, how many total outputs can be controlled, and how many building block chips will be used?

23.8 In the previous exercise, draw a diagram similar to the one in Figure 23.5 that shows how bits of the input are interpreted.

23.9 Look up comparator chips. How many pairs of inputs does a single comparator have?

23.10 In the previous exercise, suppose a comparator chip can compare K pairs of inputs and a designer needs to compare $2K$ pairs. How can multiple chips be used?

Appendix 1

Lab Exercises For A Computer Architecture Course

A1.1 Introduction

This appendix presents a set of lab exercises for an undergraduate computer architecture course. The labs are designed for students whose primary educational goal is learning how to build software, not hardware. Consequently, after a few weeks of introduction to digital circuits, the labs shift emphasis to programming.

The facilities required for the lab are minimal: a small amount of hardware is needed for the early weeks, and access to computers running a version of the Unix operating system (e.g., Linux) is needed for later labs. A RISC architecture works best for the assembly language labs because instructors find that CISC architectures absorb arbitrary amounts of class time on assembly language details.

One lab asks students to write a C program that detects whether an architecture is big endian or little endian. Few additional resources are needed because most of the coding and debugging can be performed on one of the two architectures, with only a trivial amount of time required to port and test the program on the other.

A1.2 Hardware Required for Digital Logic Experiments

The hardware labs covered in the first few weeks require each student to have the following:

- Solderless breadboard
- Wiring kit used with breadboard (22-gauge wire)
- Five-volt power supply
- Light-Emitting Diode (used to measure output)
- NAND and NOR logic gates

None of the hardware is expensive. To handle a class of 70 students, for example, Purdue University spent less than $1000 on hardware. Smaller classes or sharing in the lab can reduce the cost further. As an alternative, it is possible to institute a lab fee or require students to purchase their own copy of the hardware.

A1.3 Solderless Breadboard

A *solderless breadboard* is used to rapidly construct an electronic circuit without requiring connections to be soldered. Physically, a breadboard consists of a block of plastic (typically three inches by seven inches) with an array of small holes covering the surface.

The holes are arranged in rows with a small gap running down the center and extra holes around the outside. Each hole on the breadboard is a socket that is just large enough for a copper wire — when a wire is inserted in the hole, metal contacts in the socket make electrical contact with the metal wire. The size and spacing of the sockets on a breadboard are arranged to match the size and spacing of pins on a standard integrated circuit (technically an IC that uses a standard DIP package), and the gap on the breadboard matches the spacing across the pins on an IC, which means that one or more integrated circuits can be plugged into the breadboard. That is, the pins on an IC plug directly into the holes in the breadboard.

The back of a breadboard contains metal strips that interconnect various sockets. For example, the sockets on each side of the center in a given row are interconnected. Figure A1.1 illustrates sockets on a breadboard and the electrical connections among the sockets.

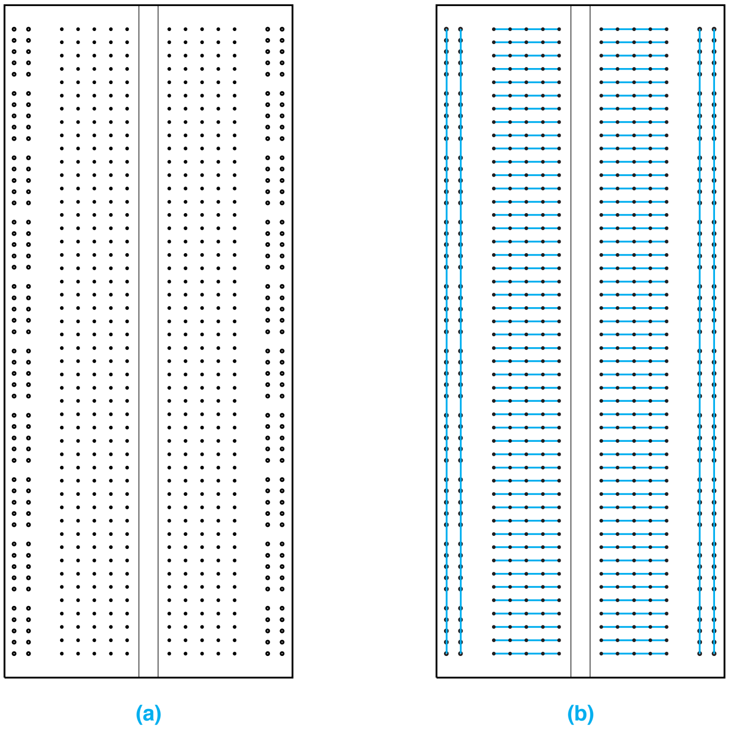


Figure A1.1 (a) Illustration of a breadboard with sockets into which wires can be inserted, and (b) blue lines showing the electrical connections among the sockets.

A1.4 Using A Solderless Breadboard

To use a breadboard, an experimenter plugs integrated circuits onto the breadboard along the center, and then uses short wires to make connections among the ICs. A wire plugged into a hole in a row connects to the corresponding pin on the IC that is plugged into the row. To make the connections, an experimenter uses a set of pre-cut wires known as a *wiring kit*. Each individual wire in the wiring kit has bare ends that plug into the breadboard, but is otherwise insulated. Thus, many wires can be added to a breadboard because the insulated area on a wire can rub against the insulation of other wires without making electrical contact.

Figure A1.2 illustrates part of a breadboard that contains a 7400 IC, with wires connecting some of the gates on the IC.

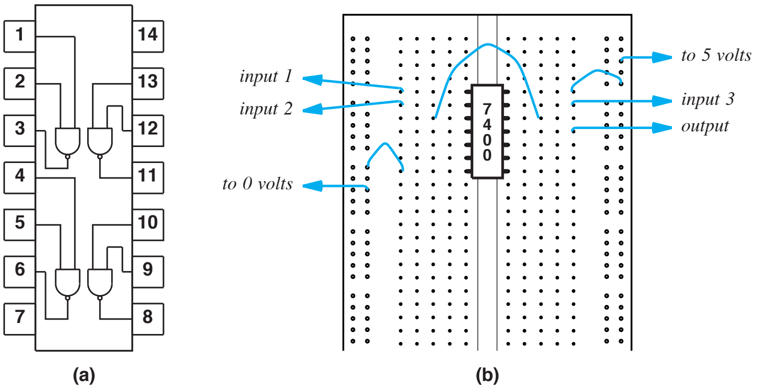


Figure A1.2 Illustrations of (a) the internal connections on a 7400 chip, and (b) part of a breadboard with blue lines indicating wires connecting a 7400 chip. Using a set of sockets to connect power and ground wires allows additional connections to be added.

A1.5 Power And Ground Connections

When multiple chips are plugged into a breadboard, each chip must have connections to *power* and *ground* (i.e., five volts and zero volts). To ensure that the power and ground connections are convenient and to keep the wires short, most experimenters choose to devote the outer sets of sockets on both sides of the breadboard to power and ground.

The wires used to connect power and ground are semi-permanent in the sense that they can be re-used for many experiments. Thus, experimenters often use the color of a wire to indicate its purpose, and choose colors for power and ground that are not used for other connections. For example, red wires can be used for all power connections, black wires can be used for all ground connections, and blue wires can be used for other connections. Of course, the wires themselves do not differ — the color of the insulation merely helps a human understand the purpose of the wire. When disassembling a breadboard after an experiment is finished, the experimenter can leave the power and ground connections for a later experiment.

A1.6 Building And Testing Circuits

The easiest approach to building a digital circuit consists of constructing the circuit in stages and testing each stage of the circuit as building proceeds. For example, after connecting power and ground to a chip, a gate on the chip can be tested to verify that the chip is working as expected. Similarly, after a particular gate has been connected, the input(s) and output(s) of the gate can be measured to determine whether the connections are working.

Although it is possible to use a voltmeter to measure the output of a digital circuit, most experimenters prefer an easy and inexpensive alternative: a *Light Emitting Diode* (*LED*). The idea is to choose an LED that can be powered directly†. The LED glows when it is connected to logical one (i.e., five volts), and is off when its input wire connects to logical zero (i.e., zero volts). For example, to test the circuit in Figure A1.2, an LED can be connected to the output (pin 11 of the integrated circuit).

A1.7 Lab Exercises

The next pages contain a series of lab exercises. Although each writeup specifies the steps to be taken in lab, additional details that pertain to the local environment or computer system must be supplied by the lab instructor. For example, the first lab asks students to establish their computer account, including environment variables. Because the set of directories to be included on the path depend on the local computer system, the set of actual paths must be supplied for each environment.

†Warning: the LED must have electrical characteristics that are appropriate for the circuit — an arbitrary LED can draw so much electrical power that it will cause a 7400-series integrated circuit to burn out.

Lab 1

Introduction And Account Configuration

Purpose

To learn about the lab facilities and set up a computer account for use in the lab during the semester.

Background Reading And Preparation

Read about the *bash shell* available with Linux, and find out how to set Linux environment variables.

Overview

Modify your lab account so your environment will be set automatically when you log in.

Procedure And Details (checkmark as each is completed)

______ 1. Modify your account startup file (e.g., *.profile* or *.bash_profile*) so your PATH includes directories as specified by your lab instructor.

______ 2. Log out and log in again.

______ 3. Verify that you can reach the files and compilers that your lab instructor specifies.

Lab 2

Digital Logic: Use Of A Breadboard

Purpose

To learn how to wire a basic breadboard and use an LED to test the operation of a gate.

Background Reading And Preparation

Read Chapter 2 to learn about basic logic gates and circuits, and read the beginning sections of this Appendix to learn about breadboards. Attend a lecture on how to properly use the breadboard and related equipment.

Overview

Place a 7400 chip on a breadboard, connect power and ground from a five-volt power supply, connect the inputs of a gate to the four possible combinations of zero and one, and use an LED to observe the output.

Procedure And Details (checkmark as each is completed)

______ 1. Obtain a breadboard, power supply, wiring kit, and parts box with the necessary logic gates. Also verify that you have a data sheet that specifies the pins for a 7400, which is a quad, two-input NAND gate. A copy of the pin diagram can also be found in Figure 2.13 of the text, which can be found on page 22.

______ 2. Place the 7400 on the breadboard as shown in Figure A1.2b on page 448.

______ 3. Connect the two wires from a five-volt power supply to two separate sets of sockets near the edge of the board.

______ 4. Add a wire jumper that connects pin 14 on the 7400 to five volts.

______ 5. Add a wire jumper that connects pin 7 on the 7400 to zero volts. NOTE: be sure not to reverse the connections to the power supply or the chip will be damaged.

______ 6. Add a wire jumper that connects pin 1 on the 7400 to zero volts.

______ 7. Add a wire jumper that connects pin 2 on the 7400 to zero volts.

______ 8. Connect the LED, from the lab kit, between pin 3 on the 7400 and ground (zero volts). NOTE: the LED must be connected with the positive lead attached to the 7400.

______ 9. Verify that the LED is lit (it should be lit because both inputs are zero which means the output should be one).

_____ 10. Move the jumper that connects pin 2 from zero volts to five volts, and verify that the LED remains lit.

_____ 11. Move the jumper that connects pin 2 back to zero volts, move the jumper that connects pin 1 from zero volts to five volts, and verify that the LED remains lit.

_____ 12. Keep the jumper from pin 1 on five volts, move the jumper that connects pin 2 to five volts, and verify that the LED goes out.

Optional Extensions (checkmark as each is completed)

_____ 13. Wire the breadboard as shown in Figure A1.2b (pin 3 connected to pin 12, and pin 13 acting as an additional input).

_____ 14. Connect the LED between pin 11 and ground.

_____ 15. Record the LED values for all possible combinations of the three inputs.

_____ 16. What Boolean function does the circuit represent?

Lab 3

Digital Logic: Building An Adder From Gates

Purpose

To learn how basic logic gates can be combined to perform complex tasks such as binary addition.

Background Reading And Preparation

Read Chapter 2 about basic logic gates and circuits, and read the beginning sections of this Appendix to learn about breadboards.

Overview

Build a half adder and full adder circuit using only basic logic gates. Combine the circuits to implement a two-bit binary adder with carry output.

Procedure And Details (checkmark as each is completed)

______ 1. Obtain a breadboard, power supply, wiring kit, and parts box with the necessary logic gates as well as lab writeups that describe both the chip pinouts and the logic diagram of an adder circuit.

______ 2. Construct a binary half adder as specified in the logic diagram that your lab instructor provides.

______ 3. Connect the outputs to LEDs, the inputs to switches, and verify that the results displayed on the LED are the correct values for a one-bit adder.

______ 4. Construct a binary full adder as specified in the logic diagram that your lab instructor provides.

______ 5. Connect the outputs to LEDs, the inputs to switches, and verify that the results displayed on the LED are the correct values for a full adder.

______ 6. Chain the half adder circuit to the full adder circuit to make a two-bit adder. Verify that the circuit correctly adds a pair of two-bit numbers and the carry out value is correct.

Optional Extensions (checkmark as each is completed)

______ 7. Draw the logic diagram for a three-bit adder.

______ 8. Draw the logic diagram for a four-bit adder.

______ 9. Give a formula for the number of gates required to implement an n-bit adder.

Notes

Lab 4

Digital Logic: Clocks And Decoding

Purpose

To understand how a clock controls a circuit and allows a series of events to occur.

Background Reading And Preparation

Read Chapter 2 to learn about basic logic gates and clocks. Concentrate on understanding how a clock functions.

Overview

Use a switch to simulate a clock, and arrange for the clock to operate a decoder circuit (informally called a *demultiplexor circuit*).

Procedure And Details (checkmark as each is completed)

______ 1. Obtain a breadboard, power supply, wiring kit, and parts box with the necessary logic gates as well as lab writeups that describe both the chip pinouts and the logic diagram of a decoder.

______ 2. Use a switch to simulate a slow clock.

______ 3. To verify that the switch is working, connect the output of the switch to an LED, and verify that the LED goes on and off as the switch is moved back and forth.

______ 4. Connect the simulated clock to the input of a four-bit binary counter (a 7493 chip).

______ 5. Use an LED to verify that each time the switch is moved through one cycle, the outputs of the counter move to the next binary value (modulo four).

______ 6. Connect the four outputs from the binary counter to the inputs of a decoder chip (a 74154).

______ 7. Use an LED to verify that as the switch moves through one cycle, exactly one output of the decoder becomes active. Warning: the 74154 is counterintuitive because the active output is low (logical zero) and all other outputs are high (logical one).

Optional Extensions (checkmark as each is completed)

______ 8. Use a 555 timer chip to construct a 1-Hz clock, and verify that the clock is working.

______ 9. Replace the switch with the clock circuit.

_____ 10. Use multiple LEDs to verify that the decoder continually cycles through each output.

Notes

Lab 5

Representation: Testing Big Endian Vs. Little Endian

Purpose

To learn how the integer representation used by the underlying hardware affects programming and data layout.

Background Reading And Preparation

Read Chapter 3 to learn about big endian and little endian integer representations and the size of an integer.

Overview

Write a C program that examines data stored in memory to determine whether a computer uses big endian or little endian integer representation.

Procedure And Details (checkmark as each is completed)

______ 1. Write a C program that creates an array of bytes in memory, fills the array with zero, and then stores integer 0x04030201 in the middle of the array.

______ 2. Examine the bytes in the array to determine whether the integer is stored in big endian or little endian order.

______ 3. Compile and run the program (without changes to the source code) on both a big endian and little endian computer, and verify that it correctly announces the integer type.

______ 4. Add code to the program to determine the integer size (hint: start with integer 1 and shift left until the value is zero).

______ 5. Compile and run the program (without changes to the source code) on both a thirty-two bit and a sixty-four bit computer, and verify the program correctly announces the integer size.

Optional Extensions (checkmark as each is completed)

______ 6. Find an alternate method of determining the integer size.

______ 7. Implement the alternate method to determine integer size, and verify that the program works correctly.

______ 8. Extend the program to announce the integer format (i.e., one’s complement or two’s complement).

Notes

Lab 6

Representation: A Hex Dump Function In C

Purpose

To learn how values in memory can be presented in hexadecimal form.

Background Reading And Preparation

Read Chapter 3 on data representation, and find both the integer and address sizes for the computer you use†. Ask the lab instructor for an exact specification for the output format.

Overview

Write a C function that produces a hexadecimal dump of memory in ASCII. The lab instructor will give details about the format for a particular computer, but the general form is as follows:

```
 Address         Words In Hexadecimal         ASCII characters
--------- -------- -------- -------- -------- ----------------
aaaaaaaa  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx cccccccccccccccc
```

In the example, each line corresponds to a set of memory locations. The string *aaaaaaaa* denotes the starting memory address (in hexadecimal) for values on the line, *xxxxxxxx* denotes the value of a word in memory (also in hexadecimal), and *cccccccccccccccc* denotes the same memory locations when interpreted as ASCII characters. Note: the ASCII output only displays printable characters; all other characters are displayed as blanks.

Procedure And Details (checkmark as each is completed)

______ 1. Create a function, *mdump* that takes two arguments that each specify an address in memory. The first argument specifies the address where the dump should start, and the second argument specifies the highest address that needs to be included in the dump. Test to ensure that the starting address is less than the ending address.

†On most computers, the address size equals the integer size.

______ 2. Modify each of the arguments so they specify an appropriate word address (i.e., an exact multiple of four bytes). For the starting address, round down to the nearest word address; for the ending address, round up.

______ 3. Test the function to verify that the addresses are rounded correctly.

______ 4. Add code that uses *printf* to produce headings for the hexadecimal dump, and verify that the headings are correct.

______ 5. Add code that iterates through the addresses and produces lines of hexadecimal values.

______ 6. To verify that function *mdump* outputs correct values, declare a *struct* in memory, place values in the fields, and invoke the *mdump* function to dump items in the struct.

______ 7. Add code that produces printable ASCII character values for each of the memory locations, as shown above.

______ 8. Verify that only printable characters are included in the output (i.e., verify that a non-printable character such as 0x01 is mapped into a blank).

Optional Extensions (checkmark as each is completed)

______ 9. Extend *mdump* to start and stop on a byte address (i.e., omit leading values on the first line of output and trailing values on the last line).

_____ 10. Modify *mdump* so that instead of printing bytes in ASCII, it displays the values of words in decimal.

_____ 11. Modify *mdump* so that instead of printing ASCII values, the function assumes the memory corresponds to machine instructions and gives mnemonic opcodes for each instruction. For example, if the first word on the line corresponds to a *load* instruction, print *load*.

_____ 12. Add an argument to function *mdump* that selects from among the various forms of output (ASCII characters, decimal, or instructions).

Lab 7

Processors: Learn A RISC Assembly Language

Purpose

To gain first-hand experience with an assembly language and understand the one-to-one mapping between assembly language instructions and machine instructions.

Background Reading And Preparation

Read Chapters 5, 7, and 9 to learn the concepts of instruction sets and operand types. Read about the specific instruction set available on your local computer. Consult the assembler reference manual to learn the syntax conventions needed for the assembler. Also read the assembler reference manual to determine the conventions used to call an external function.

Overview

Write an assembly language program that shifts an integer value to the right and then calls a C function to display the resulting value in hexadecimal.

Procedure And Details (checkmark as each is completed)

______ 1. Write a C function, *int_out*, that takes an integer argument and uses *printf* to display the argument value in hexadecimal.

______ 2. Test the function to ensure it works correctly.

______ 3. Write an assembly language program that places the integer 4 in a register and shifts the contents of the register right one bit.

______ 4. Extend the program to pass the result of the previous step as an argument to external function *int_out*.

______ 5. Verify that the program produces 0x2 as the output.

______ 6. Load integer 0xBD5A into a register and print the result to verify that sign extension works correctly.

______ 7. Instead of shifting the integer 4 right one bit, load 0xBD5B7DDE into a 32-bit register, shift right one bit, and verify that the output is correct.

Optional Extensions (checkmark as each is completed)

______ 8. Rewrite the external function *int_out* and the assembly language program to pass multiple arguments.

Notes

Lab 8

Processors: Function That Can Be Called From C

Purpose

To learn how to write an assembly language function that can be called from a C program.

Background Reading And Preparation

Read Chapter 9 to learn about the concept of subroutine calls in assembly languages, and read the C and assembler reference manuals to determine the conventions that C uses to call a function on your local computer.

Overview

Write an assembly language function that can be called from a C program to perform the *exclusive or* of two integer values.

Procedure And Details (checkmark as each is completed)

______ 1. Write a C function *xor* that takes two integer arguments and returns the exclusive-or of the arguments.

______ 2. Write a C main program that calls the *xor* function with two integer arguments and displays the result of the function.

______ 3. Write *axor*, an assembly language version of the C *xor* function that behaves exactly like the C version. (Do not merely ask the C compiler to generate an assembly file; write the new version from scratch.)

______ 4. Add a *printf* call to the *axor* function and use it to verify that the function correctly receives the two values that the C program passes as arguments (i.e., argument passing works correctly).

______ 5. Arrange for the C main program to test *axor* to verify that the code returns correct results for a reasonable range of inputs. Hint: generate values randomly.

Optional Extensions (checkmark as each is completed)

______ 6. Modify the C program and the *axor* function so that the C program passes a single structure as an argument instead of two integers. Arrange for the structure to contain two integer values.

Notes

Lab 9

Memory: Row-Major And Column-Major Array Storage

Purpose

To understand storage of arrays in memory and row-major order and column-major order.

Background Reading And Preparation

Read Chapters 10 through 13 to learn about basic memory organization and the difference between storing arrays in row-major order and column-major order.

Overview

Instead of using built-in language facilities to declare two-dimensional arrays, implement two C functions, *two_d_store* and *two_d_fetch*, that use linear storage to implement a two-dimensional array. Function *two_d_fetch* takes six arguments: the base address in memory of a region to be used as a two-dimensional array, the size (in bytes) of a single entry in the array, two array dimensions, and two index values. For example, instead of the two lines:

```
int  d[10,20];
x = d[4,0];
```

a programmer can code:

```
char  d[200*sizeof(int)];
x = two_d_fetch(d, sizeof(int), 10, 20, 4, 0);
```

Function *two_d_store* has seven arguments. The first six correspond to the six arguments of *two_d_fetch*, and the seventh is a value to be stored. For example, instead of:

```
int  d[10,20];
d[4,0] = 576;
```

a programmer can code:

```
char  d[200*sizeof(int)];
two_d_store(d, sizeof(int), 10, 20, 4, 0, 576);
```

Procedure And Details (checkmark as each is completed)

______ 1. Implement function *two_d_store*, using row-major order to store the array.

______ 2. Create an area of memory large enough to hold an array, initialize the entire area to zero, and then call *two_d_store* to store specific values in various locations. Use the hex dump program created in Lab 6 to display the result, and verify that the correct values have been stored.

______ 3. Implement function *two_d_fetch*, using row-major order to match the order used by *two_d_store*.

______ 4. Verify that your implementations of *two_d_store* and *two_d_fetch* work correctly.

______ 5. Test *two_d_store* and *two_d_fetch* for boundary conditions, such as the minimum and maximum array dimensions.

______ 6. Rewrite *two_d_store* and *two_d_fetch* to use column-major order.

______ 7. Verify that the code for column-major order works correctly.

Optional Extensions (checkmark as each is completed)

______ 8. Verify that functions *two_d_store* and *two_d_fetch* work correctly for an array that stores: characters, integers, or double-precision items.

______ 9. Extend *two_d_store* and *two_d_fetch* to work correctly with any range of array index. For example, allow the first index to range from -5 to +15, and allow the second index to range from 30 to 40.

Lab 10

Input/Output: A Buffered I/O Library

Purpose

To learn how buffered I/O operates and to compare the performance of buffered and unbuffered I/O.

Background Reading And Preparation

Read Chapters 14 through 16 to learn about I/O in general, and read Chapter 17 to learn about buffering.

Overview

Build three C functions, *buf_in*, *buf_out*, and *buf_flush* that implement buffered I/O. On each call, function *buf_in* delivers the next byte of data from file descriptor zero. When additional input is needed from the device, *buf_in* reads sixteen kilobytes of data into a buffer, and allows successive calls to return values from the buffer. On each call, function *buf_out* writes one byte of data to a buffer. When the buffer is full or when the program invokes function *buf_flush*, data from the buffer is written to file descriptor one.

Procedure And Details (checkmark as each is completed)

______ 1. Implement function *buf_in*.

______ 2. Verify that *buf_in* operates correctly for input of less than sixteen kilobytes (i.e., less than one buffer of data).

______ 3. Redirect input to a file that is larger than thirty-two kilobytes, and verify that *buf_in* operates correctly for input that requires *buf_in* to fill a buffer multiple times.

______ 4. Implement functions *buf_out* and *buf_flush*.

______ 5. Verify that *buf_out* and *buf_flush* operate correctly for output of less than one buffer (i.e., less than sixteen kilobytes).

______ 6. Verify that *buf_out* and *buf_flush* operate correctly for output that spans multiple buffers.

Optional Extensions (checkmark as each is completed)

______ 7. Compare the performance of functions *buf_in*, *buf_out*, and *buf_flush* to the performance of unbuffered I/O (i.e., *read* and *write* of one byte) for various size files. Plot the results.

______ 8. Measure the performance of *buf_in*, *buf_out*, and *buf_flush* for various size buffers when copying a large file. Use buffers that range in size from 4 bytes to 100 Kbytes, and plot the results.

Notes

Lab 11

A Hex Dump Program In Assembly Language

Purpose

To gain experience coding assembly language.

Background Reading And Preparation

Review Chapters 5, 7, and 9, and assembly language programs written in earlier labs.

Overview

Rewrite the hex dump program from Lab 6 in assembly language.

Procedure And Details (checkmark as each is completed)

______ 1. Rewrite the basic hex dump function from Lab 6 in assembly language.

______ 2. Verify that the assembly language version gives the same output as the C version.

Optional Extensions (checkmark as each is completed)

______ 3. Extend the assembly language dump function to start and stop on a byte address (i.e., omit leading values on the first line of output and trailing values on the last line).

______ 4. Change the function to print values in decimal instead of ASCII character form.

______ 5. Modify the dump function so instead of printing ASCII values, the function assumes the memory corresponds to machine instructions and gives mnemonic opcodes for each instruction. For example, if the first word on the line corresponds to a *load* instruction, print *load*.

______ 6. Add an argument to the dump function that selects from among the various forms of output (ASCII characters, decimal, or instructions)

Notes

Appendix 2

Rules For Boolean Algebra Simplification

A2.1 Introduction

Boolean expressions can be simplified by applying the rules of Boolean algebra. Specifically, there are rules that cover associative, reflexive, and distributive properties. From an engineering perspective, the motivation for simplification is that an implementation requires fewer gates. For example, consider a *logical or*. We know that if either of the two expressions is *true*, the *logical or* will also be *true*. Thus, the expression *X or true* can be replaced by *true*.

A2.2 Notation Used

In the table that follows, a dot (·) denotes *logical and*, a plus sign (+) denotes *logical or*, an apostrophe (’) denotes *logical not*, *0* denotes *false*, and *1* denotes *true*. Using the notation, the expression:

$$(X + Y) \cdot Z'$$

represents:

$$(X \textit{ or } Y) \textit{ and } (\textit{not } Z)$$

A2.3 Rules Of Boolean Algebra

Figure A2.1 lists nineteen rules of Boolean algebra. Although many of the initial rules may seem obvious, they are included for completeness.

$$
\begin{aligned}
x + 0 &= x \\
x + 1 &= 1 \\
x \cdot 0 &= 0 \\
x \cdot 1 &= x \\
x + x &= x \\
x + x' &= 1 \\
x \cdot x &= x \\
x \cdot x' &= 0 \\
(x')' &= x \\
x \cdot y &= y \cdot x \\
x + y &= y + x \\
x \cdot (y \cdot z) &= (x \cdot y) \cdot z \\
x + (y + z) &= (x + y) + z \\
x \cdot (y + z) &= (x \cdot y) + (x \cdot z) \\
x + (y \cdot z) &= (x + y) \cdot (x + z) \\
x \cdot (x + y) &= x \\
x + (x \cdot y) &= x \\
(x \cdot y)' &= x' + y' \\
(x + y)' &= x' \cdot y'
\end{aligned}
$$

Figure A2.1 Rules of Boolean algebra that can be used to simplify Boolean expressions.

Appendix 3

A Quick Introduction To x86 Assembly Language

A3.1 Introduction

Engineers use the term *x86* to refer to a series of processors that use an architecture created by Intel Corporation†. Each processor in the Intel series was more powerful than its predecessor. Over time, the design changed from a 16-bit architecture to a 32-bit architecture. During the transition, Intel enforced *backward compatibility* to guarantee that newer chips in the series could execute code written for earlier chips. Thus, the fundamentals remain the same.

The x86 has undergone another transition, this time from a 32-bit architecture to a 64-bit architecture; the change was led by AMD, an Intel competitor. Once again, backward compatibility is a key part of the transition. In this brief chapter, we will discuss the 32-bit version first, and then describe 64-bit extensions.

Because it follows a CISC approach, an x86 processor has a large, complex instruction set. In fact, the instruction set is huge — the vendor's manuals that document the instructions comprise nearly 3000 pages. An x86 can contain special instructions for high-speed graphics operations, trigonometric functions, and the large set of instructions an operating system uses to control processor modes, set protection, and handle I/O. In addition to the 32-bit instructions used by applications running on recent processors, an Intel x86 processor retains hardware that supports previous versions. Consequently, we cannot review the entire instruction set in a brief appendix. Instead, we provide an overview that introduces basics. Once a programmer masters a few fundamentals, learning new instructions is straightforward.

†The name arises because Intel assigned part numbers such as 8086, 80286, 80386, and 80486.

A3.2 The x86 General-Purpose Registers

As a result of extensions, the x86 architecture suffers from confusing and unexpected inconsistencies. For example, the architecture includes eight *general-purpose registers*, and inconsistencies are especially apparent in the way the general-purpose registers are named and referenced. In particular, the initial design used four general-purpose 16-bit registers, and the assembly language provided names for individual bytes of each register. When the registers were extended to thirty-two bits, each extended register was given a name, and the architecture mapped each of the original 16-bit registers onto the low-order sixteen bits of the corresponding extended register. Thus, the assembly language provides a set of names that allows a programmer to reference an entire 32-bit register, the low-order 16-bit region of the register, or individual bytes within the 16-bit region. Unfortunately, the names are confusing. Initially, registers were assigned specific purposes, and the names reflect the historical use. Figure A3.1 illustrates the eight general-purpose registers, lists their historical purpose, and gives names for the registers as well as each subpart†.

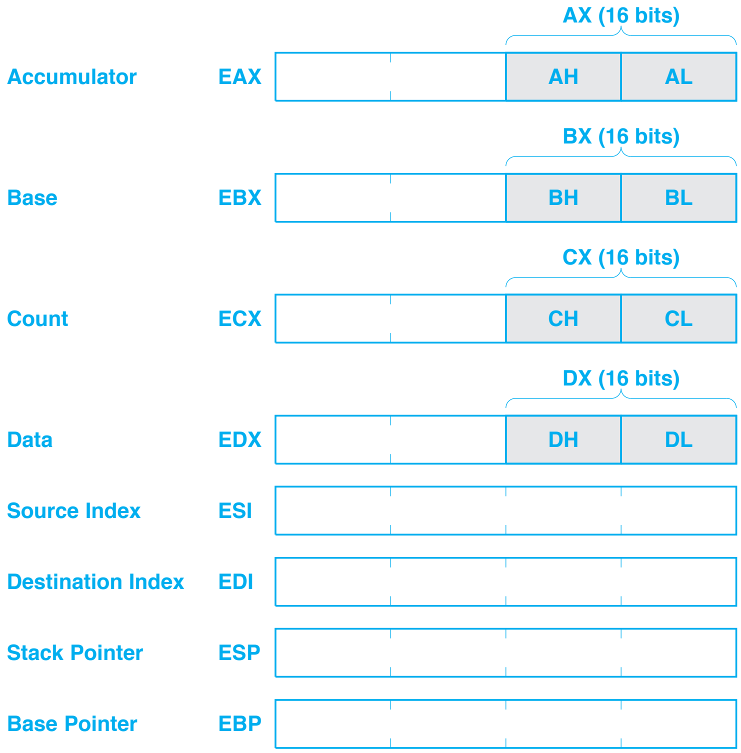


Figure A3.1 The eight general-purpose registers on an x86 processor, their historical purpose, and the names used to reference a register and the subparts.

†Because most assemblers do not distinguish between uppercase and lowercase, names *eax* and *EAX* refer to the same register. Programmers tend to use lowercase; documentation tends to use uppercase.

Although most of the registers are no longer restricted to their original purpose, the *stack pointer* (*ESP*) and *base pointer* (*EBP*) still have special meaning. The use of the base and stack pointers during a procedure call is explained below.

A3.3 Allowable Operands

Operands specify the values to be used in an operation and a location for the result. An operand can specify one of the registers, a location in memory, or a constant. Each instruction specifies the combinations that are allowed. For example, a *mov* instruction copies data from one location to another. *Mov* can copy a constant to a register or to memory, or can copy a data value from a register to memory, from memory to a register, or from one register to another. However, *mov* cannot copy data from one memory location directly to another. Thus, to copy data between two memory locations, a programmer must use two instructions. First, a programmer uses a *mov* to copy the data from memory to a register, and second, a programmer uses a *mov* to copy the data from the register to the new memory location.

Figure A3.2 lists the nomenclature used to describe the set of operands that are allowed for a given instruction.

Name	Meaning
<reg32>	Any 32-bit register, such as EAX, EBX, ...
<reg16>	Any 16-bit register, such as AX, BX, ...
<reg8>	Any 8-bit register, such as AH, AL, BH, BL...
<reg>	Any 32-bit, 16-bit, or 8-bit register
<con32>	Any 32-bit constant
<con16>	Any 16-bit constant
<con8>	Any 8-bit constant
<con>	Any 32-bit, 16-bit, or 8-bit constant
<mem>	Any memory address

Figure A3.2 Nomenclature used to specify allowable operands.

A later section explains how a memory address can be computed. For now, it is sufficient to understand that we will use the terminology from Figure A3.2 to explain instructions. As an example, consider the *mov* instruction, which copies a data item specified by a *target* operand into the location specified by a *source* operand. Figure A3.3 uses the nomenclature in Figure A3.2 to list the allowable operand combinations for *mov*.

Source Operand	Target Operand
<reg>	<reg>
<mem>	<reg>
<reg>	<mem>
<con>	<reg>
<con>	<mem>

Figure A3.3 Allowable operand combinations for a *mov* instruction.

A3.4 Intel And AT&T Forms Of x86 Assembly Language

Before we examine instructions, it is important to know a few assembly language basics. For example, assembly language employs a fixed statement format with one statement per line:

label *opcode* *operands*...

The *label* on a statement. which is optional, consists of a name used to identify the statement. If the statement defines a data item, the label specifies a name for the item; if a statement contains code, the label is followed by a colon, and can be used to pass control to the statement. The *opcode* field defines the type of data item or specifies an instruction; zero or more *operands* follow the opcode to give further details for the data or operation.

Unfortunately, many x86 assemblers have been created, and each has features that distinguish it from the others. Rather than examining each individual assembler, we will focus on two major categories. The first category employs a syntax that was originally defined by Intel and adopted by Microsoft; it is known informally as *Intel assembly language* or *Microsoft-Intel assembly language*. The second category employs a syntax originally defined by AT&T Bell Labs for Unix and adopted by the open source community for use in Linux; it is known as *AT&T assembly language* or *GNU assembly language* (*gas*).

Assemblers in either category are functionally equivalent in that they allow a programmer to code an arbitrary sequence of x86 instructions and to declare arbitrary data items in memory. Despite overall similarities, the two types of assemblers differ in many details. For example, the order in which operands are listed, the way registers are referenced, and the comment syntax differ. Although either type can be used, a programmer may find that one type is more intuitive, more convenient, or helps catch more programming errors. Because both types of assemblers are widely used in industry, we will examine examples for each.

A3.4.1 Characteristics Of Intel Assembly Language

An Intel assembler has the following characteristics:

- Operand order is right-to-left, with the source on the right and the target on the left
- Comments begin with a semicolon (;)
- Register names are coded without punctuation (e.g., eax)
- Immediate constants are written without punctuation
- The assembler deduces opcode type from the operand(s)

To remember the operand order, a programmer can think of an assignment statement in a high-level language: the expression is on the right and the value is assigned to the variable on the left. Thus, in Intel assembly language, a data movement operation is written:

```
mov    target, source
```

For example, the following code adds two to the contents of register EBX and places the result in register EAX:

```
mov    eax, ebx+2
```

The x86 hardware has *implicit* operand types, which means that at run-time, the opcode specifies the types of the operands. For example, instead of one *mov* instruction, the hardware contains an opcode for each possible operand type. That is, the x86 has an opcode to move a byte, another opcode to move a word, and so on. The hexadecimal values are 88, 89, 8A, 8B, 8C, ... When it produces binary code, an Intel assembler *deduces* the correct opcode from the operand types. If the target is a single byte, the assembler chooses the opcode that moves a byte, if the target is a sixteen-bit register, the assembler chooses the opcode that moves a sixteen-bit value, and so on. Each instruction follows the same pattern — although a programmer uses a single mnemonic in the program (e.g., *add* for addition or *sub* for subtraction), the processor has a set of opcodes for each operation, and the Intel assembler chooses the opcode that is appropriate for the operands the programmer specifies.

A3.4.2 Characteristics Of AT&T Assembly Language

An AT&T assembler has the following characteristics:

- Operand order is left-to-right, with the source on the left and the target on the right
- Comments are enclosed in /*... */ or begin with a hash mark (#)
- Register names are preceded by a percent sign (e.g., %eax)
- Immediate constants are preceded by a dollar sign ($)
- The programmer chooses a mnemonic that indicates the type as well as the operation

The operand order is the exact opposite of that used by an Intel assembler. Thus, in AT&T assembly language, a data movement operation is written:

```
mov    source, target
```

For example the following code adds two to the contents of register EBX and places the thirty-two-bit result in register EAX.

```
movl   %ebx+2, %eax
```

A3.5 Arithmetic Instructions

Addition And Subtraction. Many arithmetic and logical operations on the x86 take two arguments: a *source* and a *target*. The *target* specifies a location, such as a register, and the *source* specifies a location or a constant. The processor uses the two operands to perform the specified operation, and then places the result in the target operand. For example, the instruction:

```
Intel:        add    eax,ebx
AT&T:         add    %ebx,%eax
```

causes the processor to add the values in registers EAX and EBX, and then place the result in register EAX. In other words, the processor changes EAX by adding the value in EBX. Figure A3.4 lists the allowable combinations of operands for addition and subtraction.

Source Operand	Target Operand
<reg>	<reg>
<mem>	<reg>
<reg>	<mem>
<con>	<reg>
<con>	<mem>

Figure A3.4 The allowable combinations of operands for add or subtract.

Increment And Decrement. In addition to *add* and *sub*, the x86 offers increment and decrement instructions that add or subtract one. The instructions, which have opcodes *inc* and *dec* (followed by a designator on the AT&T assembler), each take a single argument that can be any register or any memory location. For example, the instruction:

```
Intel:        inc    ecx
AT&T:         incl   %ecx
```

increments the value of register ECX by one. A programmer must decide whether to use *inc* or *add*.

The inclusion of increment and decrement instructions in the instruction set illustrates an important principle about the architecture:

> *A CISC architecture, such as the one used with x86, often provides more than one instruction to perform a given computation.*

Multiplication And Division. Integer multiplication and division pose an interesting challenge for computer architects. The product that results when a pair of registers are multiplied can exceed a single register. In fact, the product can be twice as long as a register. Most computers also permit the dividend used in integer division to be larger than a single register.

The x86 includes many variations of integer multiplication and division. Some of the variations of multiplication allow a programmer to restrict the result to a specific size (e.g., restrict the product to thirty-two bits). To handle the case where the product will exceed one register, the x86 uses a combination of two registers to hold the result. For example, when multiplying two thirty-two-bit values, the x86 places the sixty-four-bit result in the EDX and EAX registers, with EDX holding the most significant thirty-two bits and EAX holding the least significant thirty-two bits.

The x86 also permits integer division to have a sixty-four-bit operand, stored in a pair of registers. Of course, integer division can also be used with small items. Even if the dividend does not occupy sixty-four bits, an x86 can use two registers to hold the result of an integer division: one holds the quotient and the other holds the remainder. Having a way to capture a remainder makes computations such as hashing efficient.

An x86 offers two basic forms of multiplication. The first form follows the same paradigm as addition or subtraction: the multiplication instruction has two arguments, and the result overwrites the value in the first argument. The second form takes three arguments, the third of which is a constant. The processor multiplies the second and third arguments, and places the result in the location specified by the first argument. For example,

```
Intel:          imul   eax,edi,42

AT&T:           imul   %edi,42,%eax
```

multiplies the contents of register EDI by 42, and places the result in register EAX.

A3.6 Logical Operations

The x86 processors offer *logical* operations that treat data items as a string of bits and operate on individual bits. Three of the logical operations perform bit-wise operations on two operands: logical *and*, *or*, and *xor*. The fourth logical operation, *not*, per-

Logical and, or, xor	
Source Operand	**Target Operand**
<reg>	<reg>
<mem>	<reg>
<reg>	<mem>
<con>	<reg>
<con>	<mem>

Logical not
Target Operand
<reg>
<mem>

Figure A3.5 The allowable combinations of operands for *and*, *or*, *xor*, and *not* instructions.

forms bit inversion on a single operand. Figure A3.5 lists the operand types used with logical operations.

In addition to bit-wise logical operations, an x86 supports bit *shifting*. Shifting can be applied to a register or memory location. In essence, a shift takes the current value, moves bits left or right by the amount specified, and places the result back in the register or memory location. When shifting, the x86 supplies zero bits to fill in when needed. For example, when shifting left by *K* bits, the hardware sets the low-order *K* bits of the result to zero, and when shifting right by *K* bits, the hardware sets the high-order *K* bits of the result to zero. Figure A3.6 lists the allowable operands used with left and right shift operations.

shift left (shl) and shift right (shr)	
Source Operand	**Target Operand**
<con8>	<reg>
<con8>	<mem>
<cl>	<reg>
<cl>	<mem>

Figure A3.6 Allowable operand combinations for shift instructions. The notation <*cl*> refers to the 8-bit register CL.

A3.7 Basic Data Types

Assembly language for an x86 allows a programmer to define initialized and uninitialized data items. Data declarations must be preceded by a *.data* assembler directive, which tells the assembler that the items are to be treated as data. A programmer can define each individual data item or can define a sequence of unnamed data items to occupy successive memory locations. Figure A3.7 lists the basic data types available; the figure assumes the AT&T assembler is set to produce code for a thirty-two-bit processor.

Intel Name	AT&T Name	Size In Bytes
DB (Data Byte)	.byte (single byte)	1
DW (Data Word)	.hword (half word)	2
DD (Data Double)	.long (long word)	4
DQ (Data Quad)	.quad (quad word)	8

Figure A3.7 Basic data types used by Intel and AT&T assemblers.

Each type of assembler permits a programmer to assign an initial value to data items. In Intel assembly program, a label starts in column 1, the data type appears next, and an initial value for the item follows the data type. An Intel assembler uses a question mark to indicate that the data item is uninitialized. In an AT&T assembly program, a label ends in a colon, and is followed by the type and an initial value; if the initialization is omitted, zero is assumed. Figures A3.8 and A3.9 illustrate declarations for the two types of assemblers.

```
        .DATA             ; start of data declarations  (Intel assembler)
z       DD       ?        ; four bytes that are uninitialized
y       DD       0        ; four bytes that are initialized to zero
x       DW       -54      ; two bytes initialized to -54
w       DW       ?        ; two bytes that are uninitialized
v       DB       ?        ; one byte that is uninitialized
u       DB       6        ; one byte initialized to 6
```

Figure A3.8 Examples of data declarations when using an Intel assembler.

An assembler places successive data items in adjacent bytes of memory. In the figures, the item named *u* is placed in the byte following the item named *v*. Similarly, *y* is placed just beyond *z*; because *z* is four bytes long, *y* starts four bytes beyond the location at which *z* starts.

```
        .data             ; start of data declarations  (AT&T assembler)
z:      .long             ; four bytes that are initialized to zero
y:      .long    0        ; four bytes that are initialized to zero
x:      .hword   -54      ; two bytes initialized to -54
w:      .hword            ; two bytes that are initialized to zero
v:      .byte             ; one byte that is initialized to zero
u:      .byte    6        ; one byte initialized to 6
```

Figure A3.9 Examples of data declarations when using an AT&T assembler.

A3.8 Data Blocks, Arrays, And Strings

Although it does not provide data aggregates, such as structs, x86 assembly languages do allow a programmer to declare multiple occurrences of a data item that occupy contiguous memory locations. For example, to declare three sixteen-bit items that are initialized to 1, 2, and 3, a programmer can write three separate lines that each declare one item or can list multiple items on a single line:

```
Intel:          q     DW      1, 2, 3

AT&T:           q:    .hword  1, 2, 3
```

The Intel assembler uses the modifier *K DUP(value)* to repeat a data value multiple times; the AT&T assembler uses *.space* to fill a specified size of memory with a value. For example, to declare one thousand repetitions of a data byte that is initialized to the numeric value 220, one writes:

```
Intel:          s     DB      1000 DUP(220)

AT&T:           s:    .space  1000, 220
```

The AT&T assembler provides a *.rept* macro to declare repetitions of larger items, such as a dozen occurrences of four-byte zero:

```
Intel:                DD      12 DUP(0)

AT&T:                 .rept   12
                      .long   0
                      .endr
```

In addition to numeric values, x86 assembly language allows a programmer to use ASCII characters as initial values. The Intel assembler encloses character constants in single quote marks, and allows multiple characters to be used to form a *string*. The assembler does not add a trailing zero (null termination). An AT&T assembler surrounds a string of characters with double quotes, and uses the directive *.ascii* or *.asciz* to declare a string; *.ascii* does not add a null termination byte, and *.asciz* does. For example, a programmer can declare a byte in memory that is initialized to the letter *Q* or a string that contains the characters *hello world*, with or without null termination.

```
Intel:          c     DB      ’Q’
                d     DB      ’hello world’
                e     DB      ’hello world’, 0

AT&T:           c:    .ascii  "Q"
                d:    .ascii  "hello world"
                e:    .asciz  "hello world"
```

A3.9 Memory References

As we have seen, many x86 instructions permit an operation to reference memory, either to fetch a value for use in the instruction or to store a result. The x86 hardware offers a complex mechanism that a programmer can use to compute a memory address: an address can be formed by adding the contents of two general-purpose registers plus a constant. Furthermore, one of the registers can be multiplied by two, four, or eight. A few examples will illustrate some of the possibilities.

Data Names. The most straightforward form of memory reference consists of a reference to a named data item. Intel assemblers use square brackets to enclose the name of a memory item, and AT&T assemblers precede the name by a dollar sign. In either case, the assembler computes the memory address assigned to the item, and substitutes the constant in the instruction. For example, if an assembly program contains a declaration for a 16-bit data item named *T*, the following instructions are used to move the 16-bit value from the memory into register DX:

```
Intel:                   mov     dx, [T]
AT&T:                    movw    $T, %dx
```

Indirection Through A Register. A programmer can compute a numeric value, place the value in a register, and then specify that the register should be used as a memory address. For example, the instruction:

```
Intel:                   mov     eax, [ebx]
AT&T:                    movl    (%ebx), %eax
```

uses the contents of register EBX as a memory address, and moves four bytes starting at that address into register EAX.

Expressions that compute an address are permitted, provided they adhere to the rule of adding at most two registers and a constant, with the option of multiplying one of the registers by two, four, or eight. For example, it is possible to form a memory address by adding the contents of EAX, the contents of ECX, and the constant 16, and then using the address to store the value of register EDI. In Intel notation, the operation is written:

```
mov     [eax+exb+16], edi
```

The rules for addresses can be difficult to master at first because they seem somewhat arbitrary. Figure A3.10 lists examples of valid and invalid memory references.

Valid references

```
mov   eax, [lab1]       ; move 4 bytes from label lab1 in memory to EAX
mov   [lab2], ebx       ; store 4 bytes from EBX to label lab2 in memory
and   eax, [esi-4]      ; and EAX with 4 bytes at address given by ESI - 4
not   [edi+8]           ; invert 32 bits at location given by EDI + 8
mov   [eax+2*ebx],0     ; store zero in 4 bytes at address given by EAX+2*EBX
mov   cl, [esi+4*ebx]   ; move byte from ESI+4*EBX into register CL
```

Invalid references

```
mov   eax, [esi-ebx]    ; cannot subtract two registers
mov   [eax+ebx+cl], 0   ; cannot specify more than two registers
```

Figure A3.10 Examples of valid and invalid memory references using Intel notation.

A3.10 Data Size Inference And Explicit Size Directives

Because a memory address can be calculated at run time, an address merely consists of an unsigned thirty-two-bit integer value. That is, an address by itself does not specify the size of an item in memory. An x86 assembler uses heuristics to *infer* the data size whenever possible. For example, because the following instruction moves a value from memory into register EAX, which is four bytes long, an assembler will infer that the memory address refers to a four-byte value. For example, in Intel notation, the instruction is written:

```
mov   eax, [ebx+esi+12]
```

Similarly, if a name has been assigned to a data item that is declared to be a single byte, an assembler infers that a memory reference to the name refers to one byte. In some cases, however, a programmer must use an explicit *size directive* to specify the size of a data item. For example, suppose a programmer wishes to store –1 in a sixteen-bit word in memory. The programmer computes a memory address, which is placed in register EAX. An assembler cannot know that the programmer thinks of the address as pointing to a sixteen-bit (i.e., two-byte) data item, and will infer that it refers to a four-byte item. Therefore, a programmer must add a size directive before the memory reference, as in the following example that uses Intel notation:

```
mov   WORD PTR [eax], -1
```

It is good programming practice to use a size directive to make the intention clear if there is any doubt, even in cases where the inference rules of the assembler produce the correct result. Figure A3.11 summarizes the three size directives available.

Directive	Meaning
BYTE PTR	The address refers to a single byte in memory
WORD PTR	The address refers to a 16-bit value in memory
DWORD PTR	The address refers to a 32-bit value in memory

Figure A3.11 Size directives that can be prepended to memory references for the Intel assembler.

A3.11 Computing An Address

We said that an integer value can be computed, placed in a register, and then used as a memory address. However, most address computation begins with a known location in memory, such as the initial location of an array. For example, for the Intel assembler, suppose an array of four-byte integers has been declared using the name *iarray* and initialized to zero:

```
iarray    DB   1000 DUP(0)
```

The memory location of the i^{th} element can be computed by multiplying *i* by four (because each element is four bytes long) and adding the result to the address of the first byte of the array.

How can a running program obtain the address of the first byte of an array? More generally, how can a program obtain the memory address of an arbitrary variable? The answer lies in a special instruction that loads an address into a register rather than a value. Specified by the name *load effective address* and the mnemonic *lea*, the special instruction takes a register and a memory location as operands. Unlike the *mov* instruction, *lea* does not access an item in memory. Instead, *lea* stops after it computes the memory address, and places the address in the specified register. For example,

```
lea     eax, [iarray]
```

places the memory address of the first byte of item *iarray* in register EAX.

Observe that computing the offset of the i^{th} element of an array of four-byte integers is straightforward. First, place *i* in a register, for example, EBX. Once the index is in the register, the memory location corresponding to that element of the array can be computed with a single *lea* instruction:

```
mov   ebx, [i]              ; obtain index from variable i in memory
lea   eax, [4*ebx+iarray]   ; place address of ith element in EAX
```

A3.12 The Stack Operations Push And Pop

The x86 hardware includes instructions that manipulate a memory *stack*. The stack operates as a *Last-In-First-Out* (*LIFO*) data structure, with the most recently added item being accessed first. Like the stack on other processors, an x86 stack grows downward, with new items being added at successively lower memory addresses. Despite growing downward in memory, we say that the most recently added item is on the "top" of the stack.

When an item is added to a stack, we say the item is *pushed* onto the stack, and the top of the stack corresponds to the new item. When the top item is removed from the stack, we say that the stack has been *popped*.

An x86 stack always uses four-byte items — when an item is pushed onto a stack, four additional bytes of memory are used. Similarly, when an item is popped from a stack, the item contains four bytes, and the stack occupies four fewer bytes of memory.

In an x86, the ESP register (stack pointer) contains the current address of the top of the stack. Thus, although ESP does not appear explicitly, stack manipulation instructions always change the value in ESP. The names of stack instructions reflect the generic terminology described above: *push* and *pop*. Figure A3.12 lists the allowable argument types.

```
push   <reg32>                pop   <reg32>
push   <mem>                  pop   <mem>
push   <con32>
```

Figure A3.12 Operands allowed with the *push* and *pop* instructions.

Once register ESP has been set, adding items to the stack is trivial. For example, the instruction:

```
push    eax
```

pushes the value of register EAX onto the stack, and the instruction:

```
pop     [qqqq]
```

pops the top of the stack and places the value in the memory location with name *qqqq*. Similarly, the instruction:

```
push    -1
```

pushes the constant -1 onto the stack. The x86 hardware does not have a stack bound, which means that a programmer must plan stack use carefully to avoid situations in which a stack grows downward into a memory area used for other variables.

A3.13 Flow Of Control And Unconditional Branch

Normally, after a statement is executed, the processor proceeds to the next statement. An x86 supports three types of instructions that change the flow of control:

- unconditional branch
- conditional branch
- procedure call and return

An *unconditional branch* instruction is the easiest to understand: the opcode is *jmp* (for “jump”), and the only operand is the label on a statement. When it encounters a *jmp* instruction, the processor immediately moves to the specified label and continues execution. For example,

```
jmp     prntname
```

means the next instruction the processor will execute is the instruction with label *prntname*. The programmer must have placed the label on an instruction (presumably the first instruction in a sequence that prints a name). In Intel notation, the programmer writes:

```
prntname:  mov      eax, [nam]
             .
             .
             .
```

A3.14 Conditional Branch And Condition Codes

Each arithmetic instruction sets an internal value in the processor known as a *condition code*. A *conditional branch* instruction uses the value of the condition code to choose whether to branch or continue execution with the next sequential statement. A set of conditional branch instructions exists; each instruction encodes a specific test. Figure A3.13 summarizes.

Opcode	Meaning
jeq	jump if equal
jne	jump if not equal
jz	jump if zero
jnz	jump if not zero
jg	jump if greater than
jge	jump if greater than or equal
jl	jump if less than
jle	jump if less than or equal

Figure A3.13 Conditional branch instructions and the meaning of each.

For example, the following code in Intel notation decrements register EBX and jumps to label *atzero* if the resulting value is zero.

```
dec   ebx          ; subtract 1 from ebx
jz    atzero       ; jump to label atzero if EBX reaches zero
```

Some of the instructions in Figure A3.13 require a programmer to compare two items. For example, *jge* tests for greater-than-or-equal. However, conditional branch instructions do not perform comparisons — they have a single operand that consists of a label specifying where to branch. As with arithmetic tests, conditional branches involving a comparison rely on the condition code. Various instructions set the condition code, which means that a conditional branch can be executed immediately after the condition code has been set. If a conditional branch does not immediately follow the instruction that sets the condition code, a programmer must code an extra instruction that sets the condition. The x86 architecture includes two instructions used to set a condition code: *test* and *cmp*. Neither of the two modifies the contents of registers or memory. Instead, they merely compare two values and set the condition code. The *cmp* instruction checks for equality. In essence, a *cmp* performs a subtraction and then discards the answer, keeping only the condition code. For example, the following code in Intel notation tests whether the four-byte value in memory location *var1* has the value 123, and jumps to label *bb* if it does.

```
cmp  DWORD PTR [var1], 123   ; compare memory item var1 to 123
jeq  bb                      ; jump to label bbb if they are equal
```

The *test* instruction is more sophisticated: it performs a bit-wise *and* of the two operands, and sets various condition code bits accordingly. As a result, *test* sets conditions such as whether a data value contains odd or even parity.

A3.15 Subprogram Call And Return

The x86 hardware supports *subroutine invocation* (i.e., the ability to call a subprogram and have the subprogram return to its caller). Subroutine invocation forms a key part of the run-time support needed for a high-level procedural language.

Figure A3.14 summarizes the two x86 instructions that make subprograms possible: one instruction is used to invoke a subprogram and the other is used by a subprogram to return to its caller.

```
call      <label>
ret
```

Figure A3.14 Instructions used to invoke a subprogram: *call* invokes a subprogram, and *ret* returns to the caller.

Subprogram call and return use the run-time stack. For example, a *call* instruction pushes a return address on the stack. The next section discusses details.

A3.16 C Calling Conventions And Argument Passing

The term *calling conventions* refers to rules that calling and called programs use to guarantee agreement about details, such as the location of arguments. Calling conventions assign responsibilities to the calling program and the called subprogram. For example, the conventions specify exactly how a calling program pushes arguments on the stack for the subprogram to use, and exactly how a subprogram can return a value for the calling program to use.

Each high-level language defines a set of calling conventions. We will use the popular C calling conventions in examples. Although the conventions are intended to allow C or C++ programs to invoke an assembly language program and an assembly language program to invoke C functions, C calling conventions can also be used when an assembly language program invokes an assembly language subprogram. Thus, our examples are general.

The easiest way to understand calling conventions is to visualize the contents of a run-time stack when a subprogram is invoked. Our example consists of a call that passes three integer arguments (four bytes per argument) with values 100, 200, and 300 to a subprogram that has four local variables, each of which is thirty-two bits. The calling conventions specify the following during a call:

- *Caller Actions*. The caller pushes the values of registers EAX, ECX, and EDX onto the stack to save them. The caller then pushes arguments onto the stack in reverse order. Thus, if the arguments are 100, 200, and 300, the caller pushes 300, pushes 200, and then pushes 100. Finally, the caller invokes the *call* instruction, which pushes the *return address* (i.e., the address immediately following the *call* instruction) onto the stack and jumps to the subprogram.

- *Called Subprogram Actions*. The called subprogram pushes the EBP register onto the stack, and sets the EBP to the current top of the stack. The caller pushes the EBX, EDI, and ESI registers onto the stack, and then pushes each local variable onto the stack (or merely changes the stack pointer to allocate space if a local variable is uninitialized).

Figure A3.15 illustrates the stack immediately after a subprogram call has occurred (i.e., after both the caller and called subprogram have followed the conventions outlined above). To understand the figure, remember that a stack grows downward in memory. That is, a *push* operation decrements the stack pointer and a *pop* operation increments the pointer.

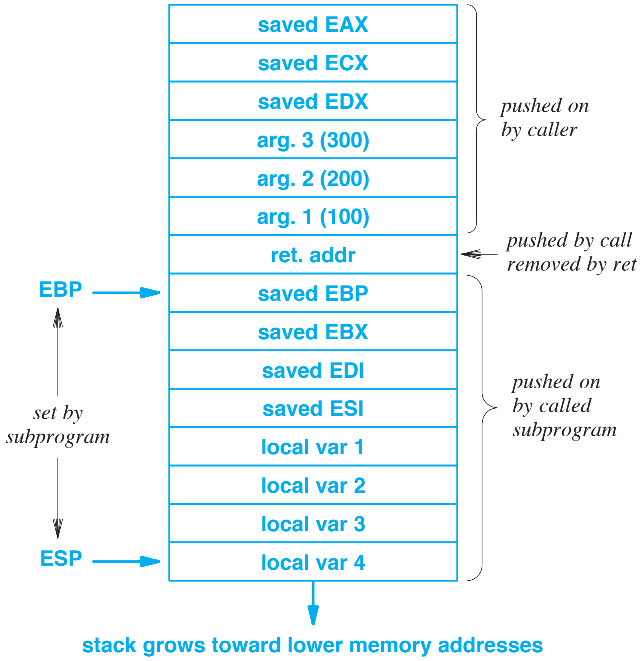


Figure A3.15 Illustration of the run-time stack after a subprogram has been called with three arguments and the subprogram has reserved space for four local variables.

When it finishes, the called subprogram must undo actions taken during the call and return to its caller. The following specifies steps the subprogram and caller take during a return.

- *Called Subprogram Return Actions*. The called subprogram deallocates local variables from the stack. To do so, the subprogram adds *4N* bytes to the stack pointer, where *N* is the number of local variables (each local variable is assumed to be four bytes long). The subprogram then restores the ESI, EDI, EBX, and EBP registers by popping their saved values off the stack. Finally, the subprogram executes a *ret* instruction to pop the return address from the stack and jump back to the caller.

- *Caller Return Actions*. When a called subprogram returns, the caller deallocates the arguments (e.g., by adding a constant to the stack pointer equal to four times the number of arguments). Finally, the caller restores the values of EDX, ECX and EAX.

A3.17 Function Calls And A Return Value

Technically, the above set of calling conventions applies to a *procedure call*. In the case of a *function call*, the subprogram must return a value to the caller. By convention, the return value is passed in register EAX. Therefore, when a function is invoked, the above calling conventions are modified so the caller does not restore the saved value of EAX.

Does it make sense for a caller to save EAX on the stack before calling a function? Once the function returns, EAX will contain the returned value. However, there are two reasons why EAX should be saved. First, a symbolic debugger will expect the stack to have the same layout for each procedure or function that has been called. Second, a caller may choose to continue computation after saving the result from a function. For example, suppose a compiler has used EAX to hold an index variable for a loop. If the loop contains a statement such as:

```
r = f(t);
```

the compiler may generate code to save the value of EAX before the call, store the return value in memory location *r* immediately after function *f* returns, and then restore EAX and allow the loop to continue.

A3.18 Extensions To Sixty-four Bits (x64)

The x86 architecture has been expanded to a sixty-four-bit version. Interestingly, AMD Corporation defined an extension scheme that was eventually adopted by Intel and other vendors. Known as *x86-64*, and often shortened to *x64*, the architecture includes many changes. For example, arithmetic and logical instructions, instructions that involve two registers, instructions that involve a register and memory location, and instructions that involve two memory locations have all been extended to operate on sixty-four-bit quantities. The stack operations have been changed so they push and pop sixty-four bits (eight bytes) at a time, and pointers are sixty-four-bits wide. The two changes most pertinent to our discussion involve general-purpose registers:

- Each general-purpose register has been extended to make it sixty-four-bits long.
- Eight additional general-purpose registers have been added, making a total of sixteen general-purpose registers.

As in the x86, the x64 architecture attempts to preserve backward compatibility. For example, the lower half of each sixty-four-bit register can be referenced as a thirty-two-bit register. Furthermore, it is possible to reference the sixteen-bit and eight-bit parts of the first four registers exactly as in the x86. Figure A3.16 illustrates the general-purpose registers available in the x64; readers should compare the figure with Figure A3.1 on page 474.

A3.19 Summary

We reviewed x86 fundamentals, including data declarations, registers, operand types, basic instructions, arithmetic and logical instructions, memory references, stack operations, conditional and unconditional branch, and subprogram invocation. Because the x86 architecture provides many instructions, a programmer may have a choice of multiple mechanisms to perform a given task. A sixty-four-bit extension has been designed that is known by the name *x64*.

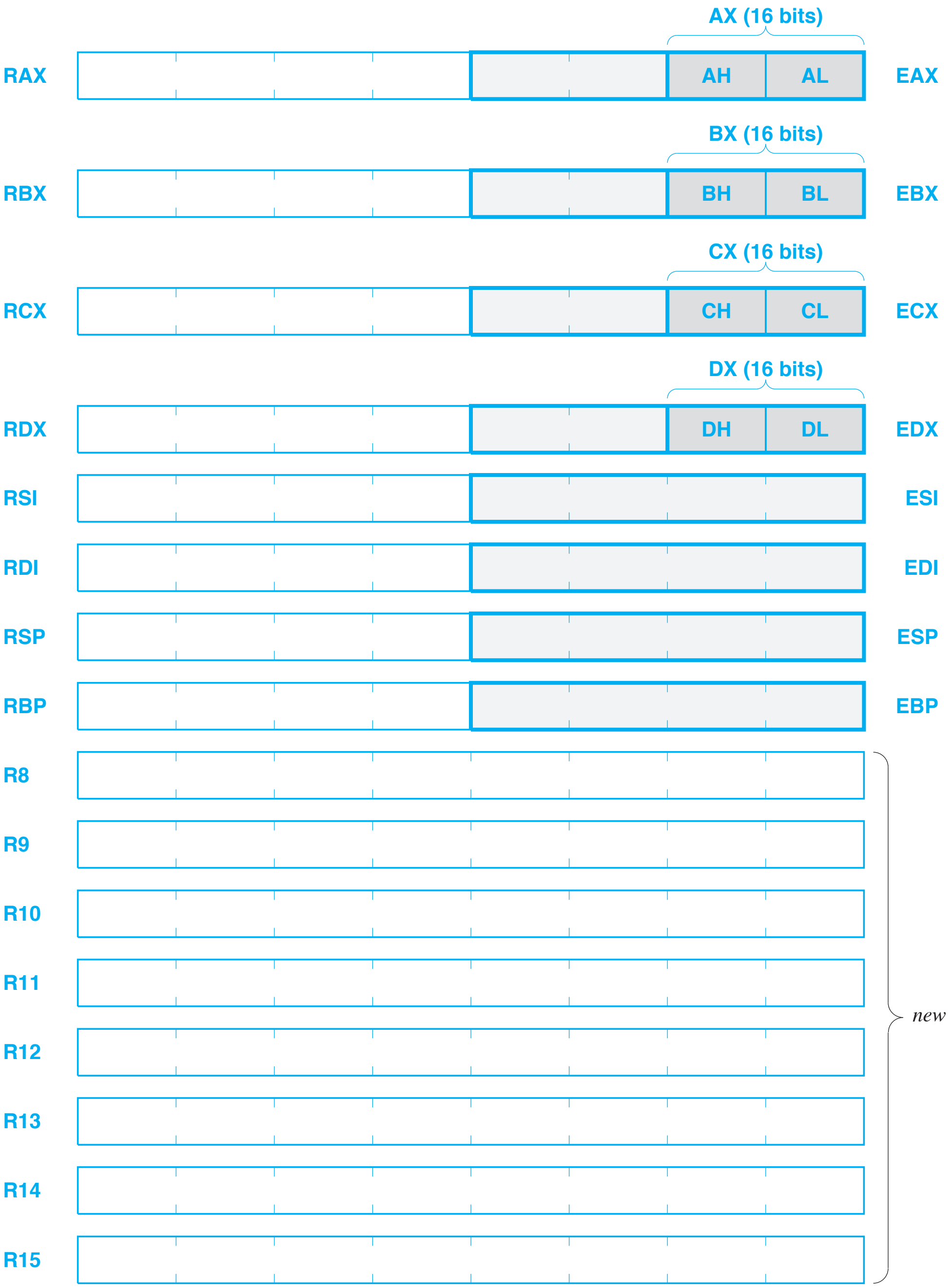


Figure A3.16 General-purpose registers in the x64 architecture.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Appendix 4

ARM Register Definitions And Calling Sequence

A4.1 Introduction

The previous appendix presents an overview of the x86 and x64 architectures. As we have seen, the x86 is a canonical example of a CISC instruction set. This appendix continues the discussion by providing information about the ARM architecture. ARM provides a canonical example of a RISC architecture.

Although ARM has defined a set of processors, this appendix focuses on features that are common to most of the thirty-two-bit ARM products. The reader is referred to ARM documentation for details about specific models.

A4.2 Registers On An ARM Processor

An ARM processor has 16 general-purpose registers numbered 0 through 15, and generally denoted with names *r0* through *r15*. Registers r0 through r3 are used to pass arguments to a called subroutine and to pass results back to the caller. Registers r4 through r11 are used to hold local variables for the subroutine that is currently being run. Register r12 is an *intra-procedural call scratch register*. Register r13 is the *stack pointer*. Register r14 is a *link register*, and is used in a subroutine call. Finally, register r15 is the *program counter* (i.e., an instruction pointer). Thus, loading an address into r15 causes the processor to branch to the address. Figure A4.1 summarizes the purpose of the registers, and gives alternate names used by the gcc assembler.

Register	Name	Purpose _
r15	pc	Program counter
r14	lr	Link register during function call
r13	sp	Stack pointer
r12	ip	Intra-procedural scratch
r11	fp	Frame or argument pointer
r10	sl	Stack limit
r9	v6	Local variable 6 (or real frame pointer)
r8	v5	Local variable 5
r7	v4	Local variable 4
r6	v3	Local variable 3
r5	v2	Local variable 2
r4	v1	Local variable 1
r3	a4	Argument 4 during a function call
r2	a3	Argument 3 during a function call
r1	a2	Argument 2 during a function call
r0	a1	Argument 1 during a function call

Figure A4.1 The general-purpose registers in an ARM architecture, the alternate name used in assembly language, and the meaning assigned to each register.

In addition to general-purpose registers, each ARM processor has a thirty-two-bit *Current Program Status Register* (*CPSR*). The CPSR is divided into many fields, including fields that control the processor mode and operation, control interrupts, report the condition code after an operation, report hardware errors, and control the endianness of the system. Figure A4.2 summarizes the bit fields in the CPSR.

A4.3 ARM Calling Conventions

Programming languages support a call mechanism in which a piece of code calls a subroutine, the subroutine executes, and control passes back to the point at which the call occurred. In terms of the run-time environment, subroutine calls are pushed onto the run-time stack. We say that code becomes a *caller* when it invokes a subroutine, and use the term *callee* to refer to the subroutine that is invoked. In the C Programming Language, a subroutine is known as a *function*; we will use the term throughout the remainder of the appendix.

Although the hardware places constraints on function invocation, a programmer or a compiler is free to choose some of the details. Throughout this chapter, we will describe the calling conventions that *gcc* follows, which have become widely accepted.

Name	Bit Range	Purpose
N	31	Negative/less than
Z	30	Zero
C	29	Carry/borrow/extend
V	28	Overflow
Q	27	Sticky overflow
J	24	Java state
DNM	20 – 23	Do not modify
GE	16-19	Greater-than-or-equal-to
IT	10 – 15 and 25 – 26	The if-then state
E	9	Data endianness
A	8	Imprecise data abort disable
I	7	IRQ disable
F	6	FIQ disable
T	5	Thumb state
M	0 – 4	Processor mode

Figure A4.2 Bits in an ARM CPSR and the meaning of each.

The argument passing conventions for ARM have the following characteristics:

- Allow a caller to pass zero or more arguments to a callee
- Optimize access for the first four arguments
- Allow a callee to return a set of results to the caller
- Specify which registers the callee can change and which must be unchanged when the call returns
- Specify how the run-time stack is used when a function is called and returns

Many functions have four or fewer parameters. To optimize access for the first four arguments, the values are passed in general-purpose registers a1 through a4 (i.e., registers r0 through r3). Additional arguments are placed on the stack in memory. Because a callee can access the first four arguments merely by referencing a register, access is extremely fast.

A callee can use registers a1 through a4 to return a result to the called program. In most programming languages, a function only returns one result, which is found in register a1. If an argument or a result is larger than 32 bits, the value is placed in memory and the address is passed in an argument register.

Figure A4.3 shows an example of the stack layout immediately after a function call. The example will clarify the calling conventions and explain how register values are preserved during a function call.

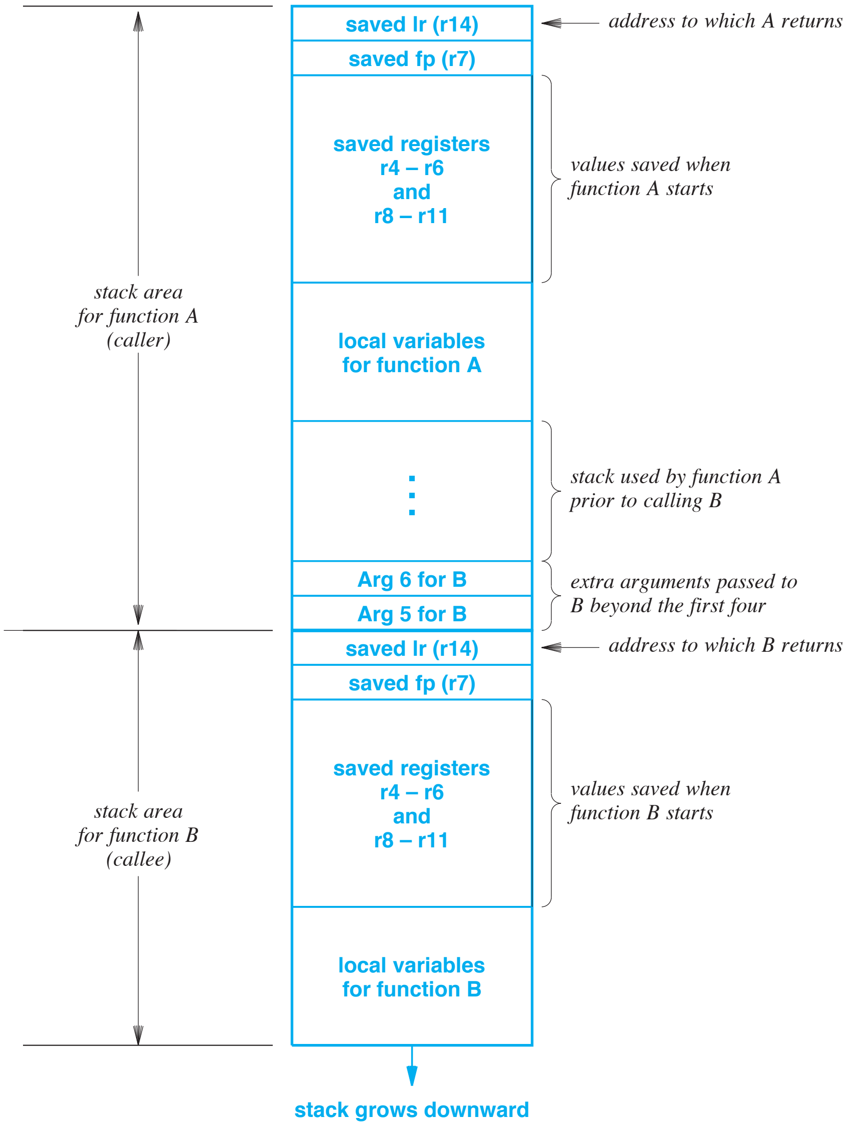

saved lr (r14)
saved fp (r7)
saved registers r4 – r6 and r8 – r11
local variables for function A
⋮
Arg 6 for B
Arg 5 for B
saved lr (r14)
saved fp (r7)
saved registers r4 – r6 and r8 – r11
local variables for function B
stack grows downward
address to which A returns
values saved when function A starts
stack used by function A prior to calling B
extra arguments passed to B beyond the first four
address to which B returns
values saved when function B starts
stack area for function A (caller)
stack area for function B (callee)

Figure A4.3 Layout of items on the run-time stack just after function A calls function B with six arguments.

In the figure, function *A* was executing and has called function *B*, which takes six arguments. The first four arguments are passed in registers†, which means they do not appear on the stack. However, arguments beyond the first four must be passed on the stack. Therefore, function A pushes arguments 5 and 6 onto the runtime stack in re-

†Recall that the first four arguments are passed in registers a1 through a4.

verse order before calling function B. As the figure shows, the extra arguments are the last two items on the stack when the call occurs.

A caller expects that values in most of the general-purpose registers will be *preserved* during a function call. That is, a caller expects the called function will not disturb register values. Of course, most functions need to use registers. Therefore, a called function saves the register contents upon entry and restores them before returning. As the figure illustrates, the prelude code in function B pushes the link register (r14), the frame pointer (r7), registers r4 through r6, and registers r8 through r11 onto the stack. The prelude code in function B then reserves space on the stack for its local variables (if any). Once local storage has been allocated, function B is ready to run. Before function B returns, postlude code in the function runs. The postlude code restores the registers from the saved values on the stack, and leaves the stack exactly as it was before the call.

Index