

# Unit-1

**CPU Architecture:** Instruction format, control signals in CPU, micro program control unit and hardwired control unit, ALU & sequencer, look ahead carry generator, MIPS ISA

## Unit 1

Sr. No.	Reference	Reading sequence
1.	Performance and Numericals	Chapter 1: Hennessy Patterson (COAD)
2.	Instruction Format, General Discussion	Chapter 2: Hamacher Zaky (CO)
3.	Instruction Cycle, Subcycle	Chapter 14: William Stallings
4.	Data path and related concepts and Control Signal; Hardwired, Microprogrammed Control	Chapter 5: Hamachar Zaky (CO)
5.	Control unit operations	Chapter 20 & 21: William Stallings
6.	Data path and control signal for RISC-V instructions ADD, LOAD etc.	Chapter 4.1 to 4.4: Hennessy Patterson (COAD)

---

# Computer Abstractions and Technology

---

## Abstract

This chapter explains that although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's very probable that they will be much better than they are today. To participate in these advances, computer designers and programmers must understand a wider variety of issues, including factors such as power, reliability, cost of ownership, and scalability. While this chapter focuses on cost, performance, and power, it emphasizes that the best designs will strike the appropriate balance for a given market among all the factors. This chapter also discusses the post-PC era, with personal mobile devices (PMDs) and tablets largely replacing desktop computers, and Cloud computing and warehouse scale computers (WSCs) taking over from the traditional server.

## Keywords

power wall; uniprocessor; multiprocessor; Intel Core i7; desktop computer; server; supercomputer; datacenter; embedded computer; multicore microprocessor; systems software; operating system; compiler; instruction; assembler; assembly language; machine language; high-level programming language; input device; output device; liquid crystal display; active matrix display; motherboard; integrated circuit; chip; memory; dynamic random access memory; DRAM; dual inline memory module; DIMM; central processor unit; CPU; datapath; control; cache memory; static random access memory; SRAM; abstraction; instruction set architecture; architecture; application binary interface; ABI; implementation; volatile memory; nonvolatile

memory; main memory; primary memory; secondary memory; magnetic disk; hard disk; flash memory; local area network; LAN; wide area network; WAN; vacuum tube; transistor; very large scale integrated circuit; VLSI; response time; execution time; throughput; bandwidth; CPU execution time; CPU time; user CPU time; system CPU time; clock cycle; clock period; clock cycles per instruction; CPI; instruction count; instruction mix; silicon; semiconductor; silicon crystal ingot; wafer; defect; die; yield; workload; benchmark; Amdahl's Law; million instructions per second; MIPS; cloud computing; mobile computing; cloud; mobile; personal mobile device; PMD; warehouse scale computer; WSC; Software as a Service; SaaS

*Civilization advances by extending the number of important operations which we can perform without thinking about them.*

*Alfred North Whitehead, An Introduction to Mathematics, 1911*

## OUTLINE

1.1	Introduction	3
1.2	Eight Great Ideas in Computer Architecture	11
1.3	Below Your Program	13
1.4	Under the Covers	16
1.5	Technologies for Building Processors and Memory	24
1.6	Performance	28
1.7	The Power Wall	40
1.8	The Sea Change: The Switch from Uniprocessors to Multiprocessors	43
1.9	Real Stuff: Benchmarking the Intel Core i7	46
1.10	Fallacies and Pitfalls	49
1.11	Concluding Remarks	52
	1.12 Historical Perspective and Further Reading	54
1.13	Exercises	54

## 1.1 Introduction

Welcome to this book! We're delighted to have this opportunity to

convey the excitement of the world of computer systems. This is not a dry and dreary field, where progress is glacial and where new ideas atrophy from neglect. No! Computers are the product of the incredibly vibrant information technology industry, all aspects of which are responsible for almost 10% of the gross national product of the United States, and whose economy has become dependent in part on the rapid improvements in information technology promised by Moore's Law. This unusual industry embraces innovation at a breath-taking rate. In the last 30 years, there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

This race to innovate has led to unprecedented progress since the inception of electronic computing in the late 1940s. Had the transportation industry kept pace with the computer industry, for example, today we could travel from New York to London in a second for a penny. Take just a moment to contemplate how such an improvement would change society—living in Tahiti while working in San Francisco, going to Moscow for an evening at the Bolshoi Ballet—and you can appreciate the implications of such a change.

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and changed the ways in which the search for new knowledge is carried out. There is now a new vein of scientific investigation, with computational scientists joining theoretical and experimental scientists in the exploration of new frontiers in astronomy, biology, chemistry, and physics, among others.

The computer revolution continues. Each time the cost of computing improves by another factor of 10, the opportunities for computers multiply. Applications that were economically infeasible suddenly become practical. In the recent past, the following applications were "computer science fiction."

- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce

pollution, improve fuel efficiency via engine controls, and increase safety through blind spot warnings, lane departure warnings, moving object detection, and air bag inflation to protect occupants in a crash.

- *Cell phones*: Who would have dreamed that advances in computer systems would lead to more than half of the planet having mobile phones, allowing person-to-person communication to almost anyone anywhere in the world?
- *Human genome project*: The cost of computer equipment to map and analyze human DNA sequences was hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 15 to 25 years earlier. Moreover, costs continue to drop; you will soon be able to acquire your own genome, allowing medical care to be tailored to you.
- *World Wide Web*: Not in existence at the time of the first edition of this book, the web has transformed our society. For many, the web has replaced libraries and newspapers.
- *Search engines*: As the content of the web grew in size and in value, finding relevant information became increasingly important. Today, many people rely on search engines for such a large part of their lives that it would be a hardship to go without them. Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent. Today's science fiction suggests tomorrow's killer applications: already on their way are glasses that augment reality, the cashless society, and cars that can drive themselves.

## Traditional Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see [Sections 1.4](#) and [1.5](#)) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have distinct design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three dissimilar classes of applications.

**Personal computers (PCs)** are possibly the best-known form of computing, which readers of this book have likely used extensively. Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. This class of computing drove the evolution of many computing technologies, which is merely 35 years old!

## personal computer (PC)

A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.

**Servers** are the modern form of what were once much larger computers, and are usually accessed only via a network. Servers are oriented to carrying sizable workloads, which may consist of either single complex applications—usually a scientific or engineering application—or handling many small jobs, such as would occur in building a large web server. These applications are usually based on software from another source (such as a database or simulation system), but are often modified or customized for a particular function. Servers are built from the same basic technology as desktop computers, but provide for greater computing, storage, and input/output capacity. In general, servers also place a higher emphasis on dependability, since a crash is usually more costly than it would be on a single-user PC.

## server

A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop computer without a screen or keyboard and cost a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple web serving. At the other extreme are **supercomputers**, which at the present consist of tens of thousands of processors and many **terabytes** of memory, and cost tens to hundreds of millions of dollars. Supercomputers are usually used for high-end scientific and engineering calculations, such as

weather forecasting, oil exploration, protein structure determination, and other large-scale problems. Although such supercomputers represent the peak of computing capability, they represent a relatively small fraction of the servers and thus a proportionally tiny fraction of the overall computer market in terms of total revenue.

## supercomputer

A class of computers with the highest performance and cost; they are configured as servers and typically cost tens to hundreds of millions of dollars.

## terabyte (TB)

Originally 1,099,511,627,776 ( $2^{40}$ ) bytes, although communications and secondary storage systems developers started using the term to mean 1,000,000,000,000 ( $10^{12}$ ) bytes. To reduce confusion, we now use the term **tebibyte (TiB)** for  $2^{40}$  bytes, defining *terabyte* (TB) to mean  $10^{12}$  bytes. [Figure 1.1](#) shows the full range of decimal and binary values and names.

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	$10^3$	kibibyte	KiB	$2^{10}$	2%
megabyte	MB	$10^6$	mebibyte	MiB	$2^{20}$	5%
gigabyte	GB	$10^9$	gibibyte	GiB	$2^{30}$	7%
terabyte	TB	$10^{12}$	tebibyte	TiB	$2^{40}$	10%
petabyte	PB	$10^{15}$	pebibyte	PiB	$2^{50}$	13%
exabyte	EB	$10^{18}$	exbibyte	EiB	$2^{60}$	15%
zettabyte	ZB	$10^{21}$	zebibyte	ZiB	$2^{70}$	18%
yottabyte	YB	$10^{24}$	yobibyte	YiB	$2^{80}$	21%

**FIGURE 1.1** The  $2^X$  vs.  $10^Y$  bytes ambiguity was resolved by adding a binary notation for all the common size terms.

In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is  $10^9$  bits while *gibibits* (Gib) is  $2^{30}$  bits.

**Embedded computers** are the largest class of computers and span the widest range of applications and performance. Embedded computers include the microprocessors found in your car, the computers in a television set, and the networks of processors that control a modern airplane or cargo ship. Embedded computing systems are designed to run one application or one set of related applications that are normally integrated with the hardware and delivered as a single system; thus, despite the large number of embedded computers, most users never really see that they are using a computer!

### **embedded computer**

A computer inside another device used for running one predetermined application or collection of software.

Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a music player: the processor need only to be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power is the most important objective. Despite their low cost, embedded computers often have lower tolerance for failure, since the results can vary from upsetting (when your new television crashes) to devastating (such as might occur when the computer in a plane or cargo ship crashes). In consumer-oriented embedded applications, such as a digital home appliance, dependability is achieved primarily through simplicity—the emphasis is on doing one function as perfectly as possible. In large embedded systems, techniques of redundancy from the server world are often employed. Although this book focuses on general-purpose computers, most concepts apply directly, or with slight modifications, to embedded computers.

### **Elaboration**

Elaborations are short sections used throughout the text to provide more detail on a particular subject that may be of interest. Disinterested readers may skip over an elaboration, since the subsequent material will never depend on the contents of the

elaboration.

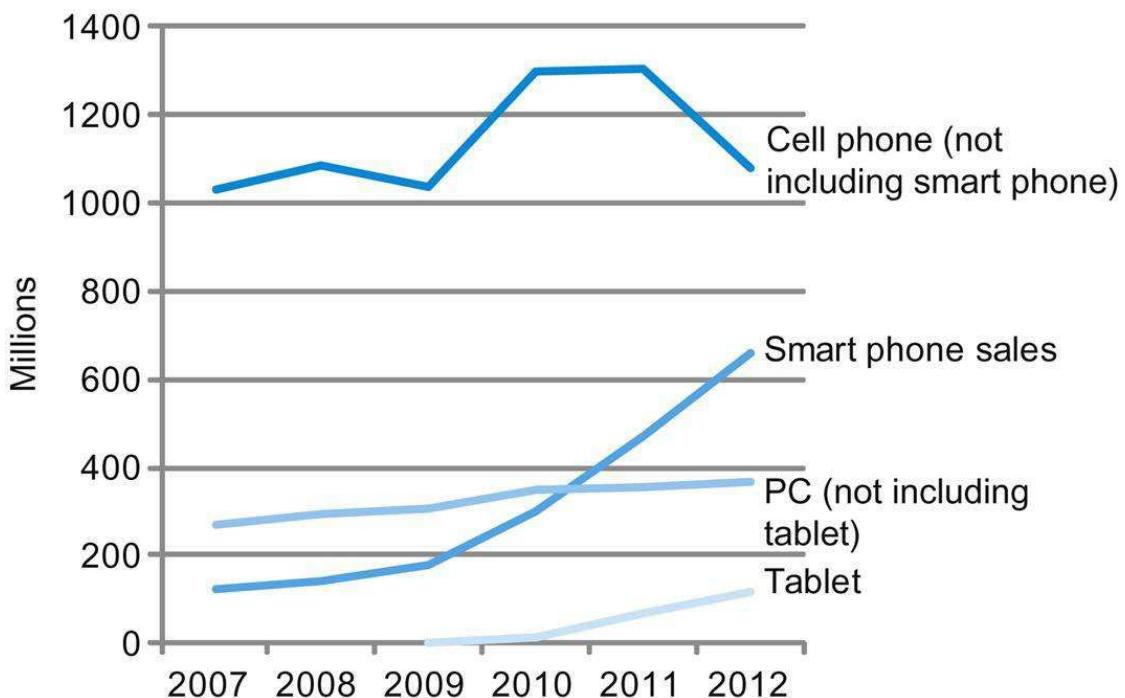
Many embedded processors are designed using *processor cores*, a version of a processor written in a hardware description language, such as Verilog or VHDL (see [Chapter 4](#)). The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip.

## Welcome to the Post-PC Era

The continuing march of technology brings about generational changes in computer hardware that shake up the entire information technology industry. Since the last edition of the book, we have undergone such a change, as significant in the past as the switch starting 30 years ago to personal computers. Replacing the PC is the **personal mobile device (PMD)**. PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software (“apps”) to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input. Today’s PMD is a smart phone or a tablet computer, but tomorrow it may include electronic glasses. [Figure 1.2](#) shows the rapid growth over time of tablets and smart phones versus that of PCs and traditional cell phones.

### Personal mobile devices (PMDs)

are small wireless devices to connect to the Internet; they rely on batteries for power, and software is installed by downloading apps. Conventional examples are smart phones and tablets.



**FIGURE 1.2** The number manufactured per year of tablets and smart phones, which reflect the post-PC era, versus personal computers and traditional cell phones.

Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly doubling between 2011 and 2012. Recent PCs and traditional cell phone categories are relatively flat or declining.

Taking over from the conventional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs). Companies like Amazon and Google build these WSCs containing 100,000 servers and then let companies rent portions of them so that they can provide software services to PMDs without having to build WSCs of their own. Indeed, **Software as a Service (SaaS)** deployed via the Cloud is revolutionizing the software industry just as PMDs and WSCs are revolutionizing the hardware industry. Today's software developers will often have a portion of their application that runs on the PMD and a portion that runs in the Cloud.

## Cloud Computing

refers to large collections of servers that provide services over the Internet; some providers rent dynamically varying numbers of

servers as a utility.

## Software as a Service (SaaS)

delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices, instead of binary code that must be installed, and runs wholly on that device. Examples include web search and social networking.

## What You Can Learn in This Book

Successful programmers have always been concerned about the performance of their programs, because getting results to the user quickly is critical in creating popular software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus, programmers often followed a simple credo: minimize memory space to make programs fast. In the last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. We demonstrate the importance of this understanding in [Chapters 3 to 6](#) by showing how to improve performance of a C program by a factor of 200. Moreover, as we explain in [Section 1.7](#), today's programmers need to worry about energy efficiency of their programs running either on the PMD or in the Cloud, which also requires understanding what is below your code. Programmers who seek to build competitive versions of software will therefore need to increase their knowledge of computer organization.

We are honored to have the opportunity to explain what's inside this revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer. By the time you complete this book, we believe you will be able to answer the following questions:

- How are programs written in a high-level language, such as C or Java, translated into the language of the hardware, and how does

the hardware execute the resulting program? Comprehending these concepts forms the basis of understanding the aspects of both the hardware and software that affect program performance.

- What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions? These concepts are vital to understanding how to write many kinds of software.
- What determines the performance of a program, and how can a programmer improve the performance? As we will see, this depends on the original program, the software translation of that program into the computer's language, and the effectiveness of the hardware in executing the program.
- What techniques can be used by hardware designers to improve performance? This book will introduce the basic concepts of modern computer design. The interested reader will find much more material on this topic in our advanced book, *Computer Architecture: A Quantitative Approach*.
- What techniques can be used by hardware designers to improve energy efficiency? What can the programmer do to help or hinder energy efficiency?
- What are the reasons for and the consequences of the recent switch from sequential processing to parallel processing? This book gives the motivation, describes the current hardware mechanisms to support parallelism, and surveys the new generation of “**multicore**” microprocessors (see [Chapter 6](#)).

### **multicore microprocessor**

A microprocessor containing multiple processors (“cores”) in a single integrated circuit.

- Since the first commercial computer in 1951, what great ideas did computer architects come up with that lay the foundation of modern computing?  
Without understanding the answers to these questions, improving the performance of your program on a modern computer or evaluating what features might make one computer better than another for a particular application will be a complex process of trial and error, rather than a scientific procedure driven by insight and analysis.

This first chapter lays the foundation for the rest of the book. It introduces the basic ideas and definitions, places the major components of software and hardware in perspective, shows how to evaluate performance and energy, introduces integrated circuits (the technology that fuels the computer revolution), and explains the shift to multicores.

In this chapter and later ones, you will likely see many new words, or words that you may have heard but are not sure what they mean. Don't panic! Yes, there is a lot of special terminology used in describing modern computers, but the terminology actually helps, since it enables us to describe precisely a function or capability. In addition, computer designers (including your authors) *love* using **acronyms**, which are *easy* to understand once you know what the letters stand for! To help you remember and locate terms, we have included a **highlighted** definition of every term in the margins the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends will be impressed as you correctly use acronyms such as BIOS, CPU, DIMM, DRAM, PCIe, SATA, and many others.

## acronym

A word constructed by taking the initial letters of a string of words. For example: **RAM** is an acronym for Random Access Memory, and **CPU** is an acronym for Central Processing Unit.

To reinforce how the software and hardware systems used to run a program will affect performance, we use a special section, *Understanding Program Performance*, throughout the book to summarize important insights into program performance. The first one appears below.

## Understanding Program Performance

The performance of a program depends on a combination of the effectiveness of the algorithms used in the program, the software systems used to create and translate the program into machine instructions, and the effectiveness of the computer in executing those instructions, which may include *input/output* (I/O) operations. This table summarizes how the hardware and software affect

performance.

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	<a href="#">Chapters 2 and 3</a>
Processor and memory system	Determines how fast instructions can be executed	<a href="#">Chapters 4, 5, and 6</a>
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	<a href="#">Chapters 4, 5, and 6</a>

To demonstrate the impact of the ideas in this book, as mentioned above, we improve the performance of a C program that multiplies a matrix times a vector in a sequence of chapters. Each step leverages understanding how the underlying hardware really works in a modern microprocessor to improve performance by a factor of 200!

- In the category of *data-level parallelism*, in [Chapter 3](#) we use *subword parallelism via C intrinsics* to increase performance by a factor of 3.8.
- In the category of *instruction-level parallelism*, in [Chapter 4](#) we use *loop unrolling to exploit multiple instruction issue and out-of-order execution hardware* to increase performance by another factor of 2.3.
- In the category of *memory hierarchy optimization*, in [Chapter 5](#) we use *cache blocking* to increase performance on large matrices by another factor of 2.0 to 2.5.
- In the category of *thread-level parallelism*, in [Chapter 6](#) we use *parallel for loops in OpenMP to exploit multicore hardware* to increase performance by another factor of 4 to 14.

## Check Yourself

*Check Yourself* sections are designed to help readers assess whether they comprehend the major concepts introduced in a chapter and understand the implications of those concepts. Some *Check Yourself* questions have simple answers; others are for discussion among a group. Answers to the specific questions can be found at the end of the chapter. *Check Yourself* questions appear only at the end of a section, making it easy to skip them if you are sure you understand

the material.

1. The number of embedded processors sold every year greatly outnumbers the number of PC and even post-PC processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of conventional computers in your home?
2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?
  - The algorithm chosen
  - The programming language or compiler
  - The operating system
  - The processor
  - The I/O system and devices

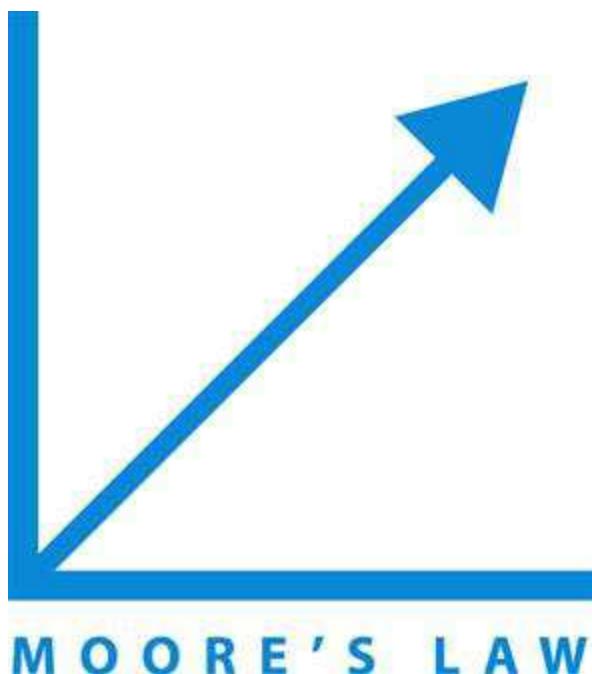
## 1.2 Eight Great Ideas in Computer Architecture

We now introduce eight great ideas that computer architects have invented in the last 60 years of computer design. These ideas are so powerful they have lasted long after the first computer that used them, with newer architects demonstrating their admiration by imitating their predecessors. These great ideas are themes that we will weave through this and subsequent chapters as examples arise. To point out their influence, in this section we introduce icons and highlighted terms that represent the great ideas and we use them to identify the nearly 100 sections of the book that feature use of the great ideas.

### Design for Moore's Law

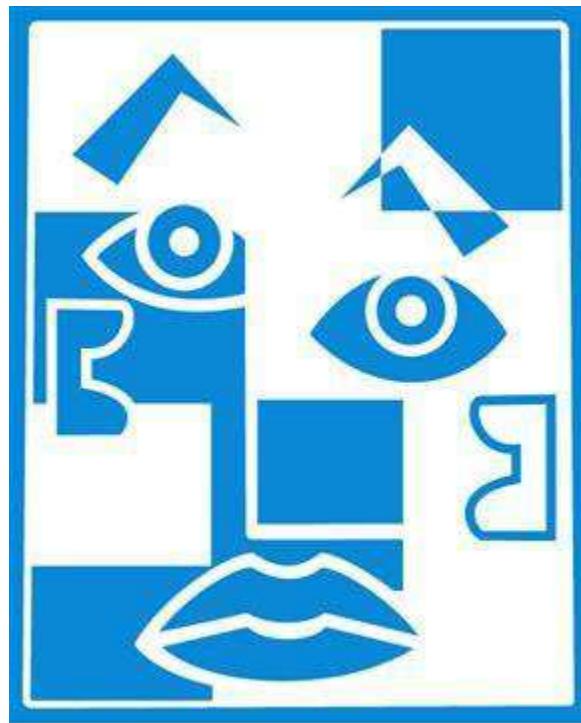
The one constant for computer designers is rapid change, which is driven largely by **Moore's Law**. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take

years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an “up and to the right” Moore’s Law graph to represent designing for rapid change.



## Use Abstraction to Simplify Design

Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore’s Law. A major productivity technique for hardware and software is to use **abstractions** to characterize the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We’ll use the abstract painting icon to represent this second great idea.



## A B S T R A C T I O N

### Make the Common Case Fast

Making the **common case fast** will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is usually easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement (see [Section 1.6](#)). We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan!



## COMMON CASE FAST

### Performance via Parallelism

Since the dawn of computing, computer architects have offered designs that get more performance by computing operations in parallel. We'll see many examples of parallelism in this book. We use multiple jet engines of a plane as our icon for **parallel performance**.

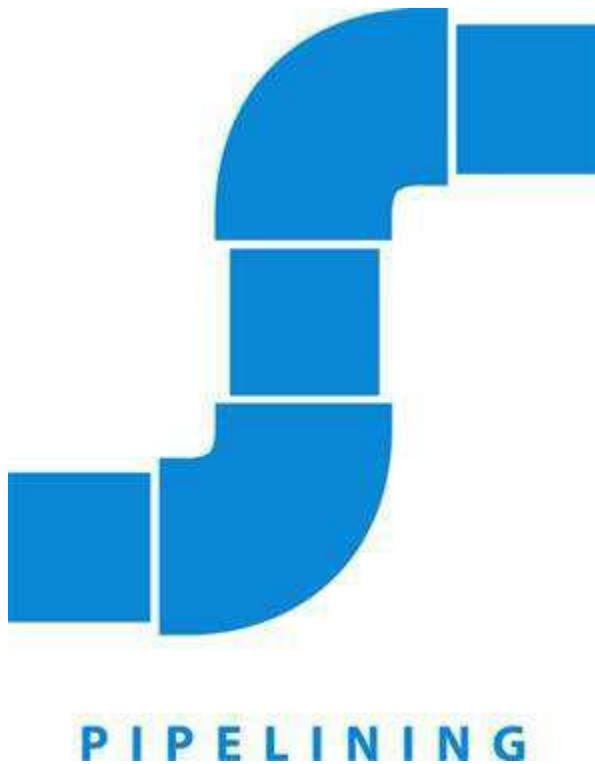


## PARALLELISM

### Performance via Pipelining

A particular pattern of parallelism is so prevalent in computer

architecture that it merits its own name: **pipelining**. For example, before fire engines, a “bucket brigade” would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. The townsfolk form a human chain to carry a water source to fire, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.



## PIPELINING

## Performance via Prediction

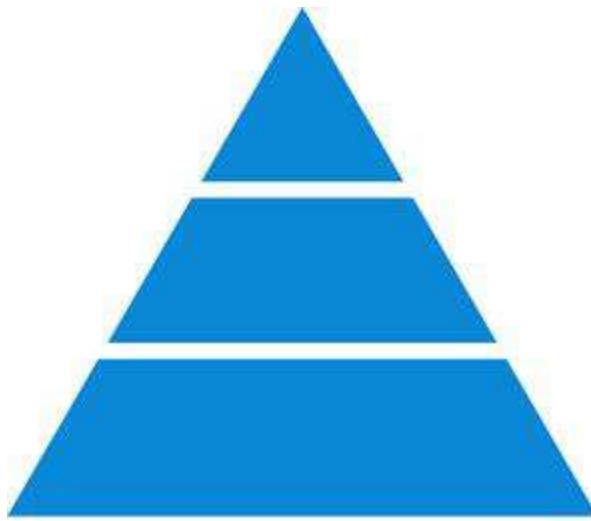
Following the saying that it can be better to ask for forgiveness than to ask for permission, the next great idea is **prediction**. In some cases, it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller’s crystal ball as our prediction icon.



## PREDICTION

### Hierarchy of Memories

Programmers want the memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy of memories**, with the fastest, smallest, and the most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. As we shall see in [Chapter 5](#), caches give the programmer the illusion that main memory is almost as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.



## HIERARCHY

### Dependability via Redundancy

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axles allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the flat tire can be fixed, thereby restoring redundancy!)



## DEPENDABILITY

### 1.3 Below Your Program

A typical application, such as a word processor or a large database

system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the primitive instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions, an example of the great idea of **abstraction**.



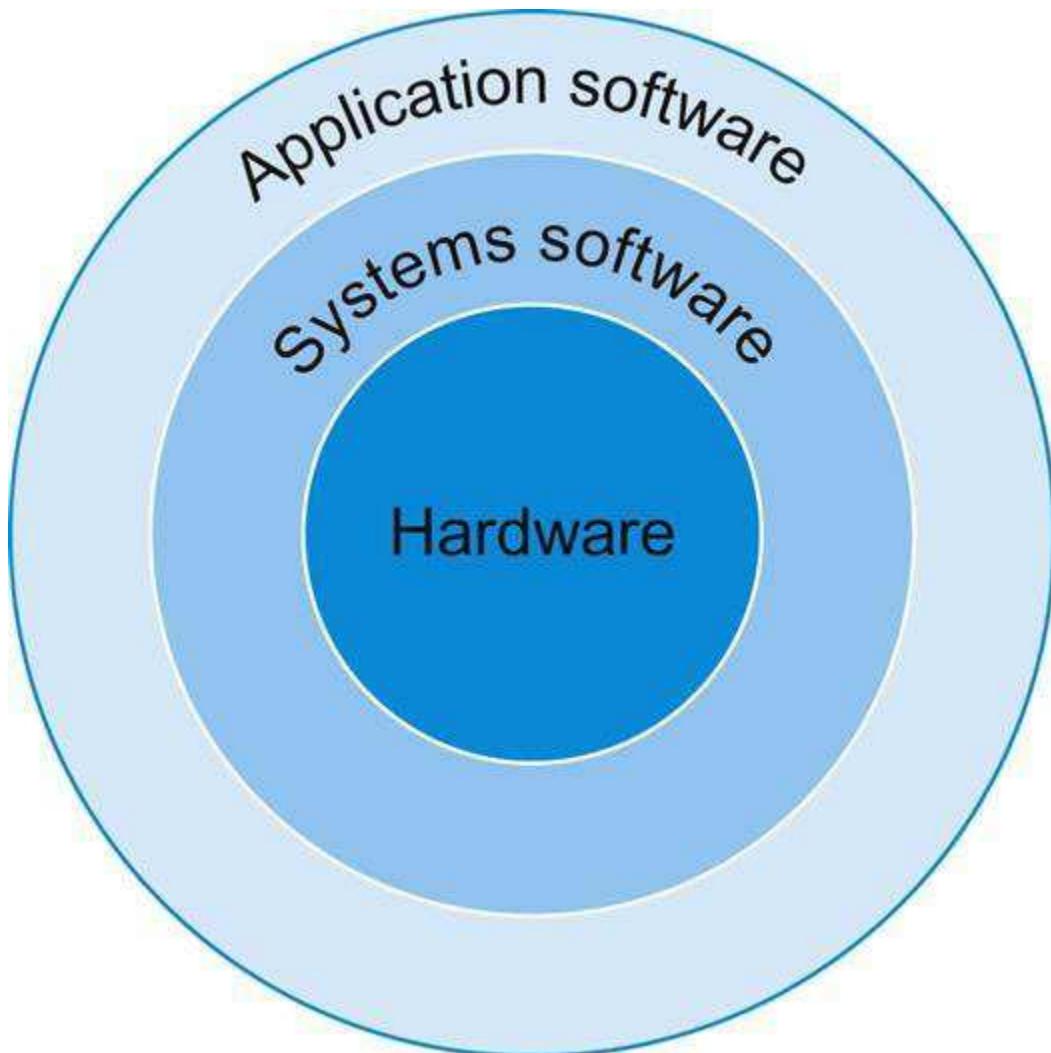
*In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.*

*Mark Twain, The Innocents Abroad, 1869*

[Figure 1.3](#) shows that these layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of **systems software** sitting between the hardware and the application software.

## **systems software**

Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.



**FIGURE 1.3** A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and application software outermost.

In complex applications, there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

There are many types of systems software, but two types of systems software are central to every computer system today: an operating system and a compiler. An **operating system** interfaces

between a user's program and the hardware and provides a variety of services and supervisory functions. Among the most important functions are:

- Handling basic input and output operations
- Allocating storage and memory
- Providing for protected sharing of the computer among multiple applications using it simultaneously

## operating system

Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.

Examples of operating systems in use today are Linux, iOS, and Windows.

**Compilers** perform another vital function: the translation of a program written in a high-level language, such as C, C++, Java, or Visual Basic into instructions that the hardware can execute. Given the sophistication of modern programming languages and the simplicity of the instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We give a brief overview of the process here and then go into more depth in [Chapter 2](#).

## compiler

A program that translates high-level language statements into assembly language statements.

## From a High-Level Language to the Language of Hardware

To speak directly to electronic hardware, you need to send electrical signals. The easiest signals for computers to understand are *on* and *off*, and so the computer alphabet is just two letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do. The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the computer language as numbers in base 2, or *binary numbers*. We refer to each

“letter” as a **binary digit** or **bit**. Computers are slaves to our commands, which are called **instructions**. Instructions, which are just collections of bits that the computer understands and obeys, can be thought of as numbers. For example, the bits

1001010100101110

tell one computer to add two numbers. [Chapter 2](#) explains why we use numbers for instructions *and* data; we don’t want to steal that chapter’s thunder, but using numbers for both instructions and data is a foundation of computing.

## binary digit

Also called a **bit**. One of the two numbers in base 2 (0 or 1) that are the components of information.

## instruction

A command that computer hardware understands and obeys.

The first programmers communicated to computers in binary numbers, but this was so tedious that they quickly invented new notations that were closer to the way humans think. At first, these notations were translated to binary by hand, but this process was still tiresome. Using the computer to help program the computer, the pioneers invented software to translate from symbolic notation to binary. The first of these programs was named an **assembler**. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

add A, B

and the assembler would translate this notation into

1001010100101110

## assembler

A program that translates a symbolic version of instructions into the binary version.

This instruction tells the computer to add the two numbers A and B. The name coined for this symbolic language, still used today, is **assembly language**. In contrast, the binary language that the machine understands is the **machine language**.

## assembly language

A symbolic representation of machine instructions.

## machine language

A binary representation of machine instructions.

Although a tremendous improvement, assembly language is still far from the notations a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books.

Assembly language requires the programmer to write one line for every instruction that the computer will follow, forcing the programmer to think like the computer.

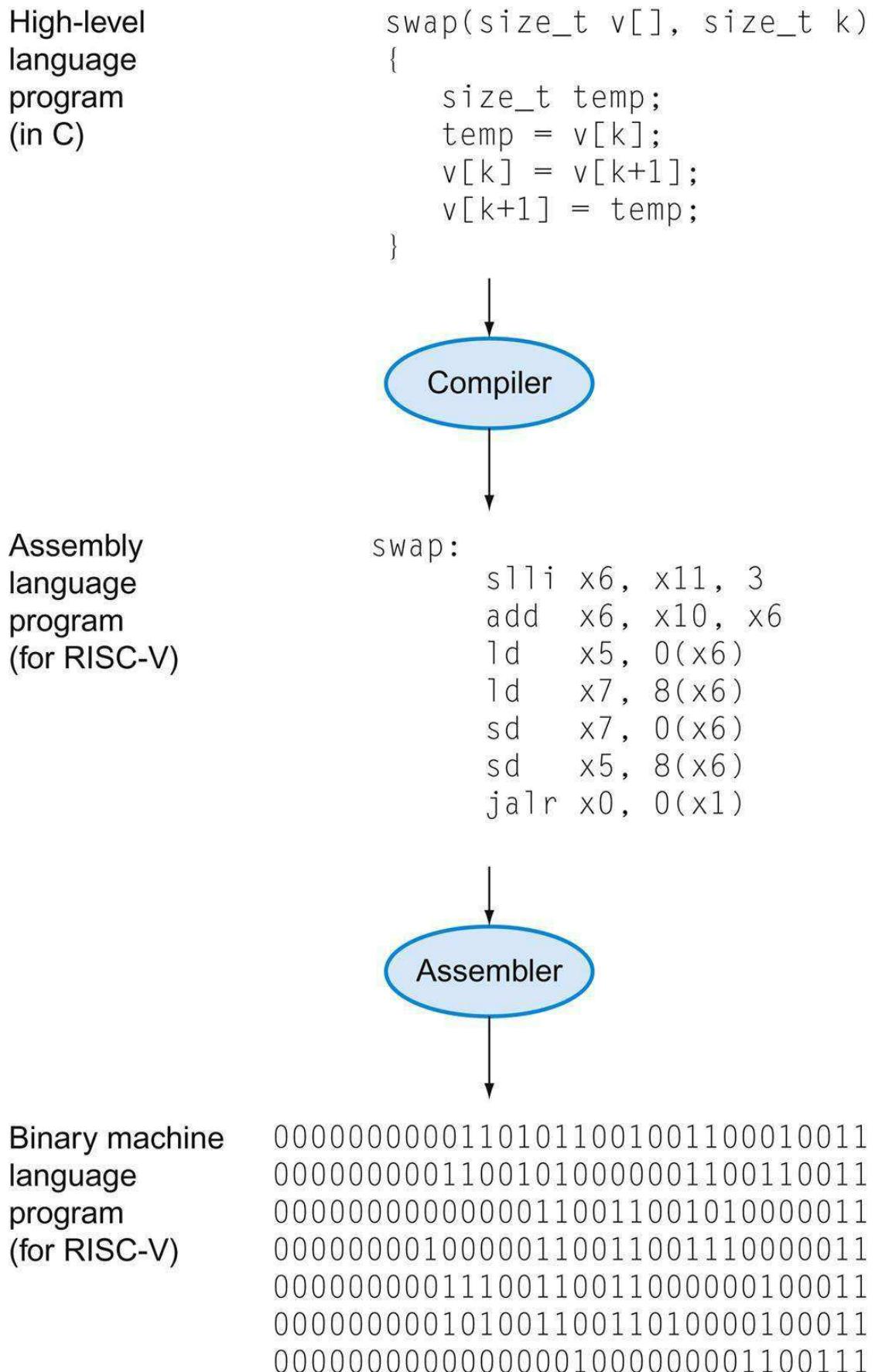
The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of **high-level programming languages** and compilers that translate programs in such languages into instructions. [Figure 1.4](#) shows the relationships among these programs and languages, which are more examples of the power of **abstraction**.



## A B S T R A C T I O N

### **high-level programming language**

A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.



**FIGURE 1.4 C program compiled into assembly**

## **language and then assembled into binary machine language.**

Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in [Chapter 2](#).

A compiler enables a programmer to write this high-level language expression:

A + B

The compiler would compile it into this assembly language statement:

add A, B

As shown above, the assembler would translate this statement into the binary instructions that tell the computer to add the two numbers A and B.

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see [Figure 1.4](#)). Moreover, they allow languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on. There are also domain-specific languages for even narrower groups of users, such as those interested in simulation of fluids, for example.

The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language.

The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any computer. These three advantages are so strong that today little programming is done in assembly language.

## 1.4 Under the Covers

Now that we have looked below your program to uncover the underlying software, let's open the covers of your computer to learn about the underlying hardware. The underlying hardware in any computer performs the same basic functions: inputting data, outputting data, processing data, and storing data. How these functions are performed is the primary topic of this book, and subsequent chapters deal with different parts of these four tasks.

When we come to an important point in this book, a point so significant that we hope you will remember it forever, we emphasize it by identifying it as a *Big Picture* item. We have about a dozen Big Pictures in this book, the first being the five components of a computer that perform the tasks of inputting, outputting, processing, and storing data.

Two key components of computers are **input devices**, such as the microphone, and **output devices**, such as the speaker. As the names suggest, input feeds the computer, and output is the result of computation sent to the user. Some devices, such as wireless networks, provide both input and output to the computer.

### input device

A mechanism through which the computer is fed information, such as a keyboard.

### output device

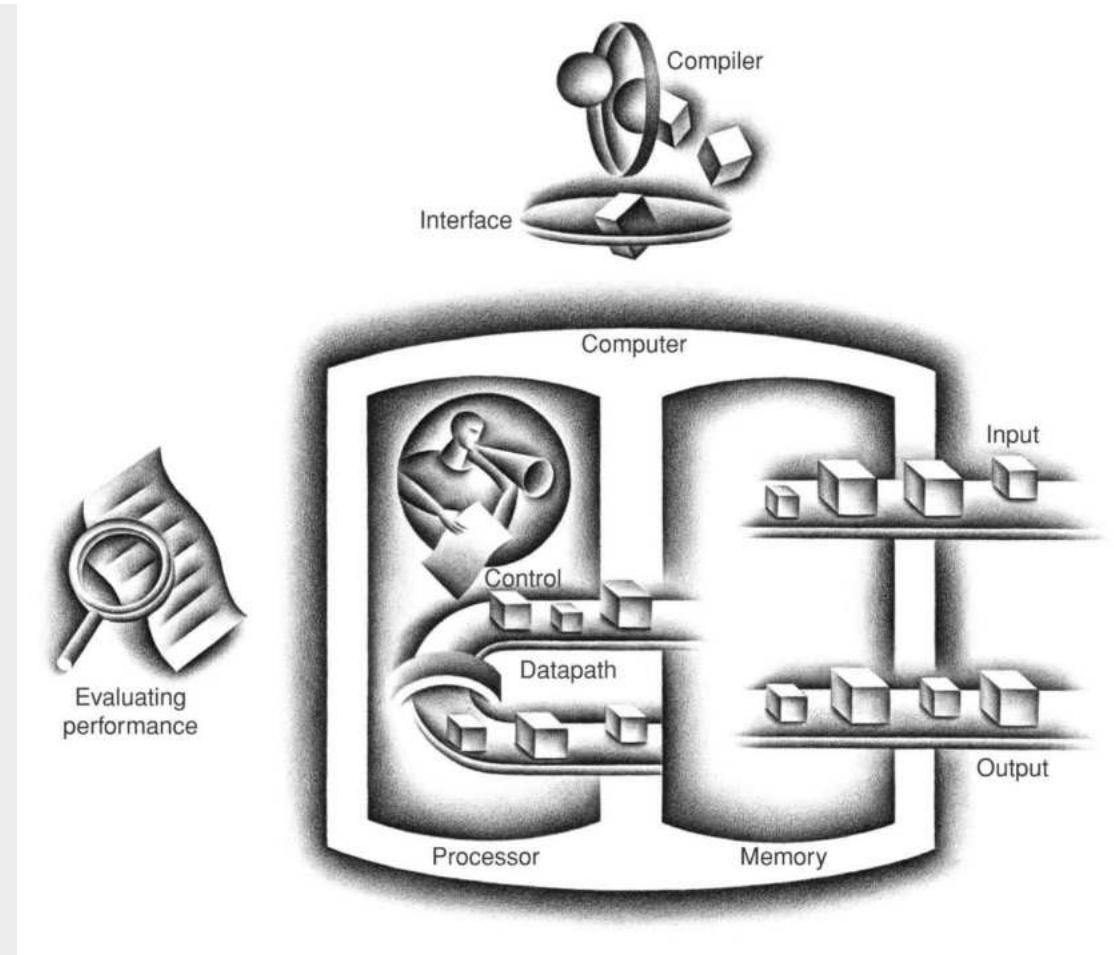
A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

[Chapters 5](#) and [6](#) describe *input/output* (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

### The BIG Picture

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. [Figure 1.5](#) shows the standard

organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of each of the following chapters, with the portion of interest to that chapter highlighted.



**FIGURE 1.5 The organization of a computer, showing the five classic components.**

The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

## Through the Looking Glass

The most fascinating I/O device is probably the graphics display. Most personal mobile devices use **liquid crystal displays (LCDs)** to get a thin, low-power display. The LCD is not the source of light; instead, it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display, from either a light source behind the display or less often from reflected light. The rods straighten out when a current is applied and no longer bend the light. Since the

liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent. Today, most LCDs use an **active matrix** that has a tiny transistor switch at each pixel to control current precisely and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three-color components in the final image; in a color active matrix LCD, there are three transistor switches at each point.

## liquid crystal display (LCD)

A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

## active matrix display

A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix in a typical tablet ranges in size from  $1024 \times 768$  to  $2048 \times 1536$ . A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

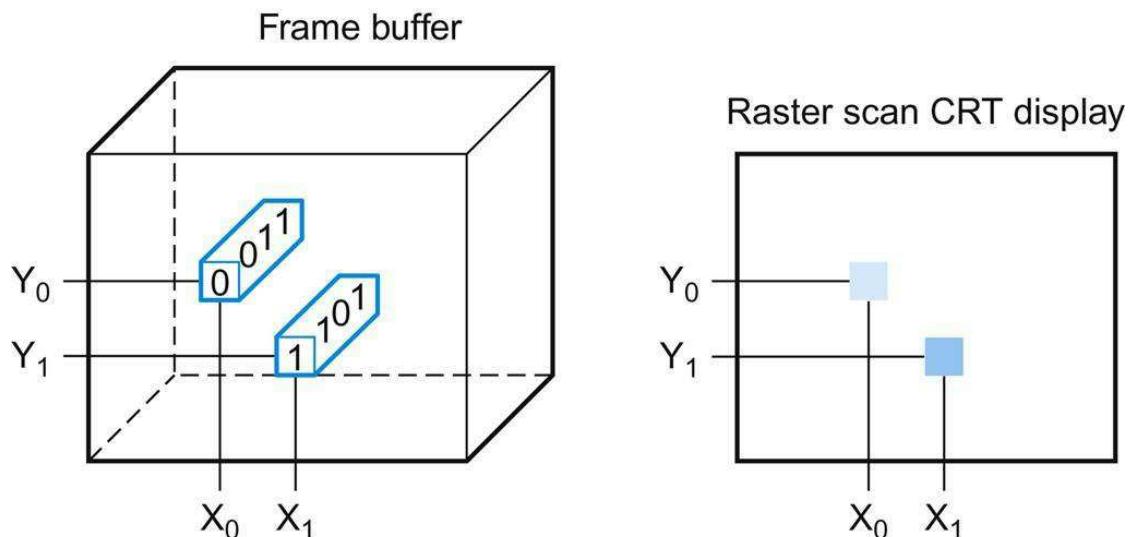
## pixel

The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

*Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.*

*Ivan Sutherland, the “father” of computer graphics, Scientific American,  
1984*

The computer hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented onscreen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. [Figure 1.6](#) shows a frame buffer with a simplified design of just 4 bits per pixel.



**FIGURE 1.6** Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right.

Pixel (X<sub>0</sub>, Y<sub>0</sub>) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X<sub>1</sub>, Y<sub>1</sub>).

The goal of the bit map is to represent faithfully what is on the screen. The challenges in graphics systems arise because the human eye is very good at detecting even subtle changes on the screen.

## Touchscreen

While PCs also use LCDs, the tablets and smartphones of the post-PC era have replaced the keyboard and mouse with touch-sensitive displays, which has the wonderful user interface advantage of users pointing directly at what they are interested in rather than indirectly with a mouse.

While there are a variety of ways to implement a touch screen,

many tablets today use capacitive sensing. Since people are electrical conductors, if an insulator like glass is covered with a transparent conductor, touching distorts the electrostatic field of the screen, which results in a change in capacitance. This technology can allow multiple touches simultaneously, which recognizes gestures that can lead to attractive user interfaces.

## Opening the Box

Figure 1.7 shows the contents of the Apple iPad 2 tablet computer. Unsurprisingly, of the five classic components of the computer, I/O dominates this reading device. The list of I/O devices includes a capacitive multitouch LCD, front-facing camera, rear-facing camera, microphone, headphone jack, speakers, accelerometer, gyroscope, Wi-Fi network, and Bluetooth network. The datapath, control, and memory are a tiny portion of the components.



**FIGURE 1.7 Components of the Apple iPad 2**

### A1395.

The metal back of the iPad (with the reversed Apple logo in the middle) is in the center. At the top is the capacitive multitouch screen and LCD. To the far right is the 3.8 V, 25 watt-hour, polymer battery, which consists of three Li-ion cell cases and offers 10 hours of battery life. To the far left is the metal frame that attaches the LCD to the back of the iPad. The small components surrounding the metal back in the center are what we think of as the computer; they are often L-shaped to fit compactly inside the case next to the battery. [Figure 1.8](#) shows a close-up of the L-shaped board to the lower left of the metal case, which is the logic printed circuit board that contains the processor and the memory. The tiny rectangle below the logic board contains a chip that provides wireless communication: Wi-Fi, Bluetooth, and FM tuner. It fits into a small slot in the lower left corner of the logic board. Near the upper left corner of the case is another L-shaped component, which is a front-facing camera assembly that includes the camera, headphone jack, and microphone. Near the right upper corner of the case is the board containing the volume control and silent/screen rotation lock button along with a gyroscope and accelerometer. These last two chips combine to allow the iPad to recognize six-axis motion. The tiny rectangle next to it is the rear-facing camera. Near the bottom right of the case is the L-shaped speaker assembly. The cable at the bottom is the connector between the logic board and the camera/volume control board. The board between the cable and the speaker assembly is the controller for the capacitive touchscreen. (Courtesy iFixit, [www.ifixit.com](http://www.ifixit.com))

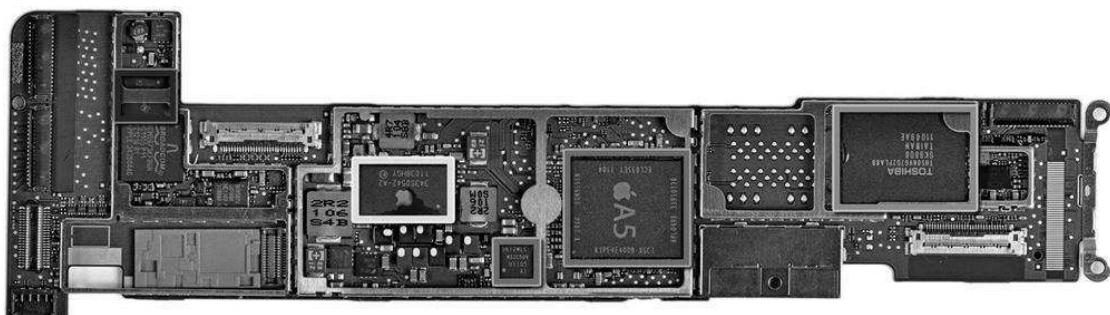
The small rectangles in [Figure 1.8](#) contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The A5 package seen in the middle of [Figure 1.8](#) contains two ARM processors that operate at a clock rate of 1 GHz. The *processor* is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. Occasionally, people call the processor the **CPU**, for the more bureaucratic-sounding **central processor unit**.

## integrated circuit

Also called a **chip**. A device combining dozens to millions of transistors.

## central processor unit (CPU)

Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.



**FIGURE 1.8** The logic board of Apple iPad 2 in Figure 1.7. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. Figure 1.9 shows a photograph of the processor chip inside the A5 package. The similar-sized chip to the left is the 32 GB flash memory chip for non-volatile storage. There is an empty space between the two chips where a second flash chip can be installed to double storage capacity of the iPad. The chips to the right of the A5 include power controller and I/O controller chips. (Courtesy iFixit, [www.ifixit.com](http://www.ifixit.com))

Descending even lower into the hardware, Figure 1.9 reveals details of a microprocessor. The processor logically comprises two main components: datapath and control, the respective brawn and brain of the processor. The **datapath** performs the arithmetic operations, and **control** tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. Chapter 4 explains the datapath and control for a higher-performance design.

## **datapath**

The component of the processor that performs arithmetic operations.

## **control**

The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.



**FIGURE 1.9** The processor integrated circuit inside the A5 package. The size of chip is 12.1 by 10.1 mm, and it was manufactured originally in a 45-nm process (see [Section 1.5](#)). It has two identical ARM processors or cores in the middle left of the chip and a PowerVR *graphics processing unit* (GPU) with four datapaths in the upper left quadrant. To the left and bottom side of the ARM cores are interfaces to main memory (DRAM). (Courtesy Chipworks, [www.chipworks.com](http://www.chipworks.com))

The A5 package in [Figure 1.8](#) also includes two memory chips,

each with 2 gibibits of capacity, thereby supplying 512 MiB. The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. The memory is built from DRAM chips. *DRAM* stands for **dynamic random access memory**. Multiple DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories, such as magnetic tapes, the *RAM* portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read.

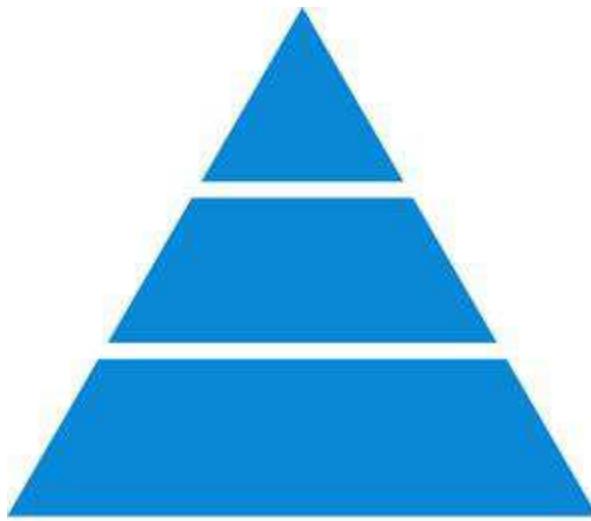
## memory

The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

## dynamic random access memory (DRAM)

Memory built as an integrated circuit; it provides random access to any location. Access times are 50 nanoseconds and cost per gigabyte in 2012 was \$5 to \$10.

Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory. **Cache memory** consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, **static random access memory (SRAM)**. SRAM is faster but less dense, and hence more expensive, than DRAM (see [Chapter 5](#)). SRAM and DRAM are two layers of the **memory hierarchy**.



## HIERARCHY

### cache memory

A small, fast memory that acts as a buffer for a slower, larger memory.

### static random access memory (SRAM)

Also memory built as an integrated circuit, but faster and less dense than DRAM.

As mentioned above, one of the great ideas to improve design is abstraction. One of the most important **abstractions** is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special name: the **instruction set architecture**, or simply **architecture**, of a computer. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the **application binary interface (ABI)**.



## ABSTRACTION

### instruction set architecture

Also called **architecture**. An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

### application binary interface (ABI)

The user portion of the instruction set plus the operating system interfaces used by application programmers. It defines a standard for binary portability across computers.

An instruction set architecture allows computer designers to talk about functions independently from the hardware that performs them. For example, we can talk about the functions of a digital clock (keeping time, displaying the time, setting the alarm) separately from the clock hardware (quartz crystal, LED displays, plastic buttons). Computer designers distinguish architecture from an **implementation** of an architecture along the same lines: an

implementation is hardware that obeys the architecture abstraction. These ideas bring us to another Big Picture.

## The BIG Picture

Both hardware and software consist of hierarchical layers using abstraction, with each lower layer hiding details from the level above. One key interface between the levels of abstraction is the *instruction set architecture*—the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

## A Safe Place for Data

Thus far, we have seen how to input data, compute using the data, and display data. If we were to lose power to the computer, however, everything would be lost because the memory inside the computer is **volatile**—that is, when it loses power, it forgets. In contrast, a DVD disk doesn't forget the movie when you turn off the power to the DVD player, and is therefore a **nonvolatile memory** technology.

### implementation

Hardware that obeys the architecture abstraction.

### volatile memory

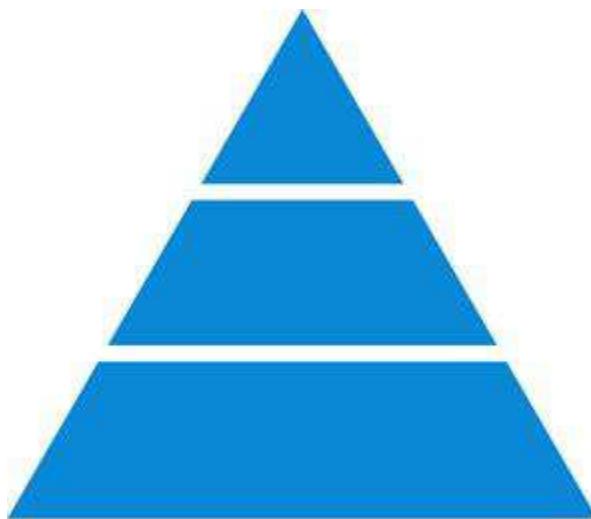
Storage, such as DRAM, that retains data only if it is receiving power.

### nonvolatile memory

A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. A DVD disk is nonvolatile.

To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term **main**

**memory** or **primary memory** is used for the former, and **secondary memory** for the latter. Secondary memory forms the next lower layer of the **memory hierarchy**. DRAMs have dominated main memory since 1975, but **magnetic disks** dominated secondary memory starting even earlier. Because of their size and form factor, personal mobile devices use **flash memory**, a nonvolatile semiconductor memory, instead of disks. [Figure 1.8](#) shows the chip containing the flash memory of the iPad 2. While slower than DRAM, it is much cheaper than DRAM in addition to being nonvolatile. Although costing more per bit than disks, it is smaller, it comes in much smaller capacities, it is more rugged, and it is more power efficient than disks. Hence, flash memory is the standard secondary memory for PMDs. Alas, unlike disks and DRAM, flash memory bits wear out after 100,000 to 1,000,000 writes. Thus, file systems must keep track of the number of writes and have a strategy to avoid wearing out storage, such as by moving popular data. [Chapter 5](#) describes disks and flash memory in more detail.



## HIERARCHY

### main memory

Also called **primary memory**. Memory used to hold programs while they are running; typically consists of DRAM in today's

computers.

## secondary memory

Nonvolatile memory used to store programs and data between runs; typically consists of flash memory in PMDs and magnetic disks in servers.

## magnetic disk

Also called **hard disk**. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material. Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2012 was \$0.05 to \$0.10.

## flash memory

A nonvolatile semi-conductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was \$0.75 to \$1.00.

# Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in [Figure 1.5](#) is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new personal mobile device or server without a network interface would be ridiculed. Networked computers have several major advantages:

- *Communication*: Information is exchanged between computers at high speeds.
- *Resource sharing*: Rather than each computer having its own I/O devices, computers on the network can share I/O devices.
- *Nonlocal access*: By connecting computers over long distances,

users need not be near the computer they are using.

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is *Ethernet*. It can be up to a kilometer long and transfer at up to 40 gigabits per second. Its length and speed make Ethernet useful to connect computers on the same floor of a building; hence, it is an example of what is generically called a **local area network**. Local area networks are interconnected with switches that can also provide routing services and security. **Wide area networks** cross continents and are the backbone of the Internet, which supports the web. They are typically based on optical fibers and are leased from telecommunication companies.

## local area network (LAN)

A network designed to carry data within a geographically confined area, typically within a single building.

## wide area network (WAN)

A network extended over hundreds of kilometers that can span a continent.

Networks have changed the face of computing in the last 30 years, both by becoming much more ubiquitous and by making dramatic increases in performance. In the 1970s, very few individuals had access to electronic mail, the Internet and web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. Local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became considerably cheaper and had a significantly higher capacity. For example, the first standardized local area network technology, developed about 30 years ago, was a version of Ethernet that had a maximum capacity (also called bandwidth) of 10 million bits per second, typically shared by tens of, if not a hundred, computers. Today, local area network technology offers a capacity of from 1 to 40

gigabits per second, usually shared by at most a few computers. Optical communications technology has allowed similar growth in the capacity of wide area networks, from hundreds of kilobits to gigabits and from hundreds of computers connected to a worldwide network to millions of computers connected. This dramatic rise in deployment of networking combined with increases in capacity have made network technology central to the information revolution of the last 30 years.

For the last decade another innovation in networking is reshaping the way computers communicate. Wireless technology is widespread, which enabled the post-PC era. The ability to make a radio in the same low-cost semiconductor technology (CMOS) used for memory and microprocessors enabled a significant improvement in price, leading to an explosion in deployment. Currently available wireless technologies, called by the IEEE standard name 802.11, allow for transmission rates from 1 to nearly 100 million bits per second. Wireless technology is quite a bit different from wire-based networks, since all users in an immediate area share the airwaves.

### Check Yourself

- Semiconductor DRAM memory, flash memory, and disk storage differ significantly. For each technology, list its volatility, approximate relative access time, and approximate relative cost compared to DRAM.

## 1.5 Technologies for Building Processors and Memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. [Figure 1.10](#) shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of

integrated circuits.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

**FIGURE 1.10 Relative performance per unit cost of technologies used in computers over time.** Source:



Computer Museum, Boston, with 2013 extrapolated by the authors. See

[Section 1.12](#).

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. When Gordon Moore predicted the continuous doubling of resources, he was forecasting the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

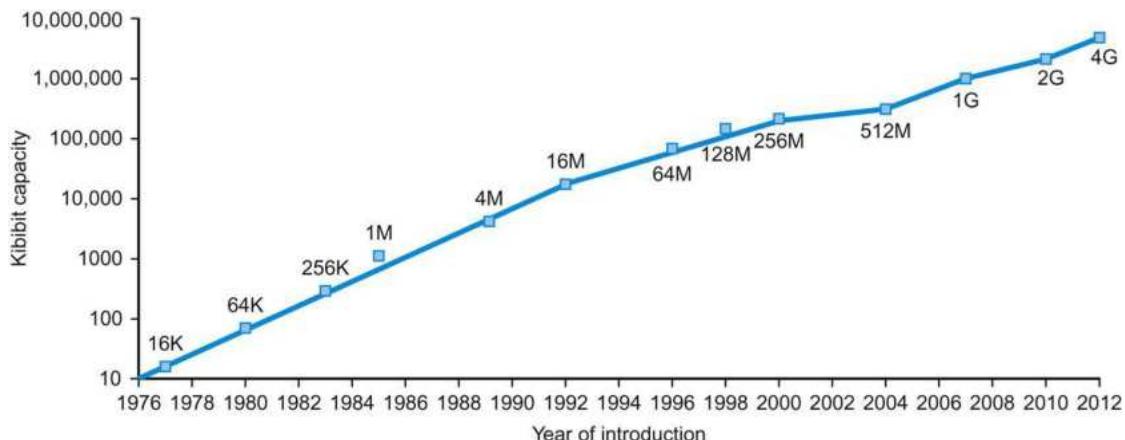
## transistor

An on/off switch controlled by an electric signal.

## very large-scale integrated (VLSI) circuit

A device containing hundreds of thousands to millions of transistors.

This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For 35 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times!



**FIGURE 1.11 Growth of capacity per DRAM chip over time.**

The y-axis is measured in kibibits ( $2^{10}$  bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two to three years.

To understand how to manufacture integrated circuits, we start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)

### silicon

A natural element that is a semiconductor.

### semiconductor

A substance that does not conduct electricity well.

- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under specific conditions (as a switch)

Transistors fall into the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package.

The manufacturing process for integrated circuits is critical to the

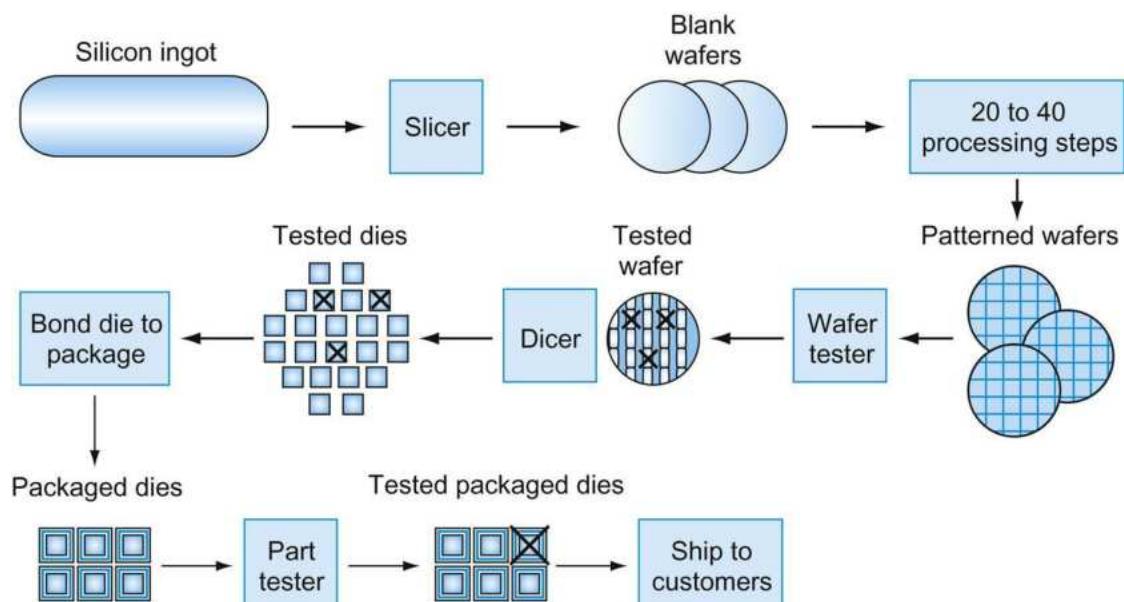
cost of the chips and hence important to computer designers. [Figure 1.12](#) shows that process. The process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 12–24 inches long. An ingot is finely sliced into **wafers** no more than 0.1 inches thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed on each wafer, creating the transistors, conductors, and insulators discussed earlier. Today's integrated circuits contain only one layer of transistors but may have from two to eight levels of metal conductor, separated by layers of insulators.

## silicon crystal ingot

A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

## wafer

A slice from a silicon ingot no more than 0.1 inches thick, used to create chips.



**FIGURE 1.12** The chip manufacturing process.

After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see [Figure 1.13](#)). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Next, the wafers are diced into dies (see

[Figure 1.9](#)). In this figure, one wafer produced 20 dies, of which 17 passed testing. (X means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

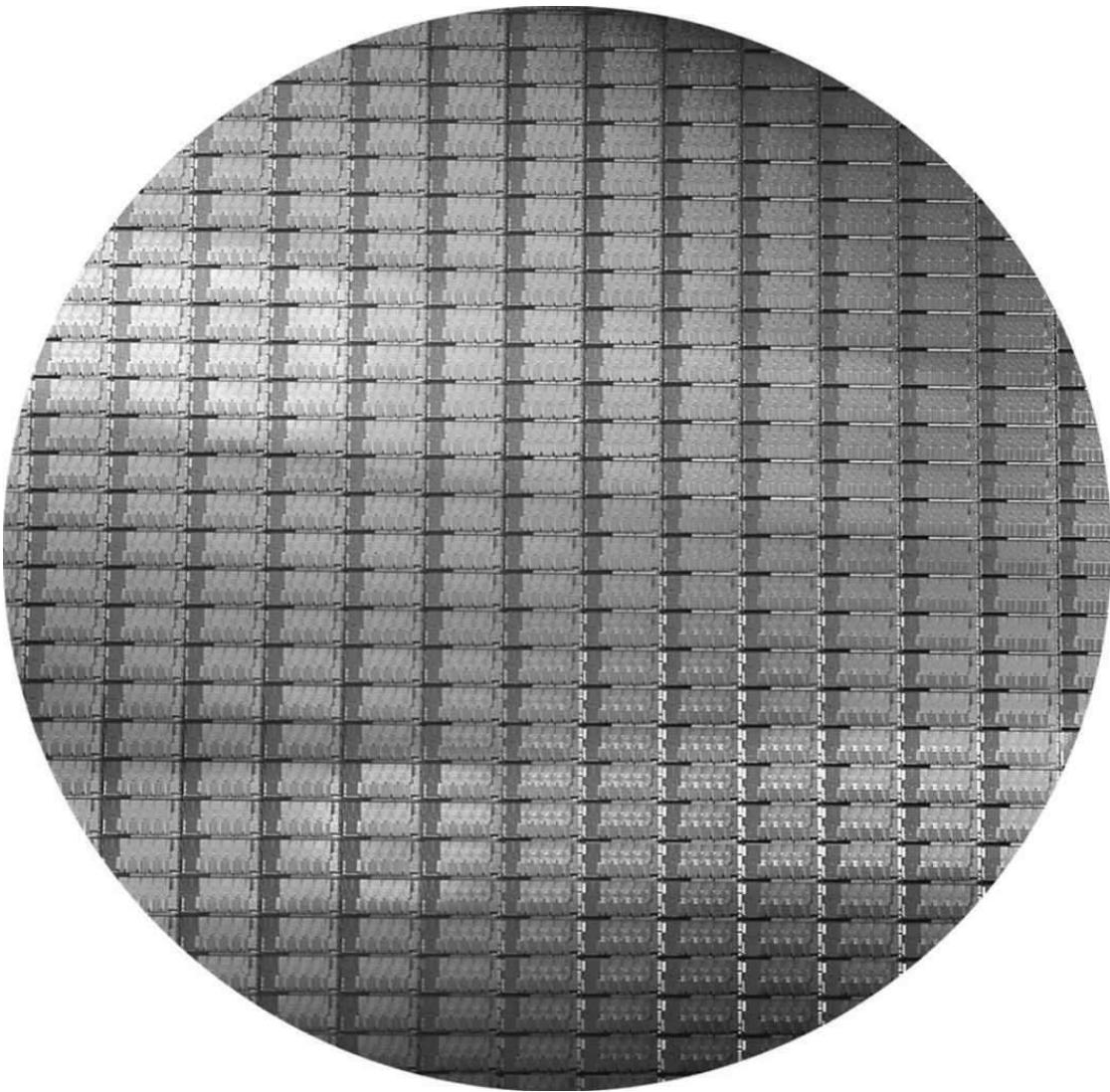
A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components, called **dies** and more informally known as **chips**. [Figure 1.13](#) shows a photograph of a wafer containing microprocessors before they have been diced; earlier, [Figure 1.9](#) shows an individual microprocessor die.

## defect

A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

## die

The individual rectangular sections that are cut from a wafer, more informally known as **chips**.



**FIGURE 1.13 A 12-inch (300 mm) wafer of Intel Core i7 (Courtesy Intel).**

The number of dies on this 300 mm (12 inch) wafer at 100% yield is 280, each 20.7 by 10.5 mm. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon.

This die uses a 32-nanometer technology, which means that the smallest features are approximately 32 nm in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as "drawn" versus the final manufactured size.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as

the percentage of good dies from the total number of dies on the wafer.

## yield

The percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and to the fewer dies that fit on a wafer. To reduce the cost, using the next generation process shrinks a large die as it uses smaller sizes for both transistors and wires. This improves the yield and the die count per wafer. A 32-nanometer (nm) process was typical in 2012, which means essentially that the smallest feature size on the die is 32 nm.

Once you've found good dies, they are connected to the input/output pins of a package, using a process called *bonding*. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

## Elaboration

The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$
$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$
$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

The first equation is straightforward to derive. The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies (see [Figure 1.13](#)). The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in the die area.

## Check Yourself

A key factor in determining the cost of an integrated circuit is volume. Which of the following are reasons why a chip made in high volume should cost less?

1. With high volumes, the manufacturing process can be tuned to a particular design, increasing the yield.
2. It is less work to design a high-volume part than a low-volume part.
3. The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
4. Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
5. High-volume parts usually have smaller die sizes than low-volume parts and therefore, have higher yield per wafer.

## 1.6 Performance

Assessing the performance of computers can be quite challenging. The scale and intricacy of modern software systems, together with the wide range of performance improvement techniques employed by hardware designers, have made performance assessment much more difficult.

When trying to choose among different computers, performance is an important attribute. Accurately measuring and comparing different computers is critical to purchasers and therefore, to designers. The people selling computers know this as well. Often, salespeople would like you to see their computer in the best possible light, whether or not this light accurately reflects the needs of the purchaser's application. Hence, understanding how best to measure performance and the limitations of those measurements is important in selecting a computer.

The rest of this section describes different ways in which performance can be determined; then, we describe the metrics for measuring performance from the viewpoint of both a computer user and a designer. We also look at how these metrics are related and present the classical processor performance equation, which we will use throughout the text.

## Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. [Figure 1.14](#) lists some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed was the Concorde (retired from service in 2003), the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

**FIGURE 1.14** The capacity, range, and speed for a number of commercial airplanes.

The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

Let's suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you were interested in transporting 450 passengers from one point to another, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several distinct ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most

jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as **execution time**. Datacenter managers often care about increasing **throughput** or **bandwidth**—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.

## response time

Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

## throughput

Also called **bandwidth**. Another measure of performance, it is the number of tasks completed per unit time.

# Throughput and Response Time

## Example

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the web

## Answer

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases.

If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in

the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned}\text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X\end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is  $n$  times faster than Y”—or equivalently “X is  $n$  times as fast as Y”—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is  $n$  times as fast as Y, then the execution time on Y is  $n$  times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

## Relative Performance

### Example

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

### Answer

We know that A is  $n$  times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology *as fast as* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say “improve performance” or “improve execution time”

when we mean “increase performance” and “decrease execution time.”

## Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. However, time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, *input/output* (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time over which the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to do accurately, because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences between operating systems.

### CPU execution time

Also called **CPU time**. The actual time the CPU spends computing for a specific task.

### user CPU time

The CPU time spent in a program itself.

## system CPU time

The CPU time spent in the operating system performing tasks on behalf of the program.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system and *CPU performance* to refer to user CPU time. We will focus on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or CPU time measurements.

## Understanding Program Performance

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software. Total elapsed time measured by a wall clock is the measurement of interest. In some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to find performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or **ticks**, **clock ticks**, **clock**

**periods, clocks, cycles**). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

## clock cycle

Also called **tick**, **clock tick**, **clock period**, **clock**, or **cycle**. The time for one clock period, usually of the processor clock, which runs at a constant rate.

## clock period

The length of each clock cycle.

## Check Yourself

1. Suppose we know that an application that uses both personal mobile devices and the Cloud is limited by network performance. For the following changes, state whether only the throughput improves, both response time and throughput improve, or neither improves.
  - a. An extra network channel is added between the PMD and the Cloud, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).
  - b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
  - c. More memory is added to the computer.
2. Computer C's performance is four times as fast as the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

## CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could

determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock cycle time}}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clockrate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle. As we will see in later chapters, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles may also increase the clock cycle time.

## Improving Performance

### Example

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

### Answer

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

## Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different instructions may take different amounts of time depending on what they do, CPI is an average of

all the instructions executed in the program. CPI provides one way of comparing two different implementations of the identical instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

## clock cycles per instruction (CPI)

Average number of clock cycles per instruction for a program or program fragment.

## Using the Performance Equation

### Example

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

### Answer

We know that each computer executes the same number of instructions for the program; let's call this number  $I$ . First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time} \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the

ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

## The Classic CPU Performance Equation

We can now write this basic performance equation in terms of **instruction count** (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

### instruction count

The number of instructions executed by the program.

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

## Comparing Code Segments

### Example

A compiler designer is trying to decide between two code sequences for a computer. The hardware designers have supplied the following facts:

CPI for each instruction class			
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

### Answer

Sequence 1 executes  $2 + 1 + 2 = 5$  instructions. Sequence 2 executes  $4 + 1 + 1 = 6$  instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

## The BIG Picture

Figure 1.15 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

$$\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

**FIGURE 1.15 The basic components of performance and how each is measured.**

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count. Similarly, because CPI depends on the type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

How can we determine the value of these factors in the performance equation? We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a computer. The instruction count and CPI can be more difficult to obtain. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.

We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture. Alternatively, we can use hardware counters, which are included in most processors, to record a variety of measurements, including the number of instructions executed, the average CPI, and often, the sources of performance loss. Since the instruction count depends on the architecture, but not on the exact implementation, we can measure the instruction count without knowing all the details of the implementation. The CPI, however, depends on a wide variety of design details in the computer, including both the memory system and the processor structure (as we will see in [Chapter 4](#) and [Chapter 5](#)), as well as on the mix of instruction types executed in an application. Thus, CPI varies by application, as well as among implementations with the same instruction set.

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by

**instruction mix**, both instruction count and CPI must be compared, even if clock rates are equal. Several exercises at the end of this chapter ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In



**Section 1.10**, we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

## instruction mix

A measure of the dynamic frequency of instructions across one or many programs.

## Understanding Program Performance

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in varied ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

## Elaboration

Although you might expect that the minimum CPI is 1.0, as we'll

see in [Chapter 4](#), some processors fetch and execute multiple instructions per clock cycle. To reflect that approach, some designers invert CPI to talk about *IPC*, or *instructions per clock cycle*. If a processor executes on average two instructions per clock cycle, then it has an IPC of 2 and hence a CPI of 0.5.

## Elaboration

Although clock cycle time has traditionally been fixed, to save energy or temporarily boost performance, today's processors can vary their clock rates, so we would need to use the *average* clock rate for a program. For example, the Intel Core i7 will temporarily increase clock rate by about 10% until the chip gets too warm. Intel calls this *Turbo mode*.

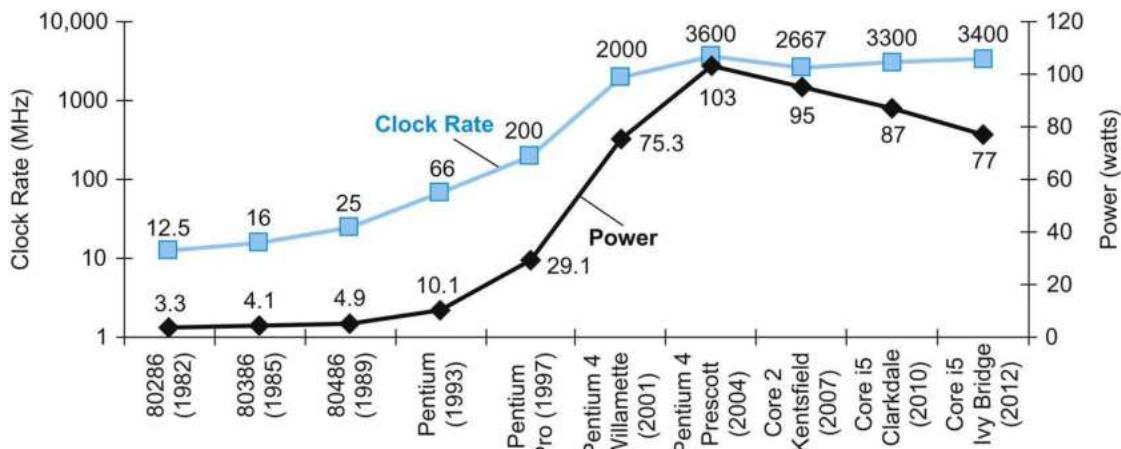
## Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

- a.  $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$
- b.  $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$
- c.  $\frac{1.5 \times 1.1}{0.6} = 27.5 \text{ sec}$

## 1.7 The Power Wall

[Figure 1.16](#) shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 years. Both clock rate and power increased rapidly for decades and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.



**FIGURE 1.16 Clock rate and power for Intel x86 microprocessors over eight generations and 30 years.**

The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

Although power provides a limit to what we can cool, in the post-PC era the really valuable resource is energy. Battery life can trump performance in the personal mobile device, and the architects of warehouse scale computers try to reduce the costs of powering and cooling 100,000 servers as the costs are high at this scale. Just as measuring time in seconds is a safer evaluation of program performance than a rate like MIPS (see [Section 1.10](#)), the energy metric joules is a better measure than a power rate like watts, which is just joules/second.

The dominant technology for integrated circuits is called CMOS (*complementary metal oxide semiconductor*). For CMOS, the primary source of energy consumption is so-called dynamic energy—that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied:

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

This equation is the energy of a pulse during the logic transition of  $0 \rightarrow 1 \rightarrow 0$  or  $1 \rightarrow 0 \rightarrow 1$ . The energy of a single transition is then

$$Energy \propto 1/2 \times Capacitive\ load \times Voltage^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors.

With regard to [Figure 1.16](#), how could clock rates grow by a factor of 1000 while power increased by only a factor of 30? Energy and thus power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. Typically, the voltage was reduced about 15% per generation. In 20 years, voltages have gone from 5 V to 1 V, which is why the increase in power is only 30 times.

## Relative Power

### Example

Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it can adjust voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

### Answer

$$\frac{Power_{new}}{Power_{old}} = \frac{(Capacitive\ load \times 0.85) \times (Voltage \times 0.85)^2 \times (Frequency\ switched \times 0.85)}{Capacitive\ load \times Voltage^2 \times Frequency\ switched}$$

Thus the power ratio is

$$0.85^4 = 0.52$$

Hence, the new processor uses about half the power of the old processor.

The modern problem is that further lowering of the voltage appears to make the transistors too leaky, like water faucets that cannot be completely shut off. Even today about 40% of the power consumption in server chips is due to leakage. If transistors started leaking more, the whole process could become unwieldy.

To try to address the power problem, designers have already attached large devices to increase cooling, and they turn off parts of the chip that are not used in a given clock cycle. Although there are many more expensive ways to cool chips and thereby raise their power to, say, 300 watts, these techniques are generally too costly for personal computers and even servers, not to mention personal mobile devices.

Since computer designers slammed into a power wall, they needed a new way forward. They chose a different path from the way they designed microprocessors for their first 30 years.

## Elaboration

Although dynamic energy is the primary source of energy consumption in CMOS, static energy consumption occurs because of leakage current that flows even when a transistor is off. In servers, leakage is typically responsible for 40% of the energy consumption. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off. A variety of design techniques and technology innovations are being deployed to control leakage, but it's hard to lower voltage further.

## Elaboration

Power is a challenge for integrated circuits for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of chip interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. Server chips can burn more than 100 watts, and cooling the chip and the surrounding system is a major expense in warehouse scale

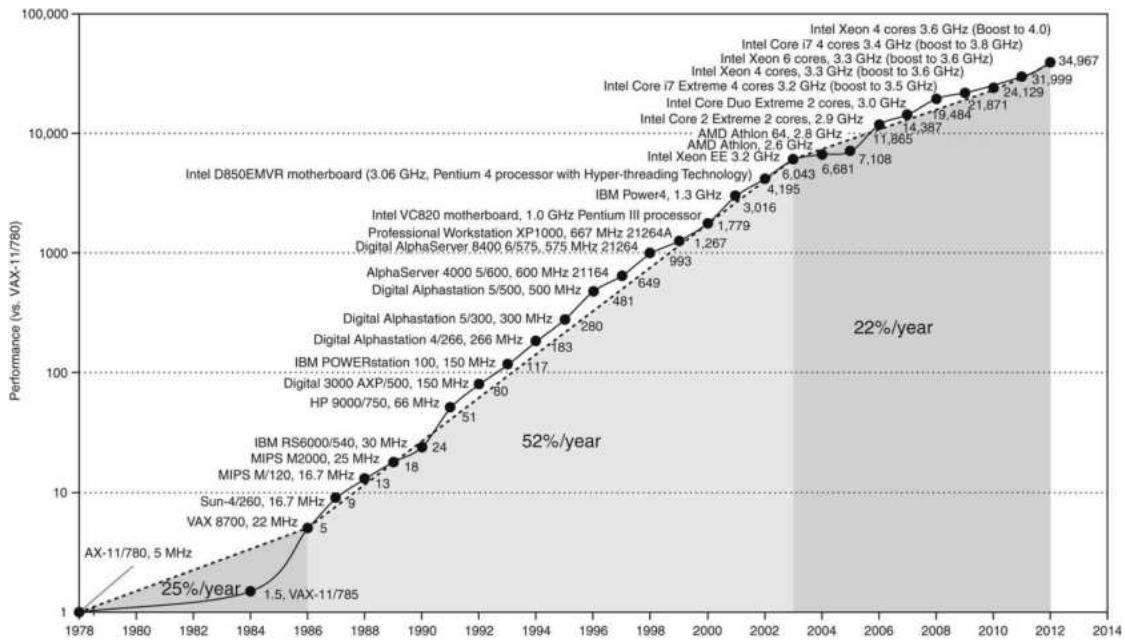
computers (see [Chapter 6](#)).

## 1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

The power limit has forced a dramatic change in the design of microprocessors. [Figure 1.17](#) shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.

*Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge.*

*Brian Hayes, Computing in a Parallel Universe, 2007.*



**FIGURE 1.17 Growth in processor performance since the mid-1980s.**

This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see [Section 1.10](#)). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven larger in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.

Rather than continuing to decrease the response time of one program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as “cores,” and such microprocessors are generically called multicore microprocessors. Hence, a “quadcore” microprocessor is a chip that contains four processors or four cores.

In the past, programmers could rely on innovations in hardware,

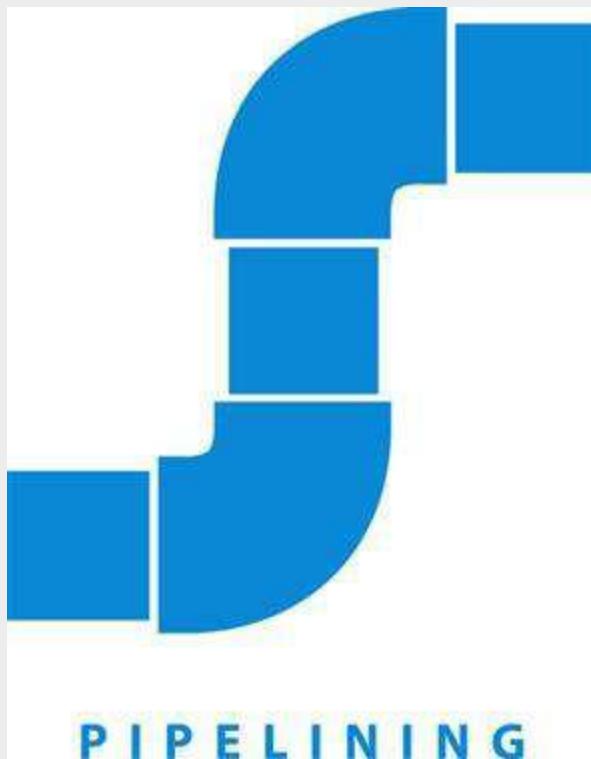
architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve the performance of their code as the number of cores increases.

To reinforce how the software and hardware systems work together, we use a special section, *Hardware/Software Interface*, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

## Hardware/Software Interface

**Parallelism** has always been crucial to performance in computing, but it was often hidden. Chapter 4 will explain **pipelining**, an elegant technique that runs programs faster by overlapping the execution of instructions. This optimization is one example of *instruction-level parallelism*, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.





Forcing programmers to be aware of the parallel hardware and to rewrite their programs to be parallel had been the “third rail” of computer architecture, for companies in the past that depended on

such a change in behavior failed (see  [Section 6.15](#)). From this historical perspective, it’s startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

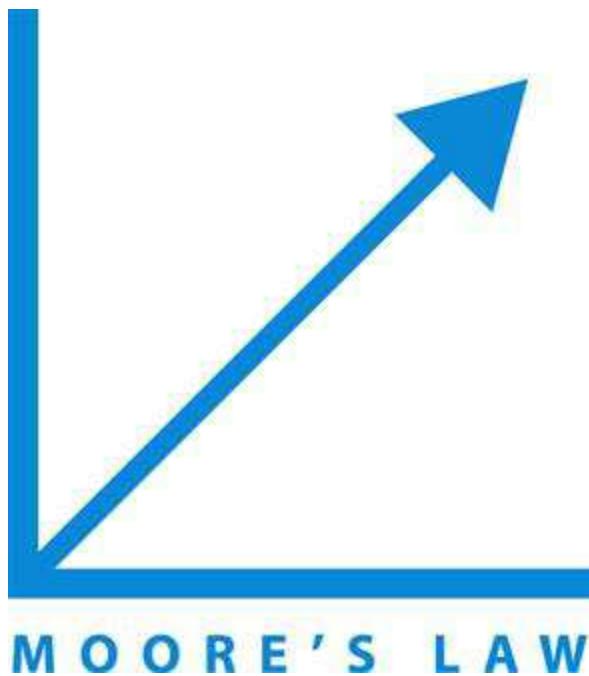
Why has it been so hard for programmers to write explicitly parallel programs? The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the program need to be correct, solve an important problem, and provide a useful interface to the people or other programs that invoke it; the program must also be fast. Otherwise, if you don’t need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn’t fritter away the potential performance benefits of parallelism.

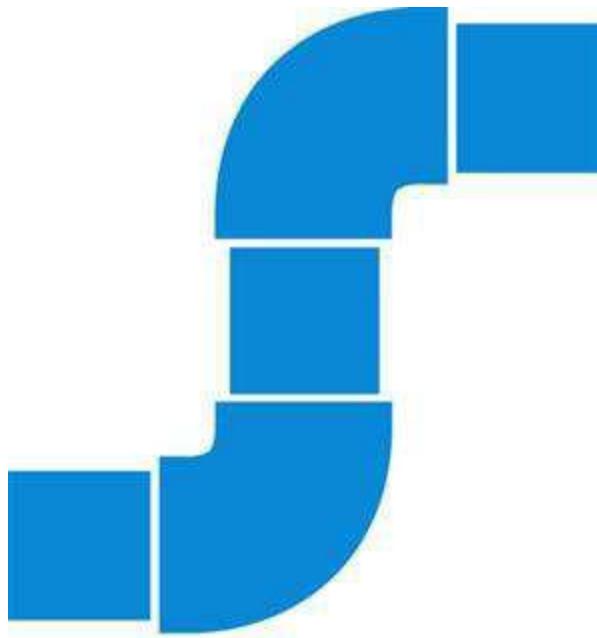
As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the sub-tasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the load* evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

To reflect this sea change in the industry, the next five chapters in this edition of the book each has a section on the implications of the parallel revolution to that chapter:

- *Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization.* Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism.* Perhaps the simplest form of parallelism to build involves computing on elements in parallel, such as when multiplying two vectors. Subword parallelism takes advantage of the resources supplied by **Moore's Law** to provide wider arithmetic units that can operate on many operands simultaneously.



- *Chapter 4, Section 4.10: Parallelism via Instructions.* Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism, initially via **pipelining**. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions concurrently and guessing on the outcomes of decisions, and executing instructions speculatively using **prediction**.

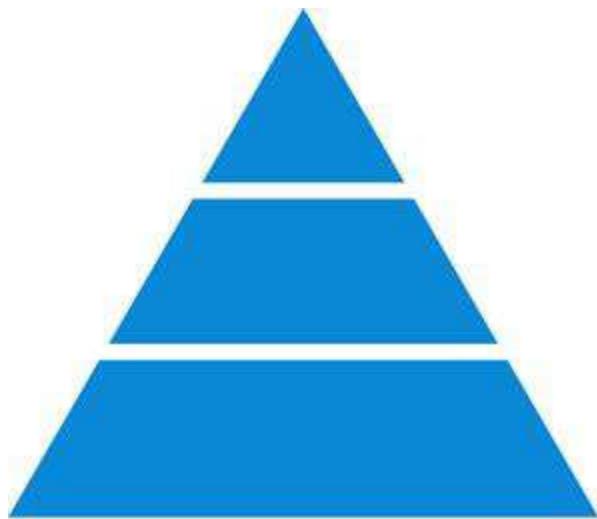


## PIPELINING

- *Chapter 5, Section 5.10: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.



## PREDICTION



## HIERARCHY

- *Chapter 5, Section 5.11: Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks.* This section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of *Redundant Arrays of*

*Inexpensive Disks* (RAID). The real popularity of RAID proved to be the much greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and dependability between the various RAID levels.

In addition to these sections, there is a full chapter on parallel processing. [Chapter 6](#) goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors; and, finally, describes and evaluates four examples of multicore microprocessors using this model.

As mentioned above, [Chapters 3 to 6](#) use matrix vector multiply as a running example to show how each type of parallelism can significantly increase performance.



[Appendix B](#) describes an increasingly popular hardware component that is included with desktop computers, the *graphics processing unit* (GPU). Invented to accelerate graphics, GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs rely on **parallelism**.



[Appendix B](#) describes the NVIDIA GPU and highlights parts of its parallel programming environment.



## PARALLELISM

*I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.*

*J. Presper Eckert, coinventor of ENIAC, speaking in 1991*

## 1.9 Real Stuff: Benchmarking the Intel Core i7

Each chapter has a section entitled “Real Stuff” that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first “Real Stuff” section, we look at how integrated circuits are manufactured and how performance and power are measured, with the Intel Core i7 as the example.

### SPEC CPU Benchmark

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a **workload**. To evaluate two

computer systems, a user would simply compare the execution time of the workload on the two computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user's workload. This alternative is usually followed by evaluating the computer using a set of **benchmarks**—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload. As we noted above, to make the **common case fast**, you first need to know accurately which case is common, so benchmarks play a critical role in computer architecture.



## COMMON CASE FAST

### workload

A set of programs run on a computer that is either the actual collection of applications run by a user or constructed from real programs to approximate such a mix. A typical workload specifies both the programs and the relative frequencies.

### benchmark

A program selected for use in comparing computer performance.

SPEC (*System Performance Evaluation Cooperative*) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. In 1989,

SPEC originally created a benchmark set focusing on processor performance (now called SPEC89), which has evolved through five generations. The latest is SPEC CPU2006, which consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006). The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

[Figure 1.18](#) describes the SPEC integer benchmarks and their execution time on the Intel Core i7 and shows the factors that explain execution time: instruction count, CPI, and clock cycle time. Note that CPI varies by more than a factor of 5.

Description	Name	Instruction Count $\times 10^9$	CPI	Clock cycle time (seconds $\times 10^{-9}$ )	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	-	-	-	-	-	-	25.7

**FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66 GHz Intel Core i7 920.**

As the equation on page 36 explains, execution time is the product of the three factors in this table: instruction count in billions, *clocks per instruction* (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios.

To simplify the marketing of computers, SPEC decided to report a single number summarizing all 12 integer benchmarks. Dividing the execution time of a reference processor by the execution time of

the evaluated computer normalizes the execution time measurements; this normalization yields a measure, called the *SPECratio*, which has the advantage that bigger numeric results indicate faster performance. That is, the SPECratio is the inverse of execution time. A CINT2006 or CFP2006 summary measurement is obtained by taking the geometric mean of the SPECratios.

## Elaboration

When comparing two computers using SPECratios, apply the geometric mean so that it gives the same relative answer no matter what computer is used to normalize the results. If we averaged the normalized execution time values with an arithmetic mean, the results would vary depending on the computer we choose as the reference.

The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$

where  $\text{Execution time ratio}_i$  is the execution time, normalized to the reference computer, for the  $i$ th program of a total of  $n$  in the workload, and

$$\prod_{i=1}^n a_i \text{ means the product } a_1 \times a_2 \times \dots \times a_n$$

## SPEC Power Benchmark

Given the increasing importance of energy and power, SPEC added a benchmark to measure power. It reports power consumption of servers at different workload levels, divided into 10% increments, over a period of time. [Figure 1.19](#) shows the results for a server using Intel Nehalem processors similar to the above.

Target Load %	Performance (ssj_ops)	Average Power (watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1922
$\sum \text{ssj\_ops} / \sum \text{power} =$		2490

**FIGURE 1.19** SPECpower\_ssj2008 running on a dual socket 2.66 GHz Intel Xeon X5650 with 16 GB of DRAM and one 100 GB SSD disk.

SPECpower started with another SPEC benchmark for Java business applications (SPECJBB2005), which exercises the processors, caches, and main memory as well as the Java virtual machine, compiler, garbage collector, and pieces of the operating system. Performance is measured in throughput, and the units are business operations per second. Once again, to simplify the marketing of computers, SPEC boils these numbers down to one number, called “overall ssj\_ops per watt.” The formula for this single summarizing metric is

$$\text{overall ssj\_ops per watt} = \left( \sum_{i=0}^{10} \text{ssj\_ops}_i \right) / \left( \sum_{i=0}^{10} \text{power}_i \right)$$

where  $\text{ssj\_ops}_i$  is performance at each 10% increment and  $\text{power}_i$  is power consumed at each performance level.

## 1.10 Fallacies and Pitfalls

*Science must begin with myths, and the criticism of myths.*

*Sir Karl Popper, The Philosophy of Science, 1957*

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call them *fallacies*. When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*, or easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these mistakes in the computers you may design or use.

Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that traps many designers and reveals an important relationship in computer design.

*Pitfall: Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of the improvement.*

The great idea of making the **common case fast** has a demoralizing corollary that has plagued designers of both hardware and software. It reminds us that the opportunity for improvement is affected by how much time the event consumes.



## COMMON CASE FAST

A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run

five times faster?

The execution time of the program after making the improvement is given by the following simple equation known as **Amdahl's Law**:

$$\frac{\text{Execution time after improvement}}{\text{Execution time affected by improvement} + \text{Execution time unaffected}} = \frac{1}{\text{Amount of improvement}}$$

## Amdahl's Law

A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. It is a quantitative version of the law of diminishing returns.

For this problem:

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$
$$0 = \frac{80 \text{ seconds}}{n}$$

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. In everyday life this concept also yields what we call the law of diminishing returns.

We can use Amdahl's Law to estimate performance improvements when we know the time consumed for some

function and its potential speedup. Amdahl's Law, together with the CPU performance equation, is a handy tool for evaluating possible enhancements. Amdahl's Law is explored in more detail in the exercises.

Amdahl's Law is also used to argue for practical limits to the number of parallel processors. We examine this argument in the Fallacies and Pitfalls section of [Chapter 6](#).

*Fallacy: Computers at low utilization use little power.*

Power efficiency matters at low utilizations because server workloads vary. Utilization of servers in Google's warehouse scale computer, for example, is between 10% and 50% most of the time and at 100% less than 1% of the time. Even given 5 years to learn how to run the SPECpower benchmark well, the specially configured computer with the best results in 2012 still uses 33% of the peak power at 10% of the load. Systems in the field that are not configured for the SPECpower benchmark are surely worse.

Since servers' workloads vary but use a large fraction of peak power, Luiz Barroso and Urs Hözle [2007] argue that we should redesign hardware to achieve "energy-proportional computing." If future servers used, say, 10% of peak power at 10% workload, we could reduce the electricity bill of datacenters and become good corporate citizens in an era of increasing concern about CO<sub>2</sub> emissions.

*Fallacy: Designing for performance and designing for energy efficiency are unrelated goals.*

Since energy is power over time, it is often the case that hardware or software optimizations that take less time save energy overall even if the optimization takes a bit more energy when it is used. One reason is that all the rest of the computer is consuming energy while the program is running, so even if the optimized portion uses a little more energy, the reduced time can save the energy of the whole system.

*Pitfall: Using a subset of the performance equation as a performance metric.*

We have already warned about the danger of predicting performance based on simply one of the clock rate, instruction count, or CPI. Another common mistake is to use only two of the three factors to compare performance. Although using two of the three factors may be valid in a limited context, the concept is also easily misused. Indeed, nearly all proposed alternatives to the use of time as the performance metric have led eventually to misleading claims, distorted results, or incorrect interpretations.

One alternative to time is **MIPS (million instructions per second)**. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

Since MIPS is an instruction execution rate, MIPS specifies performance inversely to execution time; faster computers have a higher MIPS rating. The good news about MIPS is that it is easy to understand, and quicker computers mean bigger MIPS, which matches intuition.

### million instructions per second (MIPS)

A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and  $10^6$ .

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating. For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\frac{\text{Instruction count}}{\text{Clock rate} \times \text{CPI}} \times 10^6}{\text{Clock rate}} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

The CPI varied by a factor of 5 for SPEC CPU2006 on an Intel Core i7 computer in [Figure 1.18](#), so MIPS does as well. Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance!

Consider the following performance measurements for a program:

### Check Yourself

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

- a. Which computer has the higher MIPS rating?
- b. Which computer is faster?

## 1.11 Concluding Remarks

Although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's a safe bet that they will be much better than they are today. To participate in these advances, computer designers and programmers must understand a wider variety of issues.

*Where ... the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have 1,000 vacuum tubes and perhaps weigh just 1½ tons.*

*Popular Mechanics, March 1949*

Both hardware and software designers construct computer systems in hierarchical layers, with each lower layer hiding details from the level above. This great idea of **abstraction** is fundamental to understanding today's computer systems, but it does not mean that designers can limit themselves to knowing a single abstraction.

Perhaps the most important example of abstraction is the interface between hardware and low-level software, called the *instruction set architecture*. Maintaining the instruction set architecture as a constant enables many implementations of that architecture—presumably varying in cost and performance—to run identical software. On the downside, the architecture may preclude introducing innovations that require the interface to change.



## ABSTRACTION

There is a reliable method of determining and reporting performance by using the execution time of real programs as the metric. This execution time is related to other important measurements we can make by the following equation:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

We will use this equation and its constituent factors many times. Remember, though, that individually the factors do not determine performance: only the product, which equals execution time, is a

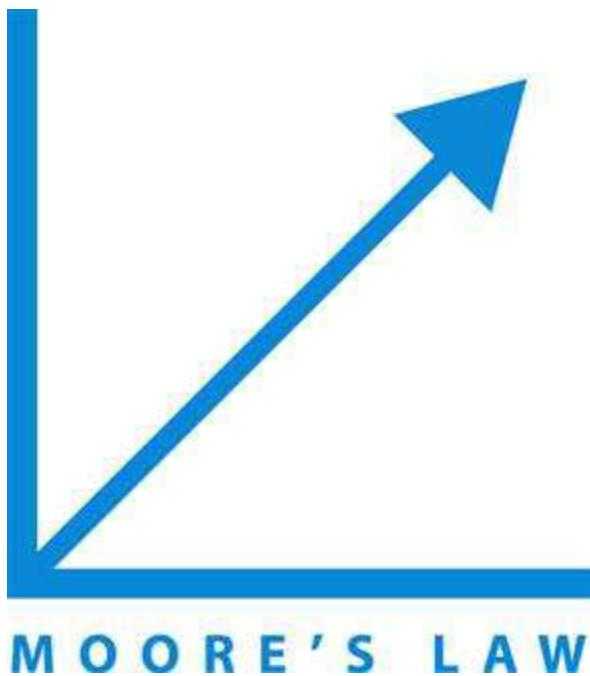
reliable measure of performance.

## The BIG Picture

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric that is sound in a limited context is extended and used beyond that context or without the additional clarification needed to make it valid.

The key hardware technology for modern processors is silicon. Equal in importance to an understanding of integrated circuit technology is an understanding of the expected rates of technological change, as predicted by **Moore's Law**. While silicon fuels the rapid advance of hardware, new ideas in the organization of computers have improved price/performance. Two of the key ideas are exploiting parallelism in the program, normally today via multiple processors, and exploiting locality of accesses to a **memory hierarchy**, typically via caches.





Energy efficiency has replaced die area as the most critical resource of microprocessor design. Conserving power while trying to increase performance has forced the hardware industry to switch to multicore microprocessors, thereby requiring the software industry to switch to programming parallel hardware. **Parallelism** is now required for performance.



## P A R A L L E L I S M

Computer designs have always been measured by cost and

performance, as well as other important factors such as energy, dependability, cost of ownership, and scalability. Although this chapter has focused on cost, performance, and energy, the best designs will strike the appropriate balance for a given market among all the factors.

## Road Map for This Book

At the bottom of these abstractions is the five classic components of a computer: datapath, control, memory, input, and output (refer to [Figure 1.5](#)). These five components also serve as the framework for the rest of the chapters in this book:

- *Datapath*: [Chapter 3](#), [Chapter 4](#), [Chapter 6](#), and  [Appendix B](#)
- *Control*: [Chapter 4](#), [Chapter 6](#), and  [Appendix B](#)
- *Memory*: [Chapter 5](#)
- *Input*: [Chapters 5 and 6](#)
- *Output*: [Chapters 5 and 6](#)

As mentioned above, [Chapter 4](#) describes how processors exploit implicit parallelism, [Chapter 6](#) describes the explicitly parallel multicore microprocessors that are at the heart of the parallel

 [Appendix B](#) describes the highly parallel graphics processor chip. [Chapter 5](#) describes how a memory hierarchy exploits locality. [Chapter 2](#) describes instruction sets—the interface between compilers and the computer—and emphasizes the role of compilers and programming languages in using the features of the instruction set. [Chapter 3](#) describes how computers handle arithmetic data. [Appendix A](#) introduces logic design.



## Historical Perspective and Further Reading

*An active field of science is like an immense anthill; the individual almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.*

*Lewis Thomas, “Natural Science,” in *The Lives of a Cell*, 1974*

For each chapter in the text, a section devoted to a historical perspective can be found online on a site that accompanies this book. We may trace the development of an idea through a series of computers or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented in this opening chapter. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By studying the past, you may be better able to understand the forces that will shape computing in the future. Each Historical Perspective section online ends with suggestions for further reading, which are also collected separately online under the section “**Further**

 **Reading.**” The rest of **Section 1.12** is found online.

## 1.12 Historical Perspective and Further Reading

*An active field of science is like an immense anthill; the individual*

*almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.*

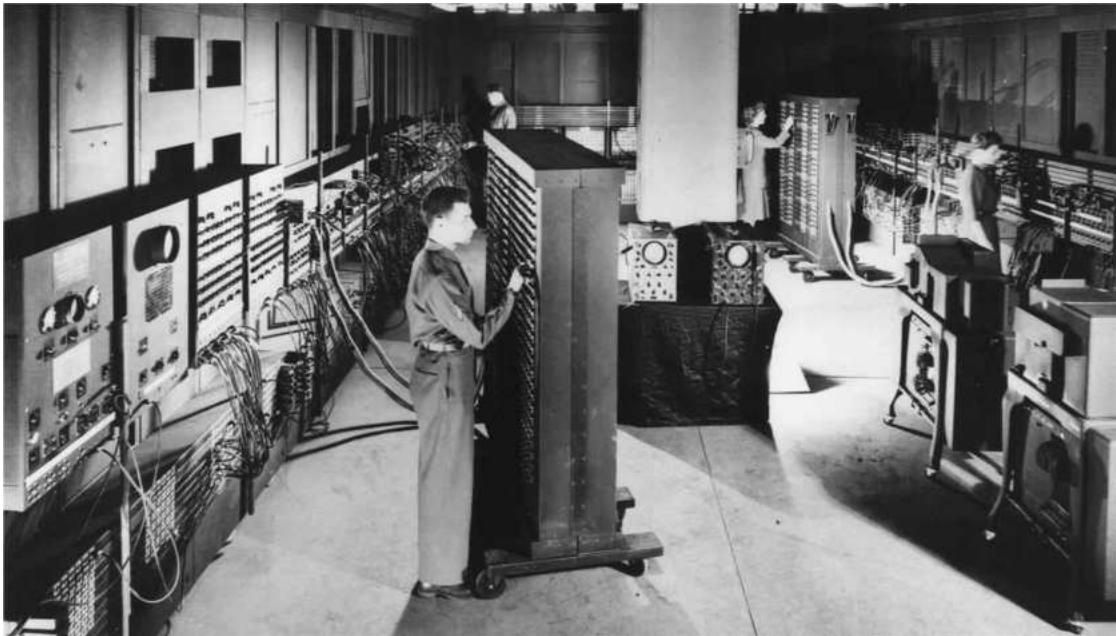
*Lewis Thomas, "Natural Science," in *The Lives of a Cell*, 1974*

For each chapter in the text, a section devoted to a historical perspective can be found online. We may trace the development of an idea through a series of machines or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented therein. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By learning the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspective section ends with suggestions for additional reading, which are also collected separately in the online section "Further Reading."

## The First Electronic Computers

J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania built what is widely accepted to be the world's first operational electronic, general-purpose computer. This machine, called ENIAC (*Electronic Numerical Integrator and Calculator*), was funded by the United States Army and started working during World War II but was not publicly disclosed until 1946. ENIAC was a general-purpose machine used for computing artillery-firing tables. [Figure e1.12.1](#) shows the U-shaped computer, which was 80 feet long by 8.5 feet high and several feet wide. Each of the 20 10-digit registers was 2 feet long. In total, ENIAC used 18,000 vacuum tubes.



**FIGURE E1.12.1 ENIAC, the world's first general-purpose electronic computer.**

In size, ENIAC was two orders of magnitude bigger than machines built today, yet it was more than eight orders of magnitude slower, performing 1900 additions per second. ENIAC provided conditional jumps and was programmable, clearly distinguishing it from earlier calculators. Programming was done manually by plugging cables and setting switches, and data were entered on punched cards. Programming for typical calculations required from half an hour to a whole day. ENIAC was a general-purpose machine, limited primarily by a small amount of storage and tedious programming.

In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; von Neumann helped crystallize the ideas and wrote a memo proposing a stored-program computer called EDVAC (*Electronic Discrete Variable Automatic Computer*). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term *von Neumann computer*. Several early pioneers in the computer field believe that this term gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. For this reason, the term does not appear elsewhere in this book or

in the online sections.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC (*Electronic Delay Storage Automatic Calculator*). EDSAC started working in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, 1985]. (A small prototype called the Mark-I, built at the University of Manchester in 1948, might be called the first operational stored-program machine.) [Section 2.5](#) explains the stored-program concept.

In 1947, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School demanded that the patent be turned over to the university, which may have helped Eckert and Mauchly conclude that they should leave. Their departure crippled the EDVAC project, delaying completion until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study (IAS) at Princeton in 1946. Together with Arthur Burks, they issued a report based on the memo written earlier [Burks et al., 1946]. The paper was incredible for the period; reading it today, you would never guess this landmark paper was written more than 50 years ago, because it discusses most of the architectural concepts seen in modern computers. This paper led to the IAS machine built by Julian Bigelow. It had a total of 1024 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports, and encouraged visitors. These reports and visitors inspired the development of a number of new computers.

Recently, there has been some controversy about the work of John Atanasoff, who built a small-scale electronic computer in the early 1940s. His machine, designed at Iowa State University, was a special-purpose computer that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC. The presence of the Atanasoff machine, together with delays in filing the ENIAC patents (the work was classified and patents could not be filed until after the war) and the distribution of von Neumann's EDVAC paper, was used to break the Eckert-Mauchly patent. Though controversy still rages over Atanasoff's role, Eckert and

Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern, 1980].

Another pioneering computer that deserves credit was a special-purpose machine built by Konrad Zuse in Germany in the late 1930s and early 1940s. Although Zuse had the design for a programmable computer ready, the German government decided not to fund scientific investigations taking more than 2 years because the bureaucrats expected the war would be won by that deadline.

Across the English Channel, during World War II special-purpose electronic computers were built to decrypt intercepted German messages. A team at Bletchley Park, including Alan Turing, built the Colossus in 1943. The machines were kept secret until 1970; after the war, the group had little impact on commercial British computers.

While work on ENIAC went forward, Howard Aiken was building an electro-mechanical computer called the Mark-I at Harvard (a name that Manchester later adopted for its machine). He followed the Mark-I with a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. In contrast to earlier machines like EDSAC, which used a single memory for instructions and data, the Mark-III and Mark-IV had separate memories for instructions and data. The machines were regarded as reactionary by the advocates of stored-program computers; the term *Harvard architecture* was coined to describe machines with distinct memories. Paying respect to history, this term is used today in a different sense to describe machines with a single main memory but with separate caches for instructions and data.

The Whirlwind project was begun at MIT in 1947 and was aimed at applications in real-time radar signal processing. Although it led to several inventions, its most important innovation was magnetic core memory. Whirlwind had 2048 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

## Commercial Developments

In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the BINAC, was built

for Northrop and was shown in August 1949. After some financial difficulties, their firm was acquired by Remington-Rand, where they built the UNIVAC I (Universal Automatic Computer), designed to be sold as a general-purpose computer ([Figure e1.12.2](#)). Originally delivered in June 1951, UNIVAC I sold for about \$1 million and was the first successful commercial computer—48 systems were built! This early machine, along with many other fascinating pieces of computer lore, may be seen at the Computer History Museum in Mountain View, California.



**FIGURE E1.12.2 UNIVAC I, the first commercial computer in the United States.**

It correctly predicted the outcome of the 1952 presidential election, but its initial forecast was withheld from broadcast because experts doubted the use of such early results.

IBM had been in the punched card and office automation business but didn't start building computers until 1950. The first IBM computer, the IBM 701, shipped in 1952, and eventually 19 units were sold. In the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited.

In 1964, after investing \$5 billion, IBM made a bold move with the announcement of the System/360. An IBM spokesman said the following at the time:

*We are not at all humble in this announcement. This is the most important product announcement that this corporation has ever made in its history. It's not a computer in any previous sense. It's not a product, but a line of products ... that spans in performance from the very low part of the computer line to the very high.*

Moving the idea of the architecture **abstraction** into commercial reality, IBM announced six implementations of the System/360 architecture that varied in price and performance by a factor of 25. [Figure e1.12.3](#) shows four of these models. IBM bet its company on the success of a *computer family*, and IBM won. The System/360 and its successors dominated the large computer market.





**FIGURE E1.12.3 IBM System/360 computers:  
models 40, 50, 65, and 75 were all introduced in  
1964.**

These four models varied in cost and performance by a factor of almost 10; it grows to 25 if we include models 20 and 30 (not shown). The clock rate, range of memory sizes, and approximate price for only the processor and memory of average size: (a) model 40, 1.6 MHz, 32 KB–256 KB, \$225,000; (b) model 50, 2.0 MHz, 128 KB–256 KB, \$550,000; (c) model 65, 5.0 MHz, 256 KB–1 MB, \$1,200,000; and (d) model 75, 5.1 MHz, 256 KB–1 MB, \$1,900,000. Adding I/O devices typically increased the price by factors of 1.8 to 3.5, with higher factors for cheaper models.

About a year later, *Digital Equipment Corporation* (DEC) unveiled the PDP-8, the first commercial *minicomputer*. This small machine was a breakthrough in low-cost design, allowing DEC to offer a computer for under \$20,000. Minicomputers were the forerunners of microprocessors, with Intel inventing the first microprocessor in 1971—the Intel 4004.

In 1963 came the announcement of the first *supercomputer*. This

announcement came neither from the large companies nor even from the high-tech centers. Seymour Cray led the design of the Control Data Corporation CDC 6600 in Minnesota. This machine included many ideas that are beginning to be found in the latest microprocessors. Cray later left CDC to form Cray Research, Inc., in Wisconsin. In 1976, he announced the Cray-1 ([Figure e1.12.4](#)). This machine was simultaneously the fastest in the world, the most expensive, and the computer with the best cost/performance for scientific programs.



**FIGURE E1.12.4 Cray-1, the first commercial vector supercomputer, announced in 1976.**

This machine had the unusual distinction of being both the fastest computer for scientific applications and the computer with the best price/performance for those applications. Viewed from the top, the computer looks like the letter C. Seymour Cray passed away in 1996 because of injuries sustained in an automobile accident. At the time of his death, this 70-year-old computer pioneer was working on his vision of the next generation of supercomputers. (See [www.cray.com](http://www.cray.com) for more details.)

While Seymour Cray was creating the world's most expensive

computer, other designers around the world were looking at using the microprocessor to create a computer so cheap that you could have it at home. There is no single fountainhead for the *personal computer*, but in 1977, the Apple IIe (Figure e1.12.5) from Steve Jobs and Steve Wozniak set standards for low cost, high volume, and high reliability that defined the personal computer industry.

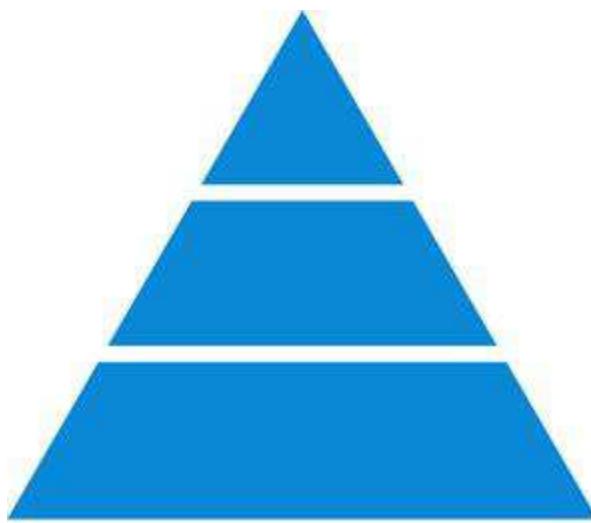


**FIGURE E1.12.5 The Apple IIc Plus.**

Designed by Steve Wozniak, the Apple IIc set standards of cost and reliability for the industry.

However, even with a 4-year head start, Apple's personal computers finished second in popularity. The IBM Personal Computer, announced in 1981, became the best-selling computer of any kind; its success gave Intel the most popular microprocessor and Microsoft the most popular operating system. Today, the most popular CD is the Microsoft operating system, even though it costs many times more than a music CD! Of course, over the more than 30 years that the IBM-compatible personal computer has existed, it has evolved greatly. In fact, the first personal computers had 16-bit processors and 64 kilobytes of **memory**, and a low-density, slow floppy disk was the only nonvolatile storage! Floppy disks were originally developed by IBM for loading diagnostic programs in

mainframes, but were a major I/O device in personal computers for almost 20 years before the advent of CDs and networking made them obsolete as a method for exchanging data.



## HIERARCHY

Of course, Intel microprocessors have also evolved since the first PC, which used a 16-bit processor with an 8-bit external interface! In Chapter 2, we write about the evolution of the Intel architecture.

The first personal computers were quite simple, with little or no graphics capability, no pointing devices, and primitive operating systems compared to those of today. The computer that inspired many of the architectural and software concepts that characterize the modern desktop machines was the Xerox Alto, shown in [Figure e1.12.6](#). The Alto was created as an experimental prototype of a future computer; there were several hundred Altos built, including a significant number that were donated to universities. Among the technologies incorporated in the Alto were:

- a bit-mapped graphics display integrated with a computer (earlier graphics displays acted as terminals, usually connected to larger computers)
- a mouse, which was invented earlier, but included on every Alto and used extensively in the user interface
- a local area network (LAN), which became the precursor to the Ethernet
- a user interface based on Windows and featuring a WYSIWYG

(what you see is what you get) editor and interactive drawing programs



**FIGURE E1.12.6** The Xerox Alto was the primary inspiration for the modern desktop computer.

It included a mouse, a bit-mapped scheme, a Windows-based user interface, and a local network connection.

In addition, both file servers and print servers were developed and interfaced via the local area network, and connections between the local area network and the wide area ARPAnet produced the first versions of Internet-style networking. The Xerox Alto was incredibly influential and clearly affected the design of a wide variety of computers and software systems, including the Apple Macintosh, the IBM-compatible PC, MacOS and Windows, and Sun and other early workstations.

## Measuring Performance

From the earliest days of computing, designers have specified performance goals—ENIAC was to be 1000 times faster than the Harvard Mark-I, and the IBM Stretch (7030) was to be 100 times faster than the fastest computer then in existence. What wasn't clear, though, was how this performance was to be measured.

The original measure of performance was the time required to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one was the same as the others. As the execution times of instructions in a computer became more diverse, however, the time required for one operation was no longer useful for comparisons.

To consider these differences, an *instruction mix* was calculated by measuring the relative frequency of instructions in a computer across many programs. Multiplying the time for each instruction by its weight in the mix gave the user the *average instruction execution time*. (If measured in clock cycles, average instruction execution time is the same as average CPI.) Since instruction sets were similar, this was a more precise comparison than add times. From average instruction execution time, then, it was only a small step to MIPS. MIPS had the virtue of being easy to understand; hence, it grew in popularity.

## The Quest for an Average Program

As processors were becoming more sophisticated and relied on memory hierarchies (the topic of [Chapter 5](#)) and pipelining (the topic of [Chapter 4](#)), a single execution time for each instruction no longer existed; neither execution time nor MIPS, therefore, could be

calculated from the instruction mix and the manual.

Although it might seem obvious today that the right thing to do would have been to develop a set of real applications that could be used as standard benchmarks, this was a difficult task until relatively recent times. Variations in operating systems and language standards made it hard to create large programs that could be moved from computer to computer simply by recompiling.

Instead, the next step was benchmarking using synthetic programs. The Whetstone synthetic program was created by measuring scientific programs written in Algol-60 (see Curnow and Wichmann's [1976] description). This program was converted to Fortran and was widely used to characterize scientific program performance. Whetstone performance is typically quoted in Whetstones per second—the number of executions of a single iteration of the Whetstone benchmark! Dhrystone is another synthetic benchmark that is still used in some embedded computing circles (see Weicker's [1984] description and methodology).

About the same time Whetstone was developed, the concept of *kernel benchmarks* gained popularity. Kernels are small, time-intensive pieces from real programs that are extracted and then used as benchmarks. This approach was developed primarily for benchmarking high-end computers, especially supercomputers. Livermore Loops and Linpack are the best-known examples. The Livermore Loops consist of a series of 21 small loop fragments. Linpack consists of a portion of a linear algebra subroutine package. Kernels are best used to isolate the performance of individual features of a computer and to explain the reasons for differences in the performance of real programs. Because scientific applications often use small pieces of code that execute for a long time, characterizing performance with kernels is most popular in this application class. Although kernels help illuminate performance, they frequently overstate the performance on real applications.

## SPECulating about Performance

An important advance in performance evaluation was the formation of the System Performance Evaluation Cooperative

(SPEC) group in 1988. SPEC comprises representatives of many computer companies—the founders being Apollo/ Hewlett-Packard, DEC, MIPS, and Sun—who have agreed on a set of real programs and inputs that all will run. It is worth noting that SPEC couldn't have come into being before portable operating systems and the popularity of high-level languages. Now compilers, too, are accepted as a proper part of the performance of computer systems and must be measured in any evaluation.

History teaches us that while the SPEC effort may be useful with current computers, it will not meet the needs of the next generation without changing. In 1991, a throughput measure was added, based on running multiple versions of the benchmark. It is most useful for evaluating timeshared usage of a uniprocessor or a multiprocessor. Other system benchmarks that include OS-intensive and I/O-intensive activities have also been added. Another change was the decision to drop some benchmarks and add others. One result of the difficulty in finding benchmarks was that the initial version of the SPEC benchmarks (called SPEC89) contained six floating-point benchmarks but only four integer benchmarks. Calculating a single summary measurement using the geometric mean of execution times normalized to a VAX-11/780 meant that this measure favored computers with strong floating-point performance.

In 1992, a new benchmark set (called SPEC92) was introduced. It incorporated additional benchmarks, dropped matrix300, and provided separate means (SPEC INT and SPECFP) for integer and floating-point programs. In addition, the SPECbase measure, which disallows program-specific optimization flags, was added to provide users with a performance measurement that would more closely match what they might experience on their own programs. The SPECFP numbers show the largest increase versus the base SPECFP measurement, typically ranging from 15% to 30% higher.

In 1995, the benchmark set was once again updated, adding some new integer and floating-point benchmarks, as well as removing some benchmarks that suffered from flaws or had running times that had become too small given the factor of 20 or more performance improvement since the first SPEC release. SPEC95 also changed the base computer for normalization to a Sun SPARC Station 10/40, since operating versions of the original base computer were becoming difficult to find!

The most recent version of SPEC is SPEC2006. What is perhaps most surprising is that all floating-point programs in SPEC2006 are new, and for integer programs just two are from SPEC2000, one from SPEC95, none from SPEC92, and one from SPEC89. The sole survivor from SPEC89 is the gcc compiler.

SPEC has also added benchmark suites beyond the original suites targeted at CPU performance. In 2008, SPEC provided benchmark sets for graphics, high-performance scientific computing, object-oriented computing, file systems, Web servers and clients, Java, engineering CAD applications, and power.

## The Growth of Embedded Computing

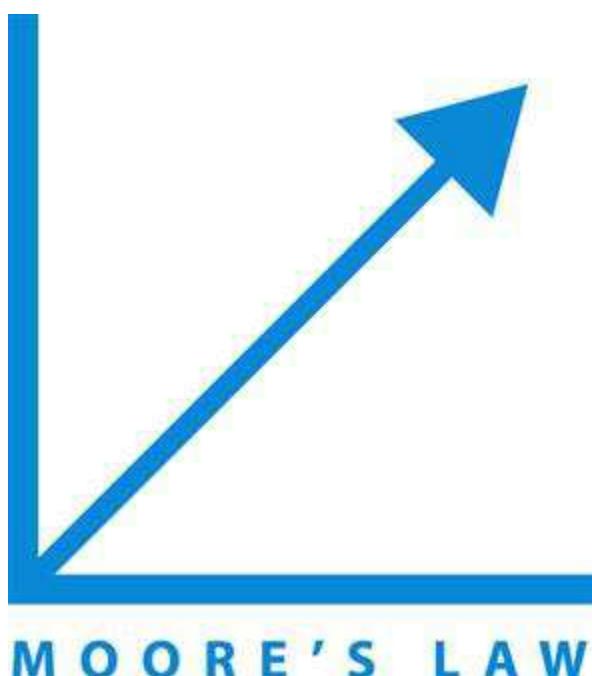
Embedded processors have been around for a very long time; in fact, the first minicomputers and the first microprocessors were originally developed for controlling functions in a laboratory or industrial application. For many years, the dominant use of embedded processors was for industrial control applications, and although this use continued to grow, the processors tended to be very cheap and the performance relatively low. For example, the best-selling processor in the world remains an 8-bit micro controller used in cars, some home appliances, and other simple applications.

The late 1980s and early 1990s saw the emergence of new opportunities for embedded processors, ranging from more advanced video games and set-top boxes to cell phones and personal digital assistants. The rapidly increasing number of information appliances and the growth of networking have driven dramatic surges in the number of embedded processors, as well as the performance requirements. To evaluate performance, the embedded community was inspired by SPEC to create the *Embedded Microprocessor Benchmark Consortium (EEMBC)*. Started in 1997, it consists of a collection of kernels organized into suites that address different portions of the embedded industry. They announced the second generation of these benchmarks in 2007.

## A Half-Century of Progress

Since 1951, there have been thousands of new computers using a wide range of technologies and having widely varying capabilities.

Figure e1.12.7 summarizes the key characteristics of some machines mentioned in this section and shows the dramatic changes that have occurred in just over **50 years**. After adjusting for inflation, price/performance has improved by almost 100 billion in 55 years, or about 58% per year. Another way to say it is we've seen a factor of 10,000 improvement in cost and a factor of 10,000,000 improvement in performance.



Year	Name	Size (cu. ft.)	Power (watts)	Performance (adds/sec)	Memory (KB)	Price	Price/ performance vs. UNIVAC	Adjusted price (2007 \$)	Adjusted price/ performance vs. UNIVAC
1951	UNIVAC I	1,000	125,000	2,000	48	\$1,000,000	1	\$7,670,724	1
1964	IBM S/360 model 50	60	10,000	500,000	64	\$1,000,000	263	\$6,018,798	319
1965	PDP-8	8	500	330,000	4	\$16,000	10,855	\$94,685	13,367
1976	Cray-1	58	60,000	166,000,000	32,000	\$4,000,000	21,842	\$13,509,798	47,127
1981	IBM PC	1	150	240,000	256	\$3,000	42,105	\$6,859	134,208
1991	HP 9000/ model 750	2	500	50,000,000	16,384	\$7,400	3,556,188	\$11,807	16,241,889
1996	Intel PPro PC (200 MHz)	2	500	400,000,000	16,384	\$4,400	47,846,890	\$6,211	247,021,234
2003	Intel Pentium 4 PC (3.0 GHz)	2	500	6,000,000,000	262,144	\$1,600	1,875,000,000	\$2,009	11,451,750,000
2007	AMD Barcelona PC (2.5 GHz)	2	250	20,000,000,000	2,097,152	\$800	12,500,000,000	\$800	95,884,051,042

**FIGURE E1.12.7 Characteristics of key commercial computers since 1950, in actual dollars and in 2007 dollars adjusted for inflation.**

The last row assumes we can fully utilize the potential performance of the four cores in Barcelona. In contrast to [Figure e1.12.3](#), here the price of the IBM S/360 model 50 includes I/O devices. (Source: The Computer History Museum and Producer Price Index for Industrial Commodities.)

Readers interested in computer history should consult *Annals of the History of Computing*, a journal devoted to the history of computing. Several books describing the early days of computing have also appeared, many written by the pioneers including Goldstine [1972], Metropolis et al. [1980], and Wilkes [1985].

## Further Reading

Barroso, L. and U. Hölzle [2007]. "The case for energy-proportional computing", *IEEE Computer* December.

*A plea to change the nature of computer components so that they use much less power when lightly utilized.*

Bell, C. G. [1996]. *Computer Pioneers and Pioneer Computers*, ACM and the Computer Museum, videotapes.

*Two videotapes on the history of computing, produced by Gordon and Gwen Bell, including the following machines and their inventors: Harvard Mark-I, ENIAC, EDSAC, IAS machine, and many others.*

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

*A classic paper explaining computer hardware and software before the first stored-program computer was built. We quote extensively from it in Chapter 3. It simultaneously explained computers to the world and was a source of controversy because the first draft did not give credit to Eckert and Mauchly.*

Campbell-Kelly, M. and W. Aspray [1996]. *Computer: A History of the Information Machine*, Basic Books, New York.

*Two historians chronicle the dramatic story. The New York Times calls it well written and authoritative.*

Ceruzzi, P. F. [1998]. *A History of Modern Computing*, MIT Press, Cambridge, MA.

*Contains a good description of the later history of computing: the integrated circuit and its impact, personal computers, UNIX, and the Internet.*

Curnow, H. J. and B. A. Wichmann [1976]. "A synthetic benchmark", *The Computer J.* 19(1):80.

*Describes the first major synthetic benchmark, Whetstone, and how it was created.*

Flemming, P. J. and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmark results", *Comm. ACM* 29:3 (March), 218–21.

*Describes some of the underlying principles in using different means to summarize performance results.*

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ.

*A personal view of computing by one of the pioneers who worked with von Neumann.*

Hayes, B. [2007]. "Computing in a parallel universe", *American Scientist* Vol. 95(November–December):476–480.

*An overview of the parallel computing challenge written for the layman.*

Hennessy, J. L. and D. A. Patterson [2007]. *Chapter 1 of Computer Architecture: A Quantitative Approach*, fourth edition, Morgan Kaufmann Publishers, San Francisco.

*Section 1.5 goes into more detail on power, Section 1.6 contains much more detail on the cost of integrated circuits and explains the reasons for the difference between price and cost, and Section 1.8 gives more details on evaluating performance.*

Lampson, B. W. [1986]. "Personal distributed computing; The Alto and Ethernet software." In ACM Conference on the History of Personal Workstations (January).

Thacker, C. R. [1986]. "Personal distributed computing: The Alto and Ethernet hardware," In ACM Conference on the History of Personal Workstations (January).

*These two papers describe the software and hardware of the landmark Alto.*

Metropolis, N., J. Howlett, and G.-C. Rota (Eds.) [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.

*A collection of essays that describe the people, software, computers, and laboratories involved in the first experimental and commercial computers. Most of the authors were personally involved in the projects. An excellent*

*bibliography of early reports concludes this interesting book.*

*Public Broadcasting System [1992]. The Machine That Changed the World, videotapes.*

*These five 1-hour programs include rare footage and interviews with pioneers of the computer industry.*

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, MA.

*Short biographies of 31 computer pioneers.*

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (October), 375–76

*A historian's perspective on Atanasoff versus Eckert and Mauchly.*

Weicker, R. P. [1984]. "Dhrystone: a synthetic systems programming benchmark", *Communications of the ACM* 27(10):1013–1030.

*Description of a synthetic benchmarking program for systems code.*

Wilkes, M.V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

*A personal view of computing by one of the pioneers.*

## 1.13 Exercises

The relative time ratings of exercises are shown in square brackets after each exercise number. On average, an exercise rated [10] will take you twice as long as one rated [5]. Sections of the text that should be read before attempting an exercise will be given in angled brackets; for example, <§1.4> means you should have read [Section 1.4](#), Under the Covers, to help you solve this exercise.

1.1 [2] <§1.1> Aside from the smart cell phones used by a billion people, list and describe four other types of computers.

1.2 [5] <§1.2> The eight great ideas in computer architecture are similar to ideas from other fields. Match the eight ideas from computer architecture, "Design for Moore's Law," "Use Abstraction to Simplify Design," "Make the Common Case Fast," "Performance via Parallelism," "Performance via Pipelining," "Performance via Prediction," "Hierarchy of Memories," and "Dependability via Redundancy" to the following ideas from other fields:

- a. Assembly lines in automobile manufacturing
- b. Suspension bridge cables
- c. Aircraft and marine navigation systems that incorporate

- wind information
- d. Express elevators in buildings
  - e. Library reserve desk
  - f. Increasing the gate area on a CMOS transistor to decrease its switching time
  - g. Adding electromagnetic aircraft catapults (which are electrically powered as opposed to current steam-powered models), allowed by the increased power generation offered by the new reactor technology
  - h. Building self-driving cars whose control systems partially rely on existing sensor systems already installed into the base vehicle, such as lane departure systems and smart cruise control systems
- 1.3 [2] <§1.3> Describe the steps that transform a program written in a high-level language such as C into a representation that is directly executed by a computer processor.
- 1.4 [2] <§1.4> Assume a color display using 8 bits for each of the primary colors (red, green, blue) per pixel and a frame size of  $1280 \times 1024$ .
- a. What is the minimum size in bytes of the frame buffer to store a frame?
  - b. How long would it take, at a minimum, for the frame to be sent over a 100 Mbit/s network?
- 1.5 [4] <§1.6> Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.
- a. Which processor has the highest performance expressed in instructions per second?
  - b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.
  - c. We are trying to reduce the execution time by 30%, but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?
- 1.6 [20] <§1.6> Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (classes A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3, and P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2.

Given a program with a dynamic instruction count of 1.0E6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which is faster: P1 or P2?

- a. What is the global CPI for each implementation?
- b. Find the clock cycles required in both cases.

1.7 [15] <§1.6> Compilers can have a profound impact on the performance of an application. Assume that for a program, compiler A results in a dynamic instruction count of 1.0E9 and has an execution time of 1.1 s, while compiler B results in a dynamic instruction count of 1.2E9 and an execution time of 1.5 s.

- a. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.
- b. Assume the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?
- c. A new compiler is developed that uses only 6.0E8 instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using compiler A or B on the original processor?

1.8 The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and voltage of 1.25 V. Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power. The Core i5 Ivy Bridge, released in 2012, has a clock rate of 3.4 GHz and voltage of 0.9 V. Assume that, on average, it consumed 30 W of static power and 40 W of dynamic power.

1.8.1 [5] <§1.7> For each processor find the average capacitive loads.

1.8.2 [5] <§1.7> Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology.

1.8.3 [15] <§1.7> If the total dissipated power is to be reduced by 10%, how much should the voltage be reduced to maintain the same leakage current? Note: power is defined as the product of voltage and current.

1.9 Assume for arithmetic, load/store, and branch instructions, a processor has CPIs of 1, 12, and 5, respectively. Also assume that on a single processor a program requires the execution of 2.56E9

arithmetic instructions,  $1.28E9$  load/store instructions, and 256 million branch instructions. Assume that each processor has a 2 GHz clock frequency.

Assume that, as the program is parallelized to run over multiple cores, the number of arithmetic and load/store instructions per processor is divided by  $0.7 \times p$  (where  $p$  is the number of processors) but the number of branch instructions per processor remains the same.

- 1.9.1 [5] <§1.7> Find the total execution time for this program on 1, 2, 4, and 8 processors, and show the relative speedup of the 2, 4, and 8 processors result relative to the single processor result.
  - 1.9.2 [10] <§§1.6, 1.8> If the CPI of the arithmetic instructions was doubled, what would the impact be on the execution time of the program on 1, 2, 4, or 8 processors?
  - 1.9.3 [10] <§§1.6, 1.8> To what should the CPI of load/store instructions be reduced in order for a single processor to match the performance of four processors using the original CPI values?
- 1.10 Assume a 15 cm diameter wafer has a cost of 12, contains 84 dies, and has  $0.020$  defects/cm $^2$ . Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has  $0.031$  defects/cm $^2$ .
- 1.10.1 [10] <§1.5> Find the yield for both wafers.
  - 1.10.2 [5] <§1.5> Find the cost per die for both wafers.
  - 1.10.3 [5] <§1.5> If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.
  - 1.10.4 [5] <§1.5> Assume a fabrication process improves the yield from 0.92 to 0.95. Find the defects per area unit for each version of the technology given a die area of 200 mm $^2$ .
- 1.11 The results of the SPEC CPU2006 bzip2 benchmark running on an AMD Barcelona has an instruction count of  $2.389E12$ , an execution time of 750 s, and a reference time of 9650 s.
- 1.11.1 [5] <§§1.6, 1.9> Find the CPI if the clock cycle time is 0.333 ns.
  - 1.11.2 [5] <§1.9> Find the SPECratio.
  - 1.11.3 [5] <§§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI.

- 1.11.4 [5] <§§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.
- 1.11.5 [5] <§§1.6, 1.9> Find the change in the SPECratio for this change.
- 1.11.6 [10] <§1.6> Suppose that we are developing a new version of the AMD Barcelona processor with a 4 GHz clock rate. We have added some additional instructions to the instruction set in such a way that the number of instructions has been reduced by 15%. The execution time is reduced to 700 s and the new SPECratio is 13.7. Find the new CPI.
- 1.11.7 [10] <§1.6> This CPI value is larger than obtained in 1.11.1 as the clock rate was increased from 3 GHz to 4 GHz.  
Determine whether the increase in the CPI is similar to that of the clock rate. If they are dissimilar, why?
- 1.11.8 [5] <§1.6> By how much has the CPU time been reduced?
- 1.11.9 [10] <§1.6> For a second benchmark, libquantum, assume an execution time of 960 ns, CPI of 1.61, and clock rate of 3 GHz. If the execution time is reduced by an additional 10% without affecting the CPI and with a clock rate of 4 GHz, determine the number of instructions.
- 1.11.10 [10] <§1.6> Determine the clock rate required to give a further 10% reduction in CPU time while maintaining the number of instructions and with the CPI unchanged.
- 1.11.11 [10] <§1.6> Determine the clock rate if the CPI is reduced by 15% and the CPU time by 20% while the number of instructions is unchanged.
- 1.12 Section 1.10 cites as a pitfall the utilization of a subset of the performance equation as a performance metric. To illustrate this, consider the following two processors. P1 has a clock rate of 4 GHz, average CPI of 0.9, and requires the execution of 5.0E9 instructions. P2 has a clock rate of 3 GHz, an average CPI of 0.75, and requires the execution of 1.0E9 instructions.
- 1.12.1 [5] <§§1.6, 1.10> One usual fallacy is to consider the computer with the largest clock rate as having the highest performance. Check if this is true for P1 and P2.
- 1.12.2 [10] <§§1.6, 1.10> Another fallacy is to consider that the processor executing the largest number of instructions will need a larger CPU time. Considering that processor P1 is

executing a sequence of  $1.0E9$  instructions and that the CPI of processors P1 and P2 do not change, determine the number of instructions that P2 can execute in the same time that P1 needs to execute  $1.0E9$  instructions.

1.12.3 [10] <§§1.6, 1.10> A common fallacy is to use MIPS (*millions of instructions per second*) to compare the performance of two different processors, and consider that the processor with the largest MIPS has the largest performance. Check if this is true for P1 and P2.

1.12.4 [10] <§1.10> Another common performance figure is MFLOPS (millions of floating-point operations per second), defined as

$$\text{MFLOPS} = \text{No. FP operations} / (\text{execution time} \times 1E6)$$

but this figure has the same problems as MIPS. Assume that 40% of the instructions executed on both P1 and P2 are floating-point instructions. Find the MFLOPS figures for the processors.

1.13 Another pitfall cited in [Section 1.10](#) is expecting to improve the overall performance of a computer by improving only one aspect of the computer. Consider a computer running a program that requires 250 s, with 70 s spent executing FP instructions, 85 s executed L/S instructions, and 40 s spent executing branch instructions.

1.13.1 [5] <§1.10> By how much is the total time reduced if the time for FP operations is reduced by 20%?

1.13.2 [5] <§1.10> By how much is the time for INT operations reduced if the total time is reduced by 20%?

1.13.3 [5] <§1.10> Can the total time be reduced by 20% by reducing only the time for branch instructions?

1.14 Assume a program requires the execution of  $50 \times 10^6$  FP instructions,  $110 \times 10^6$  INT instructions,  $80 \times 10^6$  L/S instructions, and  $16 \times 10^6$  branch instructions. The CPI for each type of instruction is 1, 1, 4, and 2, respectively. Assume that the processor has a 2 GHz clock rate.

1.14.1 [10] <§1.10> By how much must we improve the CPI of FP instructions if we want the program to run two times faster?

1.14.2 [10] <§1.10> By how much must we improve the CPI of L/S

instructions if we want the program to run two times faster?

1.14.3 [5] <§1.10> By how much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

1.15 [5] <§1.8> When a program is adapted to run on multiple processors in a multiprocessor system, the execution time on each processor is comprised of computing time and the overhead time required for locked critical sections and/or to send data from one processor to another.

Assume a program requires  $t = 100$  s of execution time on one processor. When run  $p$  processors, each processor requires  $t/p$  s, as well as an additional 4 s of overhead, irrespective of the number of processors. Compute the per-processor execution time for 2, 4, 8, 16, 32, 64, and 128 processors. For each case, list the corresponding speedup relative to a single processor and the ratio between actual speedup versus ideal speedup (speedup if there was no overhead).

## Answers to Check Yourself

§1.1, page 10: Discussion questions: many answers are acceptable.

§1.4, page 24: DRAM memory: volatile, short access time of 50 to 70 nanoseconds, and cost per GB is \$5 to \$10. Disk memory: nonvolatile, access times are 100,000 to 400,000 times slower than DRAM, and cost per GB is 100 times cheaper than DRAM. Flash memory: nonvolatile, access times are 100 to 1000 times slower than DRAM, and cost per GB is 7 to 10 times cheaper than DRAM.

§1.5, page 28: 1, 3, and 4 are valid reasons. Answer 5 can be generally true because high volume can make the extra investment to reduce die size by, say, 10% a good economic decision, but it doesn't have to be true.

§1.6, page 33: 1. a: both, b: latency, c: neither. 7 seconds.

§1.6, page 40: b.

§1.10, page 51: a. Computer A has the higher MIPS rating. b. Computer B is faster.



# INSTRUCTION SET ARCHITECTURE

## CHAPTER OBJECTIVES

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

This chapter considers the way programs are executed in a computer from the machine instruction set viewpoint. Chapter 1 introduced the general concept that both program instructions and data operands are stored in the memory. In this chapter, we discuss how instructions are composed and study the ways in which sequences of instructions are brought from the memory into the processor and executed to perform a given task. The addressing methods that are commonly used for accessing operands in memory locations and processor registers are also presented.

The emphasis here is on basic concepts. We use a generic style to describe machine instructions and operand addressing methods that are typical of those found in commercial processors. A sufficient number of instructions and addressing methods are introduced to enable us to present complete, realistic programs for simple tasks. These generic programs are specified at the assembly-language level, where machine instructions and operand addressing information are represented by symbolic names. A complete instruction set, including operand addressing methods, is often referred to as the *instruction set architecture* (ISA) of a processor. For the discussion of basic concepts in this chapter, it is not necessary to define a complete instruction set, and we will not attempt to do so. Instead, we will present enough examples to illustrate the capabilities of a typical instruction set.

The concepts introduced in this chapter and in Chapter 3, which deals with input/output techniques, are essential for understanding the functionality of computers. Our choice of the generic style of presentation makes the material easy to read and understand. Also, this style allows a general discussion that is not constrained by the characteristics of a particular processor.

Since it is interesting and important to see how the concepts discussed are implemented in a real computer, we supplement our presentation in Chapters 2 and 3 with four examples of popular commercial processors. These processors are presented in Appendices B to E. Appendix B deals with the Nios II processor from Altera Corporation. Appendix C presents the ColdFire processor from Freescale Semiconductor, Inc. Appendix D discusses the ARM processor from ARM Ltd. Appendix E presents the basic architecture of processors made by Intel Corporation. The generic programs in Chapters 2 and 3 are presented in terms of the specific instruction sets in each of the appendices.

The reader can choose only one processor and study the material in the corresponding appendix to get an appreciation for commercial ISA design. However, knowledge of the material in these appendices is not essential for understanding the material in the main body of the book.

The vast majority of programs are written in high-level languages such as C, C++, or Java. To execute a high-level language program on a processor, the program must be translated into the machine language for that processor, which is done by a compiler program. Assembly language is a readable symbolic representation of machine language. In this book we make extensive use of assembly language, because this is the best way to describe how computers work.

We will begin the discussion in this chapter by considering how instructions and data are stored in the memory and how they are accessed for processing.

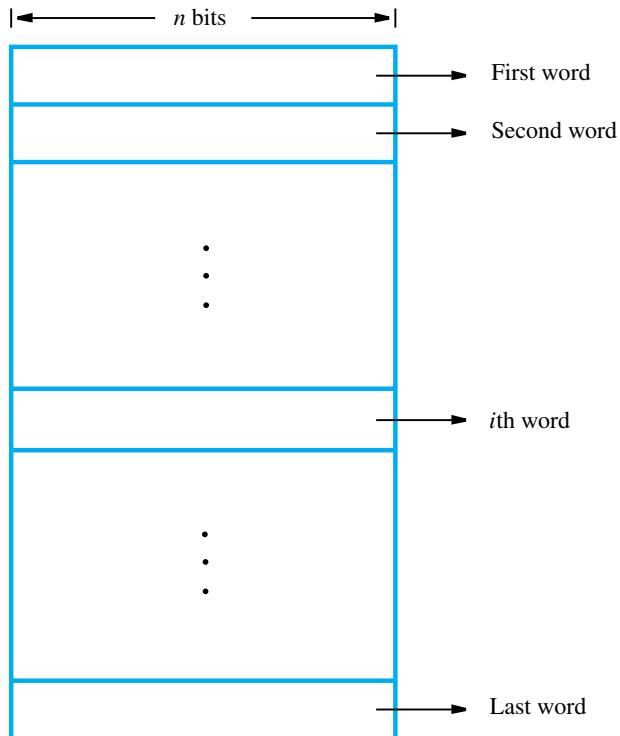
## 2.1 MEMORY LOCATIONS AND ADDRESSES

We will first consider how the memory of a computer is organized. The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For

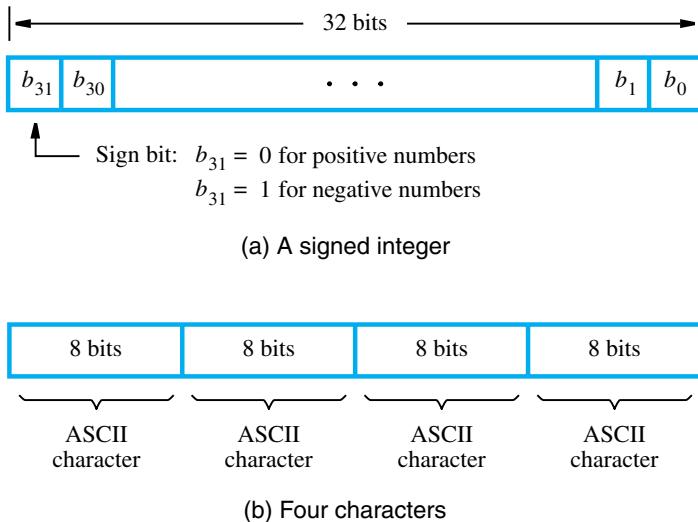
for this purpose, the memory is organized so that a group of  $n$  bits can be stored or retrieved in a single, basic operation. Each group of  $n$  bits is referred to as a *word* of information, and  $n$  is called the *word length*. The memory of a computer can be schematically represented as a collection of words, as shown in Figure 2.1.

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2. A unit of 8 bits is called a *byte*. Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section, after we have described instructions at the assembly-language level.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each location. It is customary to use numbers from 0 to  $2^k - 1$ , for some suitable value of  $k$ , as the addresses of successive locations in the memory. Thus, the memory can have up to  $2^k$  addressable locations. The  $2^k$  addresses constitute the *address space* of the computer. For example, a 24-bit address generates an address space of  $2^{24}$  (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number  $2^{20}$  (1,048,576). A 32-bit address creates an address space of  $2^{32}$  or 4G (4 giga) locations, where 1G is  $2^{30}$ . Other notational conventions



**Figure 2.1** Memory words.



**Figure 2.2** Examples of encoded information in a 32-bit word.

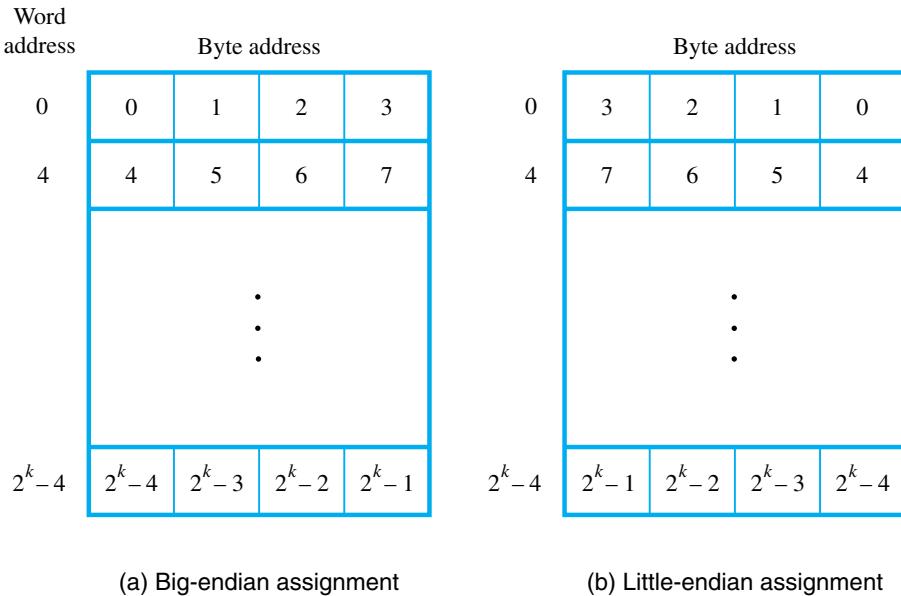
that are commonly used are K (kilo) for the number  $2^{10}$  (1,024), and T (tera) for the number  $2^{40}$ .

### 2.1.1 BYTE ADDRESSABILITY

We now have three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2, .... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, ..., with each word consisting of four bytes.

### 2.1.2 BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.3. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words “more significant” and “less significant” are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, ..., are taken as the addresses of successive words in the memory

**Figure 2.3** Byte and word addressing.

of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 2.2a. It is the most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is,  $b_7, b_6, \dots, b_0$ , from left to right.

### 2.1.3 WORD ALIGNMENT

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, ..., as shown in Figure 2.3. We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, ..., and for a word length of 64 ( $2^3$  bytes), aligned words begin at byte addresses 0, 8, 16, ... .

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. But, the most common case is to use aligned addresses, which makes accessing of memory operands more efficient, as we will see in Chapter 8.

### 2.1.4 ACCESSING NUMBERS AND CHARACTERS

A number usually occupies one word, and can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

For programming convenience it is useful to have different ways of specifying addresses in program instructions. We will deal with this issue in Section 2.4.

---

## 2.2 MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Read* and *Write*.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

The details of the hardware implementation of these operations are treated in Chapters 5 and 6. In this chapter, we consider all operations from the viewpoint of the ISA, so we concentrate on the logical handling of instructions and operands.

---

## 2.3 INSTRUCTIONS AND INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation.

### 2.3.1 REGISTER TRANSFER NOTATION

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or OUTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression

$$R2 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R2.

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

### 2.3.2 ASSEMBLY-LANGUAGE NOTATION

We need another type of notation to represent machine instructions and programs. For this, we use *assembly language*. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

Load R2, LOC

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*, which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

### 2.3.3 RISC AND CISC INSTRUCTION SETS

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in “pipelined” fashion to overlap activity and reduce total execution time of a program, as we will discuss in Chapter 6. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

We will start our presentation by concentrating on RISC-style instruction sets because they are simpler and therefore easier to understand. Later we will deal with CISC-style instruction sets and explain the key differences between the two approaches.

### 2.3.4 INTRODUCTION TO RISC INSTRUCTION SETS

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A *load/store architecture* is used, in which
  - Memory operands are accessed only using Load and Store instructions.
  - All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time. If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

Load destination, source

or more specifically

Load processor\_register, memory\_location

The memory location can be specified in several ways. The term *addressing modes* is used to refer to the different ways in which this may be accomplished, as we will discuss in Section 2.4.

Let us now consider a typical arithmetic operation. The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. For simplicity, we will refer to the addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

$$C \leftarrow [A] + [B]$$

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

Load	R2, A
Load	R3, B
Add	R4, R2, R3
Store	R4, C

We say that Add is a *three-operand*, or a *three-address*, instruction of the form

Add destination, source1, source2

The Store instruction is of the form

Store source, destination

where the source is a processor register and the destination is a memory location. Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention.

Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

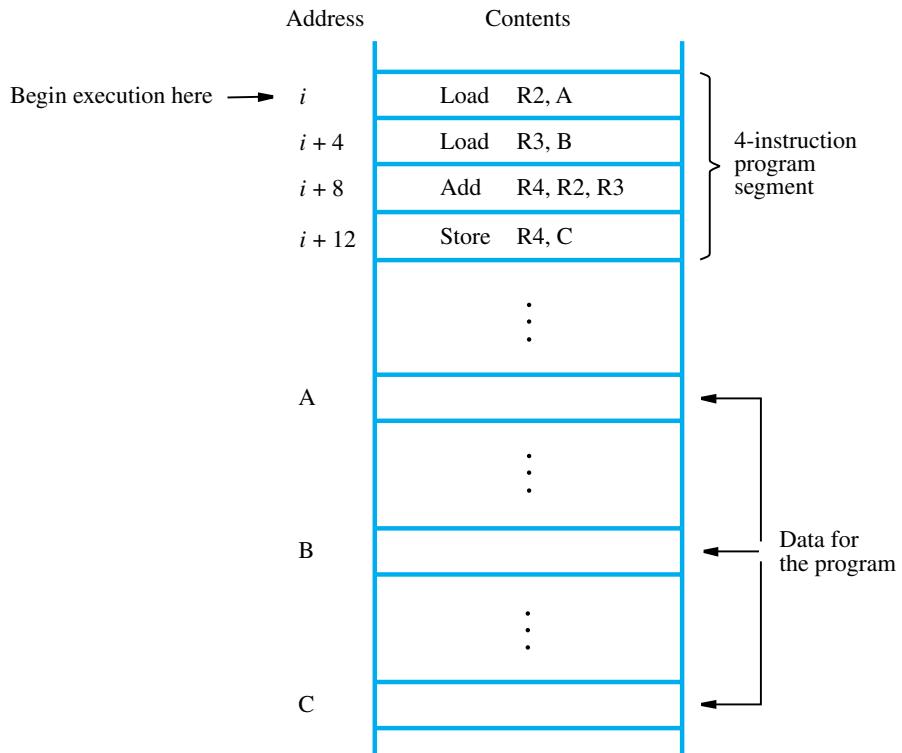
Add R3, R2, R3

and the last instruction would become

Store R3, C

### 2.3.5 INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

In the preceding subsection, we used the task  $C = A + B$ , implemented as  $C \leftarrow [A] + [B]$ , as an example. Figure 2.4 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location  $i$ . Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses  $i + 4$ ,  $i + 8$ , and  $i + 12$ . For simplicity, we assume that a desired



**Figure 2.4** A program for  $C \leftarrow [A] + [B]$ .

memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved. We will resolve this issue later in Section 2.4.

Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction ( $i$  in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location  $i + 12$  is executed, the PC contains the value  $i + 16$ , which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

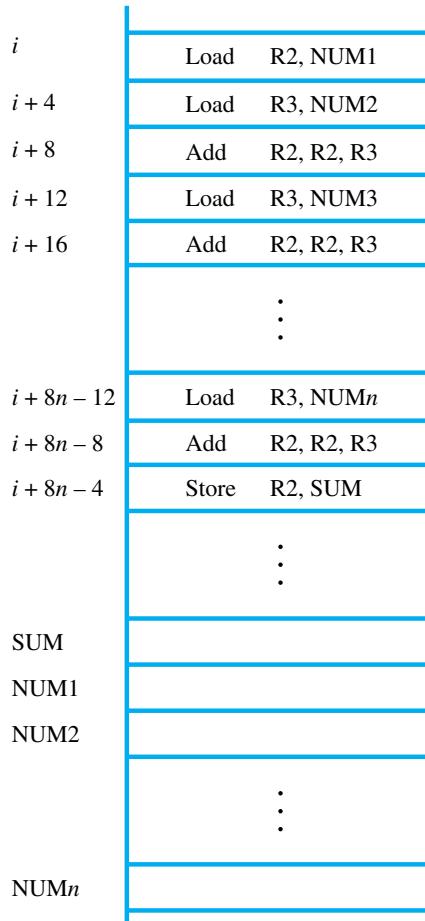
### 2.3.6 BRANCHING

Consider the task of adding a list of  $n$  numbers. The program outlined in Figure 2.5 is a generalization of the program in Figure 2.4. The addresses of the memory locations containing the  $n$  numbers are symbolically given as NUM1, NUM2, ..., NUM $n$ , and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Load and Add instructions, as in Figure 2.5, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure 2.6 shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch\_if\_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways, as will be described in Section 2.4. For now, we concentrate on how to create and control a program loop.

Assume that the number of entries in the list,  $n$ , is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction

Subtract R2, R2, #1



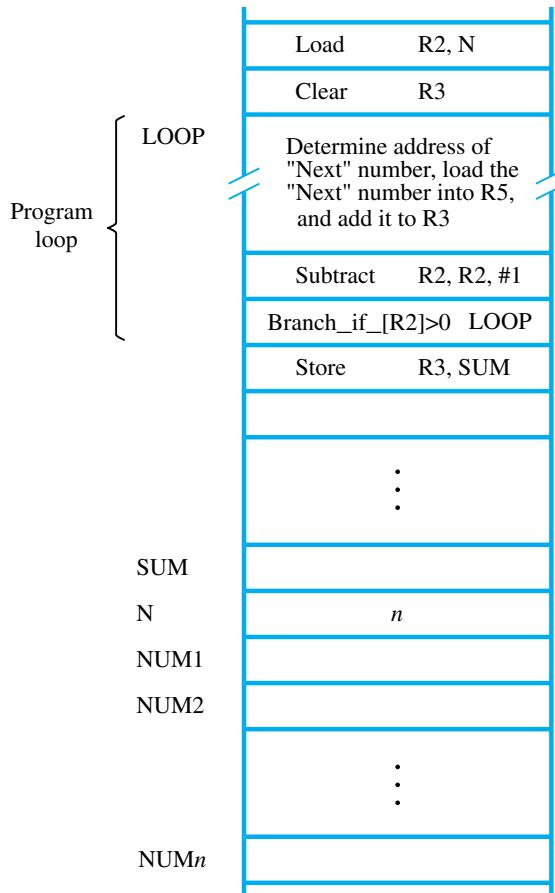
**Figure 2.5** A program for adding  $n$  numbers.

reduces the contents of R2 by 1 each time through the loop. (We will explain the significance of the number sign ‘#’ in Section 2.4.1.) Execution of the loop is repeated as long as the contents of R2 are greater than zero.

We now introduce *branch* instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 2.6, the instruction

Branch\_if\_[R2]>0    LOOP



**Figure 2.6** Using a loop to add  $n$  numbers.

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the  $n$ th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified

requirement. For example, the instruction that implements the action

Branch\_if\_[R4]>[R5] LOOP

may be written in generic assembly language as

Branch\_greater\_than R4, R5, LOOP

or using an actual mnemonic as

BGT R4, R5, LOOP

It compares the contents of registers R4 and R5, without changing the contents of either register. Then, it causes a branch to LOOP if the contents of R4 are greater than the contents of R5.

A different way of implementing branch instructions uses the concept of condition codes, which we will discuss in Section 2.10.2.

### 2.3.7 GENERATING MEMORY ADDRESSES

Let us return to Figure 2.6. The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register,  $R_i$ , is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same. We will discuss these in the next section.

---

## 2.4 ADDRESSING MODES

We have now seen some simple examples of assembly-language programs. In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways that reflect the nature of the information and how it is used. Programmers use *data structures* such as lists and arrays for organizing the data used in computations.

Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures. When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations. The

**Table 2.1** RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R $i$	EA = R $i$
Absolute	LOC	EA = LOC
Register indirect	(R $i$ )	EA = [R $i$ ]
Index	X(R $i$ )	EA = [R $i$ ] + X
Base with index	(R $i$ ,R $j$ )	EA = [R $i$ ] + [R $j$ ]

EA = effective address

Value = a signed number

X = index value

different ways for specifying the locations of instruction operands are known as *addressing modes*. In this section we present the basic addressing modes found in RISC-style processors. A summary is provided in Table 2.1, which also includes the assembler syntax we will use for each mode. The assembler syntax defines the way in which instructions and the addressing modes of their operands are specified; it is discussed in Section 2.5.

### 2.4.1 IMPLEMENTATION OF VARIABLES AND CONSTANTS

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions.

The program in Figure 2.5 uses only two addressing modes to access variables. We access an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

*Register mode*—The operand is the contents of a processor register; the name of the register is given in the instruction.

*Absolute mode*—The operand is in a memory location; the address of this location is given explicitly in the instruction.

Since in a RISC-style processor an instruction must fit in a single word, the number of bits that can be used to give an absolute address is limited, typically to 16 bits if the word length is 32 bits. To generate a 32-bit address, the 16-bit value is usually extended to 32 bits by replicating bit  $b_{15}$  into bit positions  $b_{31-16}$  (as in sign extension). This means that an absolute address can be specified in this manner for only a limited range of the full address space. We will deal with the issue of specifying full 32-bit addresses in Section 2.9. To keep our examples simple, we will assume for now that all addresses of memory locations involved in a program can be specified in 16 bits.

The instruction

Add R4, R2, R3

uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination.

The Absolute mode can represent global variables in a program. A declaration such as

Integer NUM1, NUM2, SUM;

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

The Absolute mode is used in the instruction

Load R2, NUM1

which loads the value in the memory location NUM1 into register R2.

Constants representing data or addresses are also found in almost every computer program. Such constants can be represented in assembly language using the Immediate addressing mode.

*Immediate mode*—The operand is given explicitly in the instruction.

For example, the instruction

Add R4, R6, 200<sub>immediate</sub>

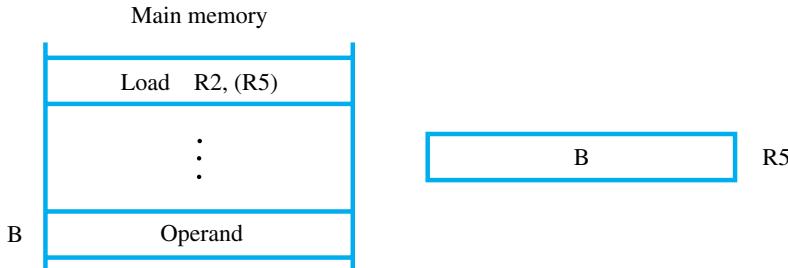
adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

Add R4, R6, #200

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

### 2.4.2 INDIRECTION AND POINTERS

The program in Figure 2.6 requires a capability for modifying the address of the memory operand during each pass through the loop. A good way to provide this capability is to use a processor register to hold the address of the operand. The contents of the register are then changed (incremented) during each pass to provide the address of the next number in the list that has to be accessed. The register acts as a *pointer* to the list, and we say that an item



**Figure 2.7** Register indirect addressing.

in the list is accessed *indirectly* by using the address in the register. The desired capability is provided by the indirect addressing mode.

*Indirect mode*—The effective address of the operand is the contents of a register that is specified in the instruction.

We denote indirection by placing the name of the register given in the instruction in parentheses as illustrated in Figure 2.7 and Table 2.1.

To execute the Load instruction in Figure 2.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

Indirection and the use of pointers are important and powerful concepts in programming. They permit the same code to be used to operate on different data. For example, register R5 in Figure 2.7 serves as a pointer for the Load instruction to load an operand from the memory into register R2. At one time, R5 may point to location B in memory. Later, the program may change the contents of R5 to point to a different location, in which case the same Load instruction will load the value from that location into R2. Thus, a program segment that includes this Load instruction is conveniently reused with only a change in the pointer value.

Let us now return to the program in Figure 2.6 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 2.8. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4. The initialization section of the program loads the counter value  $n$  from memory location N into R2. Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4. Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move R4, #NUM1

---

	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

---

**Figure 2.8** Use of indirect addressing in the program of Figure 2.6.

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as

Add R4, R0, #NUM1

It is often the case that Move is provided as a *pseudoinstruction* for the convenience of programmers, but it is actually implemented using the Add instruction.

The first three instructions in the loop in Figure 2.8 implement the unspecified instruction block starting at LOOP in Figure 2.6. The first time through the loop, the instruction

Load R5, (R4)

fetches the operand at location NUM1 and loads it into R5. The first Add instruction adds this number to the sum in register R3. The second Add instruction adds 4 to the contents of the pointer R4, so that it will contain the address value NUM2 when the Load instruction is executed in the second pass through the loop.

As another example of pointers, consider the C-language statement

A = \*B;

where B is a pointer variable and the '\*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

Load	R2, B
Load	R3, (R2)
Store	R3, A

Indirect addressing through registers is used extensively. The program in Figure 2.8 shows the flexibility it provides.

### 2.4.3 INDEXING AND ARRAYS

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

*Index mode*—The effective address of the operand is generated by adding a constant value to the contents of a register.

For convenience, we will refer to the register used in this mode as the *index register*. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as

$$X(Ri)$$

where X denotes a constant signed integer value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

The contents of the register are not changed in the process of generating the effective address.

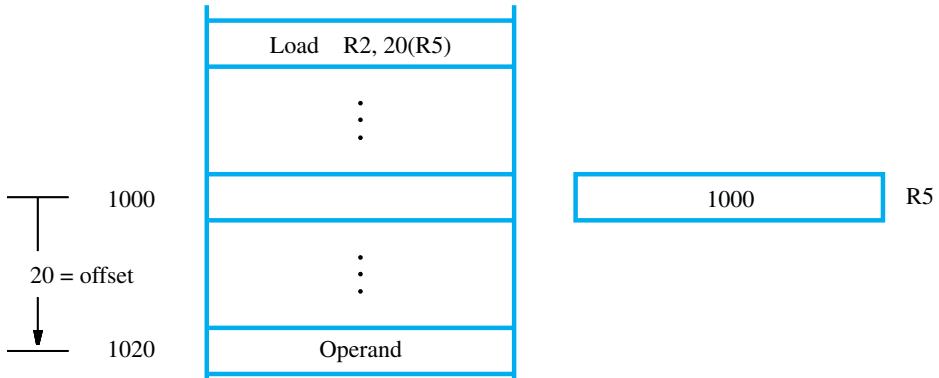
In an assembly-language program, whenever a constant such as the value X is needed, it may be given either as an explicit number or as a symbolic name representing a numerical value. The way in which a symbolic name is associated with a specific numerical value will be discussed in Section 2.5. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is restricted to fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended (see Section 1.4) to the register length before being added to the contents of the register.

Figure 2.9 illustrates two ways of using the Index mode. In Figure 2.9a, the index register, R5, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use is illustrated in Figure 2.9b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is held in a register.

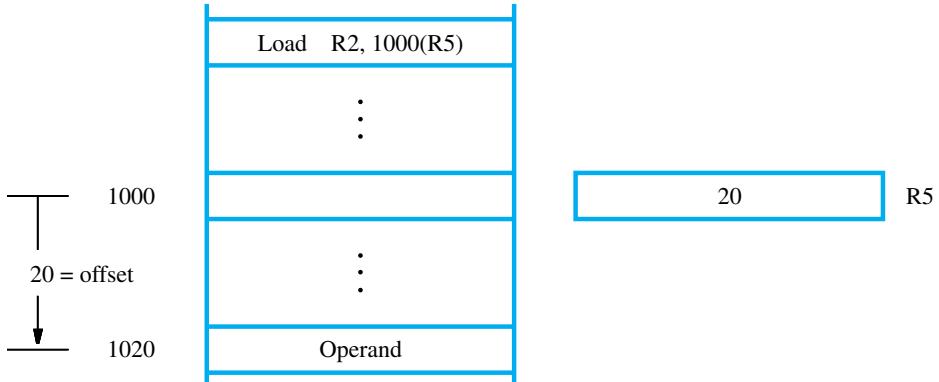
To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.10. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are  $n$  students in the class, and the value  $n$  is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

We should note that the list in Figure 2.10 represents a two-dimensional array having  $n$  rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible



(a) Offset is given as a constant

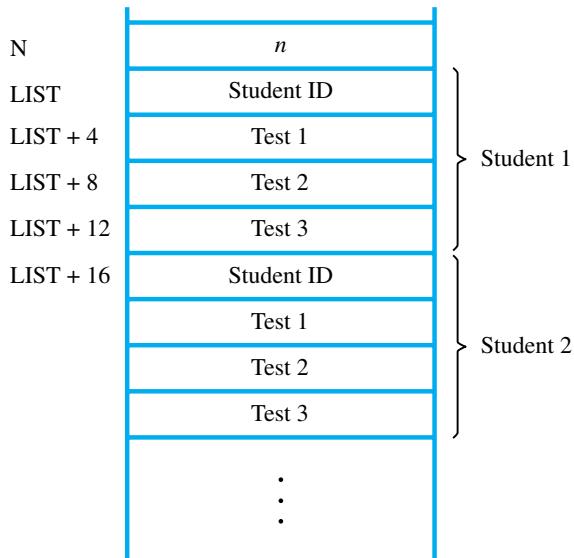


(b) Offset is in the index register

**Figure 2.9** Indexed addressing.

program for this task is given in Figure 2.11. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.9a to access each of the three scores in a student's record. Register R2 is used as the index register. Before the loop is entered, R2 is set to point to the ID location of the first student record which is the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R3, R4, and R5, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R2), 8(R2), and 12(R2). The index register R2 is then incremented by 16 to point to the ID location of the second student. Register R6, initialized to contain the value  $n$ , is decremented by 1 at the end of each pass through the loop. When the contents of R6 reach 0, all student records have been accessed, and



**Figure 2.10** A list of students' marks.

---

Move	R2, #LIST	Get the address LIST.
Clear	R3	
Clear	R4	
Clear	R5	
Load	R6, N	Load the value $n$ .
LOOP: Load	R7, 4(R2)	Add the mark for next student's
Add	R3, R3, R7	Test 1 to the partial sum.
Load	R7, 8(R2)	Add the mark for that student's
Add	R4, R4, R7	Test 2 to the partial sum.
Load	R7, 12(R2)	Add the mark for that student's
Add	R5, R5, R7	Test 3 to the partial sum.
Add	R2, R2, #16	Increment the pointer.
Subtract	R6, R6, #1	Decrement the counter.
Branch_if_[R6]>0	LOOP	Branch back if not finished.
Store	R3, SUM1	Store the total for Test 1.
Store	R4, SUM2	Store the total for Test 2.
Store	R5, SUM3	Store the total for Test 3.

---

**Figure 2.11** Indexed addressing used in accessing test scores in the list in Figure 2.10.

the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R2, are not changed when it is used in the Index addressing mode to access the scores. The contents of R2 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing that uses a register  $R_i$  and a constant offset X. Several variations of this basic form provide for efficient access to memory operands in practical programming situations (although they may not be included in some processors). For example, a second register  $R_j$  may be used to contain the offset X, in which case we can write the Index mode as

$$(R_i, R_j)$$

The effective address is the sum of the contents of registers  $R_i$  and  $R_j$ . The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(R_i, R_j)$$

In this case, the effective address is the sum of the constant X and the contents of registers  $R_i$  and  $R_j$ . This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the  $(R_i, R_j)$  part of the addressing mode.

Finally, we should note that in the basic Index mode

$$X(R_i)$$

if the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as

$$X(R_0)$$

## 2.5 ASSEMBLY LANGUAGE

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Load, Store, Add, and

Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called *mnenomics*, such as LD, ST, ADD, and BR. A shorthand notation is also useful when identifying registers, such as R3 for register 3. Finally, symbols such as LOC may be defined as needed to represent particular memory locations. A complete set of such symbolic names and rules for their use constitutes a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*. The assembler program is one of a collection of utility programs that are a part of the system software of a computer. The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk. At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine-language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*. We will discuss how the assembler program works in Section 2.5.2 and in Chapter 4. First, we present a few aspects of assembly language itself.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower-case letters. In this section, we use capital letters to denote all names and labels in our examples to improve the readability of the text. For example, we write a Store instruction as

ST R2, SUM

The mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer recognizes.

The OP-code mnemonic is followed by at least one blank space or tab character. Then the information that specifies the operands is given. In the Store instruction above, the source operand is in register R2. This information is followed by the specification of the destination operand, separated from the source operand by a comma. The destination operand is in the memory location that has its binary address represented by the name SUM.

Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used. For example, a numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode. The number sign usually denotes an immediate operand. Thus, the instruction

ADD R2, R3, #5

adds the number 5 to the contents of register R3 and puts the result into register R2. The number sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the Immediate addressing mode is indicated in the OP-code mnemonic.

For example, the previous Add instruction may be written as

```
ADDI R2, R3, 5
```

The suffix I in the mnemonic ADDI states that the second source operand is given in the Immediate addressing mode.

Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand. For example, if register R2 contains the address of a number in the memory, then this number can be loaded into register R3 using the instruction

```
LD R3, (R2)
```

### 2.5.1 ASSEMBLER DIRECTIVES

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as

```
TWENTY EQU 20
```

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program. Such statements, called *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program.

To illustrate the use of assembly language further, let us reconsider the program in Figure 2.8. In order to run this program on a computer, it is necessary to write its source code in the required assembly language, specifying all of the information needed to generate the corresponding object program. Suppose that each instruction and each data item occupies one word of memory. Also assume that the memory is byte-addressable and that the word length is 32 bits. Suppose also that the object program is to be loaded in the main memory as shown in Figure 2.12. The figure shows the memory addresses where the machine instructions and the required data items are to be found after the program is loaded for execution. If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

To provide this information, the source program may be written as shown in Figure 2.13. The program begins with the assembler directive, ORIGIN, which tells the assembler program where in the memory to place the instructions that follow. It specifies that the instructions

	100	Load	R2, N
	104	Clear	R3
	108	Move	R4, #NUM1
LOOP	112	Load	R5, (R4)
	116	Add	R3, R3, R5
	120	Add	R4, R4, #4
	124	Subtract	R2, R2, #1
	128	Branch_if_[R2]>0	LOOP
	132	Store	R3, SUM
			⋮
			⋮
SUM	200		
N	204		150
NUM1	208		
NUM2	212		
			⋮
NUM <sub>n</sub>	804		

**Figure 2.12** Memory arrangement for the program in Figure 2.8.

of the object program are to be loaded in the memory starting at address 100. It is followed by the source program instructions written with the appropriate mnemonics and syntax. Note that we use the statement

BGT R2, R0, LOOP

to represent an instruction that performs the operation

Branch\_if\_[R2]>0 LOOP

The second ORIGIN directive tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 200. This is intended to be the location in which the final sum will be stored. A 4-byte space for the sum is reserved by means of the assembler directive RESERVE. The next word, at address 204, has to contain the value 150 which is the number of entries in the list.

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions	LOOP:	LD CLR MOV LD ADD ADD SUB BGT ST	R2, N R3 R4, #NUM1 R5, (R4) R3, R3, R5 R4, R4, #4 R2, R2, #1 R2, R0, LOOP R3, SUM
next instruction			
Assembler directives	SUM: N: NUM1:	ORIGIN RESERVE DATAWORD RESERVE END	200 4 150 600

**Figure 2.13** Assembly language representation for the program in Figure 2.12.

The DATAWORD directive is used to inform the assembler of this requirement. The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data. This directive does not cause any data to be loaded in these locations. Data may be loaded in the memory using an input procedure, as we will explain in Chapter 3. The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

We previously described how the EQU directive can be used to associate a specific value, which may be an address, with a particular name. A different way of associating addresses with names or labels is illustrated in Figure 2.13. Any statement that results in instructions or data being placed in a memory location may be given a memory address label. The assembler automatically assigns the address of that location to the label. For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1. Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200. Whenever SUM is encountered in the program, it will be replaced with this value. Using SUM as a label in

this manner is equivalent to using the assembler directive

```
SUM EQU 200
```

Similarly, the labels N and NUM1 are assigned the values 204 and 208, respectively, because they represent the addresses of the two word locations immediately following the word location with address 200.

Most assembly languages require statements in a source program to be written in the form

Label:	Operation	Operand(s)	Comment
--------	-----------	------------	---------

These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters. The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded. Labels may also be associated with addresses of data items. In Figure 2.13 there are four labels: LOOP, SUM, N, and NUM1.

The Operation field contains an assembler directive or the OP-code mnemonic of the desired instruction. The Operand field contains addressing information for accessing the operands. The Comment field is ignored by the assembler program. It is used for documentation purposes to make the program easier to understand.

We have introduced only the very basic characteristics of assembly languages. These languages differ in detail and complexity from one computer to another.

## 2.5.2 ASSEMBLY AND EXECUTION OF PROGRAMS

A source program written in an assembly language must be assembled into a machine-language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the addresses given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands, and it reserves memory space as requested by RESERVE commands.

A key part of the assembly process is determining the values that replace the names. In some cases, where the value of a name is specified by an EQU directive, this is a straightforward task. In other cases, where a name is defined in the Label field of a given instruction, the value represented by the name is determined by the location of this instruction in the assembled object program. Hence, the assembler must keep track of addresses as it generates the machine code for successive instructions. For example, the names LOOP and SUM in the program of Figure 2.13 will be assigned the values 112 and 200, respectively.

In some cases, the assembler does not directly replace a name representing an address with the actual value of this address. For example, in a branch instruction, the name that specifies the location to which a branch is to be made (the branch target) is not replaced by the actual address. A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter

to the target instruction. The assembler computes this *branch offset*, which can be positive or negative, and puts it into the machine instruction. We will show how branch instructions may be implemented in Section 2.13.

The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a *loader* must already be in the memory. Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored. The assembler usually places this information in a header preceding the object code. Having loaded the object code, the loader starts execution of the object program by branching to the first instruction to be executed, which may be identified by an address label such as START. The assembler places that address in the header of the object code for the loader to use at execution time.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger* program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

In this section, we introduced some important issues in assembly and execution of programs. Chapter 4 provides a more detailed discussion of these issues.

### 2.5.3 NUMBER NOTATION

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

ADDI R2, R3, 93

or as a binary number identified by an assembler-specific prefix symbol such as a percent sign, as in

ADDI R2, R3, %01011101

Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The first ten patterns 0000, 0001, . . . , 1001, referred to as *binary-coded decimal* (BCD), are represented by the digits 0, 1, . . . , 9. The remaining six 4-bit patterns, 1010, 1011, . . . , 1111, are represented by the letters A, B, . . . , F. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by the prefix 0x (as in the C language) or

by a dollar sign prefix. Thus, we would write

ADDI R2, R3, 0x5D

## 2.6 STACKS

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, *last-in-first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the *stack pointer* (SP), is used to point to a particular stack structure called the *processor stack*, whose use will be explained shortly.

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

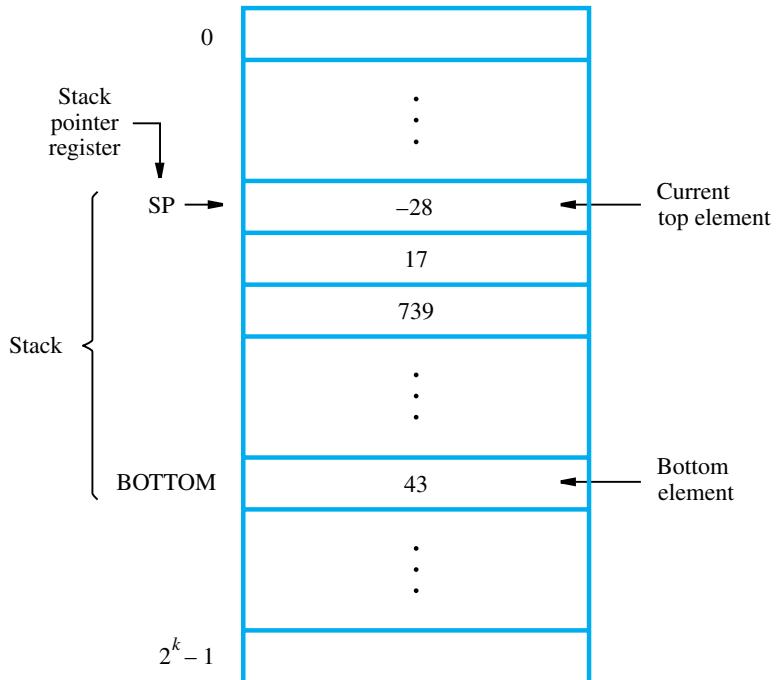
Figure 2.14 shows an example of a stack of word data items. The stack contains numerical values, with 43 at the bottom and –28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

Subtract	SP, SP, #4
Store	R <sub>j</sub> , (SP)

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register R<sub>j</sub>, the Store instruction will place this value on the stack. These two instructions copy the word from R<sub>j</sub> onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as

Load	R <sub>j</sub> , (SP)
Add	SP, SP, #4

These two instructions load (pop) the top value from the stack into register R<sub>j</sub> and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.15 shows the effect of each of these operations on the stack in Figure 2.14.



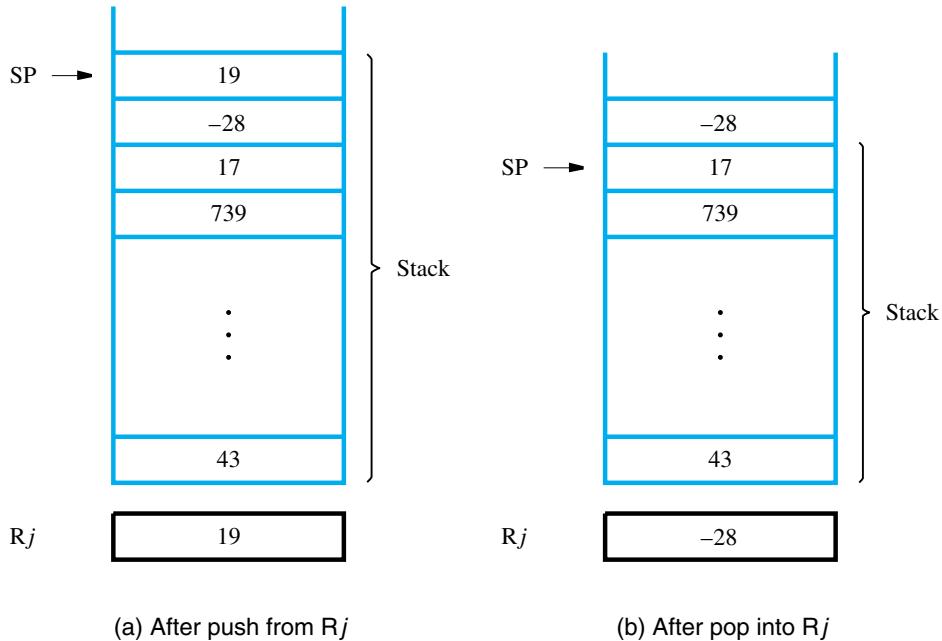
**Figure 2.14** A stack of words in the memory.

## 2.7 SUBROUTINES

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*. For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order.

It is possible to reproduce the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it, and it does so by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling



**Figure 2.15** Effect of stack operations on the stack in Figure 2.14.

program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

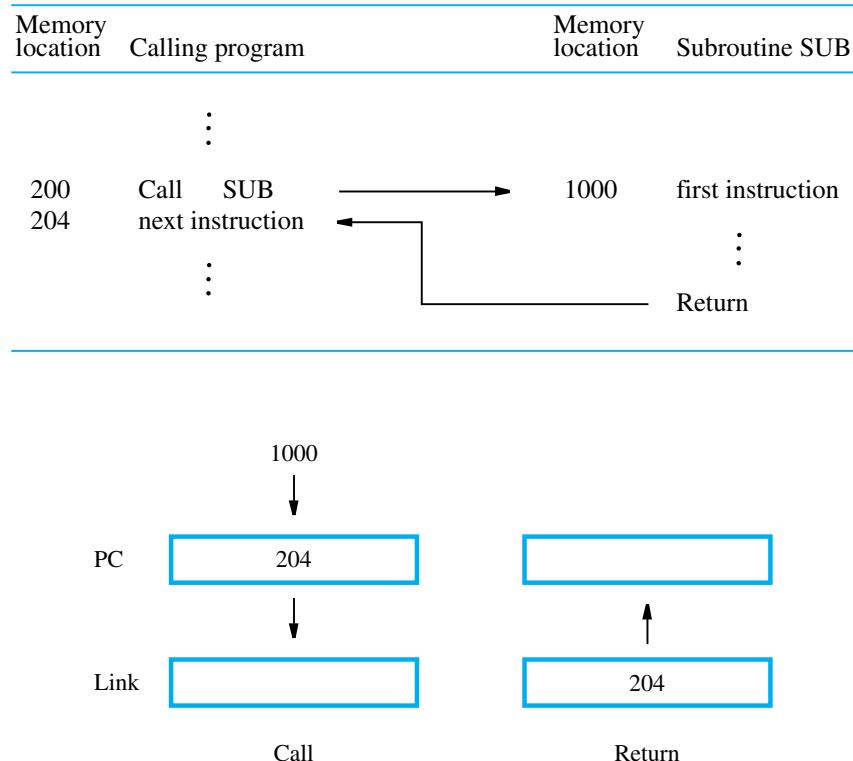
The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

Figure 2.16 illustrates how the PC and the link register are affected by the Call and Return instructions.



**Figure 2.16** Subroutine linkage using a link register.

### 2.7.1 SUBROUTINE NESTING AND THE PROCESSOR STACK

A common programming practice, called *subroutine nesting*, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, overwriting its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Correct sequencing of nested calls is achieved if a given subroutine SUB1 saves the return address currently in the link register on the stack, accessed through the stack pointer, SP, before it calls another subroutine SUB2. Then, prior to executing its own Return instruction, the subroutine SUB1 has to pop the saved return address from the stack and load it into the link register.

## 2.7.2 PARAMETER PASSING

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as *parameter passing*. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack.

Passing parameters through processor registers is straightforward and efficient. Figure 2.17 shows how the program in Figure 2.8 for adding a list of numbers can be implemented as a subroutine, LISTADD, with the parameters passed through registers. The size of the list,  $n$ , contained in memory location N, and the address, NUM1, of the first number, are passed through registers R2 and R4. The sum computed by the subroutine is passed back to the calling program through register R3. The first four instructions in Figure 2.17 constitute the relevant part of the calling program. The first two instructions load  $n$  and NUM1 into

### Calling program

Load	R2, N	Parameter 1 is list size.
Move	R4, #NUM1	Parameter 2 is list location.
Call	LISTADD	Call subroutine.
Store	R3, SUM	Save result.
:		

### Subroutine

LISTADD:	Subtract	SP, SP, #4	Save the contents of
	Store	R5, (SP)	R5 on the stack.
	Clear	R3	Initialize sum to 0.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer by 4.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	
	Load	R5, (SP)	Restore the contents of R5.
	Add	SP, SP, #4	
	Return		Return to calling program.

**Figure 2.17** Program of Figure 2.8 written as a subroutine; parameters passed through registers.

R2 and R4. The Call instruction branches to the subroutine starting at location LISTADD. This instruction also saves the return address (i.e., the address of the Store instruction in the calling program) in the link register. The subroutine computes the sum and places it in R3. After the Return instruction is executed by the subroutine, the sum in R3 is stored in memory location SUM by the calling program.

In addition to registers R2, R3, and R4, which are used for parameter passing, the subroutine also uses R5. Since R5 may be used in the calling program, its contents are saved by pushing them onto the processor stack upon entry to the subroutine and restored before returning to the calling program.

If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine. The processor stack provides a convenient and flexible mechanism for passing an arbitrary number of parameters. Figure 2.18 shows the program of Figure 2.8 rewritten as a subroutine, LISTADD, which uses the processor stack for parameter passing. The address of the first number in the list and the number of entries are pushed onto the processor stack pointed to by register SP. The subroutine is then called. The computed sum is placed on the stack before the return to the calling program.

Figure 2.19 shows the stack entries for this example. Assume that before the subroutine is called, the top of the stack is at level 1. The calling program pushes the address NUM1 and the value  $n$  onto the stack and calls subroutine LISTADD. The top of the stack is now at level 2. The subroutine uses four registers while it is being executed. Since these registers may contain valid data that belong to the calling program, their contents should be saved at the beginning of the subroutine by pushing them onto the stack. The top of the stack is now at level 3. The subroutine accesses the parameters  $n$  and NUM1 from the stack using indexed addressing with offset values relative to the new top of the stack (level 3). Note that it does not change the stack pointer because valid data items are still at the top of the stack. The value  $n$  is loaded into R2 as the initial value of the count, and the address NUM1 is loaded into R4, which is used as a pointer to scan the list entries.

At the end of the computation, register R3 contains the sum. Before the subroutine returns to the calling program, the contents of R3 are inserted into the stack, replacing the parameter NUM1, which is no longer needed. Then the contents of the four registers used by the subroutine are restored from the stack. Also, the stack pointer is incremented to point to the top of the stack that existed when the subroutine was called, namely the parameter  $n$  at level 2. After the subroutine returns, the calling program stores the result in location SUM and lowers the top of the stack to its original level by incrementing the SP by 8.

Observe that for subroutine LISTADD in Figure 2.18, we did not use a pair of instructions

Subtract	SP, SP, #4
Store	R $j$ , (SP)

to push the contents of each register on the stack. Since we have to save four registers, this would require eight instructions. We needed only five instructions by adjusting SP immediately to point to the top of stack that will be in effect once all four registers are saved. Then, we used the Index mode to store the contents of registers. We used the same optimization when restoring the registers before returning from the subroutine.

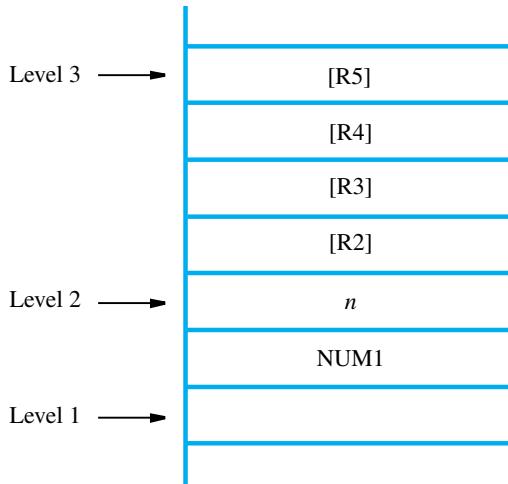
---

Assume top of stack is at level 1 in Figure 2.19.

	Move	R2, #NUM1	Push parameters onto stack.
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Load	R2, N	
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Call	LISTADD	Call subroutine (top of stack is at level 2).
	Load	R2, 4(SP)	Get the result from the stack
	Store	R2, SUM	and save it in SUM.
	Add	SP, SP, #8	Restore top of stack (top of stack is at level 1).
	:		
LISTADD:	Subtract	SP, SP, #16	Save registers
	Store	R2, 12(SP)	
	Store	R3, 8(SP)	
	Store	R4, 4(SP)	
	Store	R5, (SP)	(top of stack is at level 3).
	Load	R2, 16(SP)	Initialize counter to $n$ .
	Load	R4, 20(SP)	Initialize pointer to the list.
	Clear	R3	Initialize sum to 0.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer by 4.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	
	Store	R3, 20(SP)	Put result in the stack.
	Load	R5, (SP)	Restore registers.
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Add	SP, SP, #16	(top of stack is at level 2).
	Return		Return to calling program.

---

**Figure 2.18** Program of Figure 2.8 written as a subroutine; parameters passed on the stack.



**Figure 2.19** Stack contents for the program in Figure 2.18.

We should also note that some computers have special instructions for loading and storing multiple registers. For example, the four registers in Figure 2.18 may be saved on the stack by using the instruction

StoreMultiple R2–R5, -(SP)

The source registers are specified by the range R2–R5. The notation  $-(SP)$  specifies that the stack pointer must be adjusted accordingly. The minus sign in front indicates that SP must be decremented (by 4) before the contents of each register are placed on the stack.

Similarly, the instruction

LoadMultiple R2–R5, (SP)+

will load registers R2, R3, R4, and R5, in reverse order, with the values that were saved on the stack. The notation  $(SP)+$  indicates that the stack pointer must be incremented (by 4) after each value has been loaded into the corresponding register. We will discuss the addressing modes denoted by  $-(SP)$  and  $(SP)+$  in more detail in Section 2.9.1.

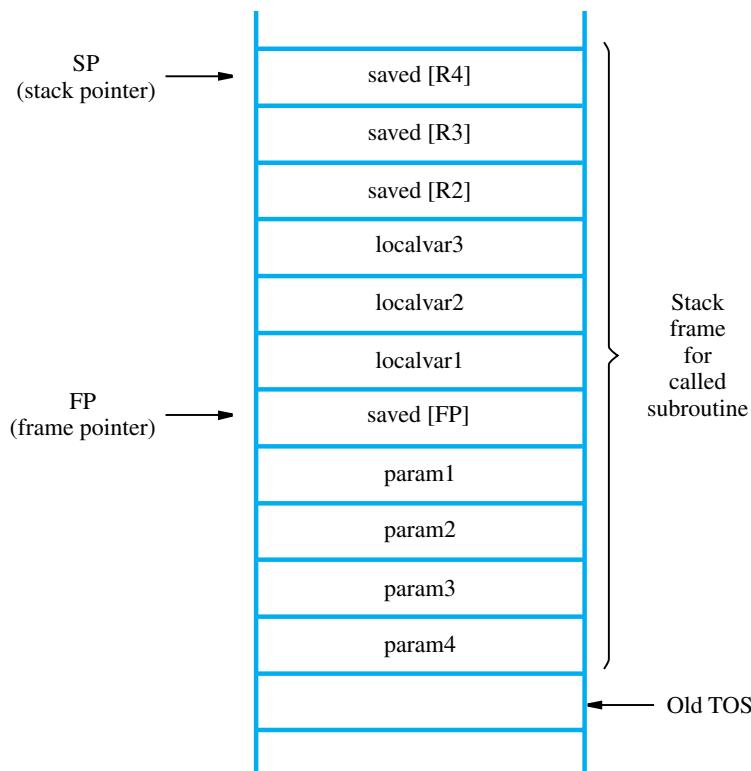
### Parameter Passing by Value and by Reference

Note the nature of the two parameters, NUM1 and *n*, passed to the subroutines in Figures 2.17 and 2.18. The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called *passing by reference*. The second parameter is *passed by value*, that is, the actual number of entries, *n*, is passed to the subroutine.

### 2.7.3 THE STACK FRAME

Now, observe how space is used in the stack in the example in Figures 2.18 and 2.19. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program. Such space is called a *stack frame*. If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack.

Figure 2.20 shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the *frame pointer* (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the subroutine. When nested subroutines are used, the stack frame of the calling subroutine would also include the return address, as we will see in the example that follows.



**Figure 2.20** A subroutine stack frame example.

With the FP register pointing to the location just above the stored parameters, as shown in Figure 2.20, we can easily access the parameters and the local variables by using the Index addressing mode. The parameters can be accessed by using addresses 4(FP), 8(FP), . . . . The local variables can be accessed by using addresses −4(FP), −8(FP), . . . . The contents of FP remain fixed throughout the execution of the subroutine, unlike the stack pointer SP, which must always point to the current top element in the stack.

Now let us discuss how the pointers SP and FP are manipulated as the stack frame is allocated, used, and deallocated for a particular invocation of a subroutine. We begin by assuming that SP points to the old top-of-stack (TOS) element in Figure 2.20. Before the subroutine is called, the calling program pushes the four parameters onto the stack. Then the Call instruction is executed. At this time, SP points to the last parameter that was pushed on the stack. If the subroutine is to use the frame pointer, it should first save the contents of FP by pushing them on the stack, because FP is usually a general-purpose register and it may contain information of use to the calling program. Then, the contents of SP, which now points to the saved value of FP, are copied into FP.

Thus, the first three instructions executed in the subroutine are

Subtract	SP, SP, #4
Store	FP, (SP)
Move	FP, SP

The Move instruction copies the contents of SP into FP. After these instructions are executed, both SP and FP point to the saved FP contents. Space for the three local variables is now allocated on the stack by executing the instruction

Subtract SP, SP, #12

Finally, the contents of processor registers R2, R3, and R4 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in Figure 2.20.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R4, R3, and R2 back into those registers, deallocates the local variables from the stack frame by executing the instruction

Add SP, SP, #12

and pops the saved old value of FP back into FP. At this point, SP points to the last parameter that was placed on the stack. Next, the Return instruction is executed, transferring control back to the calling program.

The calling program is responsible for deallocating the parameters from the stack frame, some of which may be results passed back by the subroutine. After deallocation of the parameters, the stack pointer points to the old TOS, and we are back to where we started.

### Stack Frames for Nested Subroutines

When nested subroutines are used, it is necessary to ensure that the return addresses are properly saved. When a calling program calls a subroutine, say SUB1, the return address is saved in the link register. Now, if SUB1 calls another subroutine, SUB2, it must save the

current contents of the link register before it makes the call to SUB2. The appropriate place for saving this return address is within the stack frame for SUB1. If SUB2 then calls SUB3, it must save the current contents of the link register within the stack frame associated with SUB2, and so on.

An example of a main program calling a first subroutine SUB1, which then calls a second subroutine SUB2, is shown in Figure 2.21. The stack frames corresponding to these two nested subroutines are shown in Figure 2.22. All parameters involved in this example are passed on the stack. The two figures only show the flow of control and data among the main program and the two subroutines. The actual computations are not shown.

The flow of execution is as follows. The main program pushes the two parameters *param2* and *param1* onto the stack, in that order, and then calls SUB1. This first subroutine is responsible for computing a single result and passing it back to the main program on the stack. During the course of its computations, SUB1 calls the second subroutine, SUB2, in order to perform some other subtask. SUB1 passes a single parameter *param3* to SUB2, and the result is passed back to it via the same location on the stack. After SUB2 executes its Return instruction, SUB1 loads this result into register R4. SUB1 then continues its computations and eventually passes the required answer back to the main program on the stack. When SUB1 executes its return to the main program, the main program stores this answer in memory location RESULT, restores the stack level, then continues with its computations at the next instruction at address 2040. Note how the return address to the calling program, 2028, is stored within the stack frame for SUB1 in Figure 2.22.

The comments in Figure 2.21 provide the details of how this flow of execution is managed. The first action performed by each subroutine is to save on the stack the contents of all registers used in the subroutine, including the frame pointer and link register (if needed). This is followed by initializing the frame pointer. SUB1 uses four registers, R2 to R5, and SUB2 uses two registers, R2 and R3. These registers, the frame pointer, and the link register in the case of SUB1, are restored just before the Return instructions are executed.

The Index addressing mode involving the frame pointer register FP is used to load parameters from the stack and place answers back on the stack. The byte offsets used in these operations are always 4, 8, . . . , as discussed for the general stack frame in Figure 2.20. Finally, note that each calling routine is responsible for removing its own parameters from the stack. This is done by the Add instructions, which lower the top of the stack.

---

## 2.8 ADDITIONAL INSTRUCTIONS

So far, we have introduced the following instructions: Load, Store, Move, Clear, Add, Subtract, Branch, Call, and Return. These instructions, along with the addressing modes in Table 2.1, have allowed us to write programs to illustrate machine instruction sequencing, including branching and subroutine linkage. In this section we introduce a few more instructions that are found in most instruction sets.

	Memory location	Instructions	Comments
<b>Main program</b>			
		:	
2000	Load	R2, PARAM2	Place parameters on stack.
2004	Subtract	SP, SP, #4	
2008	Store	R2, (SP)	
2012	Load	R2, PARAM1	
2016	Subtract	SP, SP, #4	
2020	Store	R2, (SP)	
2024	Call	SUB1	Call the subroutine.
2028	Load	R2, (SP)	Store result.
2032	Store	R2, RESULT	
2036	Add	SP, SP, #8	Restore stack level.
2040		next instruction	
		:	
<b>First subroutine</b>			
2100	SUB1:	Subtract	SP, SP, #24 Save registers.
2104	Store	LINK_reg, 20(SP)	
2108	Store	FP, 16(SP)	
2112	Store	R2, 12(SP)	
2116	Store	R3, 8(SP)	
2120	Store	R4, 4(SP)	
2124	Store	R5, (SP)	
2128	Add	FP, SP, #16 Initialize the frame pointer.	
2132	Load	R2, 8(FP)	Get first parameter.
2136	Load	R3, 12(FP)	Get second parameter.
		:	
	Load	R4, PARAM3 Place a parameter on stack.	
	Subtract	SP, SP, #4	
	Store	R4, (SP)	
	Call	SUB2	
	Load	R4, (SP) Get result from SUB2.	
	Add	SP, SP, #4	
		:	
	Store	R5, 8(FP) Place answer on stack.	
	Load	R5, (SP) Restore registers.	
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Load	FP, 16(SP)	
	Load	LINK_reg, 20(SP)	
	Add	SP, SP, #24	
	Return		Return to Main program.

...continued in part b.

**Figure 2.21** Nested subroutines (part a).

Memory location	Instructions		Comments
<b>Second subroutine</b>			
3000	SUB2:	Subtract SP, SP, #12	Save registers.
3004	Store	FP, 8(SP)	
	Store	R2, 4(SP)	
	Store	R3, (SP)	
	Add	FP, SP, #8	Initialize the frame pointer.
	Load	R2, 4(FP)	Get the parameter.
	:		
	Store	R3, 4(FP)	Place SUB2 result on stack.
	Load	R3, (SP)	Restore registers.
	Load	R2, 4(SP)	
	Load	FP, 8(SP)	
	Add	SP, SP, #12	
	Return		Return to Subroutine 1.

**Figure 2.21** Nested subroutines (part b).

### 2.8.1 LOGIC INSTRUCTIONS

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described in Appendix A. It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

And R4, R2, R3

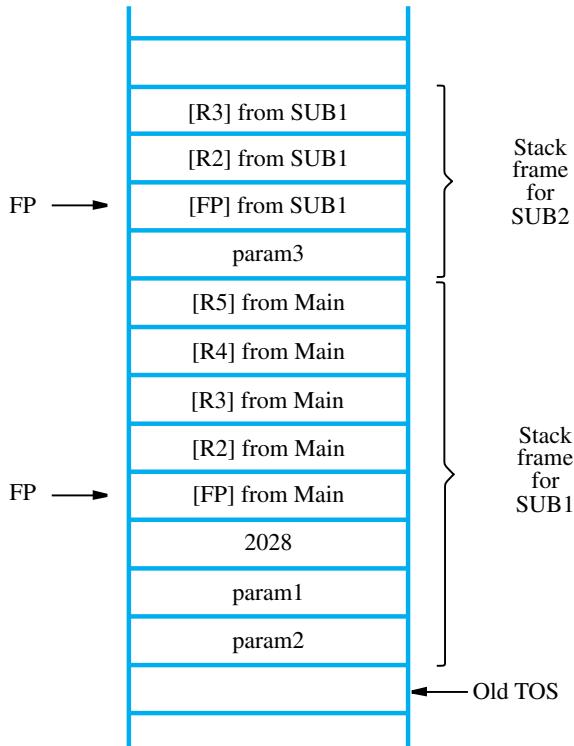
computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4. An immediate form of this instruction may be

And R4, R2, #Value

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

Consider the following application for this logic instruction. Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. If it is, then a conditional branch to FOUNDZ is to be made. From Table 1.1 in Chapter 1, we find that the ASCII code for Z is 01011010, which is expressed in hexadecimal notation as 5A. The three-instruction sequence

And	R2, R2, #0xFF
Move	R3, #0x5A
Branch_if_[R2]=[R3]	FOUNDZ



**Figure 2.22** Stack frames for Figure 2.21.

implements the desired action. The And instruction clears all bits in the leftmost three character positions of R2 to zero, leaving the rightmost character unchanged. This is the result of using an immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the left. The Move instruction loads the hex value 5A into R3. Since both R2 and R3 have 0s in the leftmost 24 bits, the Branch instruction compares the remaining character at the right end of R2 with the binary representation for the character Z, and causes a branch to FOUNDZ if there is a match.

### 2.8.2 SHIFT AND ROTATE INSTRUCTIONS

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a signed number, we use an arithmetic shift, which preserves the sign of the number.

#### Logical Shifts

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions

specified in a count operand contained in the instruction. The general form of a Logical-shift-left instruction is

LShiftL  $R_i, R_j, \text{count}$

which shifts the contents of register  $R_j$  left by a number of bit positions given by the count operand, and places the result in register  $R_i$ , without changing the contents of  $R_j$ . The count operand may be given as an immediate operand, or it may be contained in a processor register. To complete the description of the shift left operation, we need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. Vacated positions are filled with zeros. In computers that do not use condition code flags, the bits shifted out are simply dropped. In computers that use condition code flags, which will be discussed in Section 2.10.2, these bits are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word. Figure 2.23a shows an example of shifting the contents of register R3 left by two bit positions. The Logical-shift-right instruction, LShiftR, works in the same manner except that it shifts to the right. Figure 2.23b illustrates this operation.

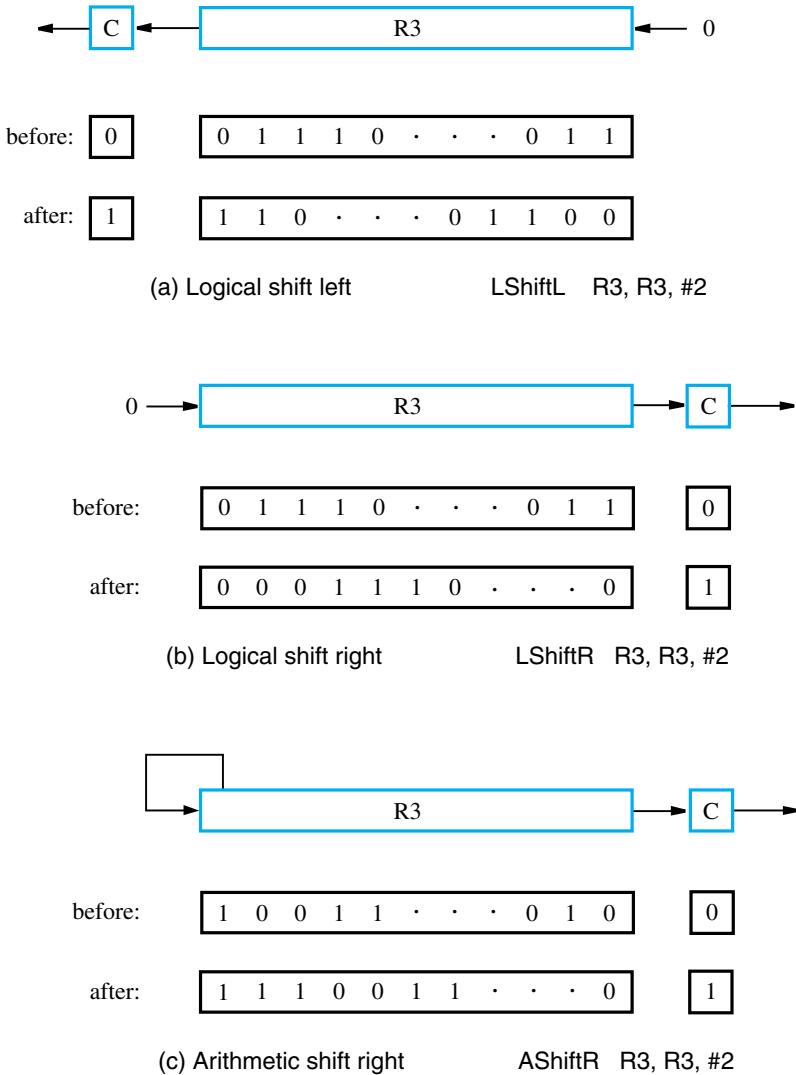
### Digit-Packing Example

Consider the following short task that illustrates the use of both shift operations and logic operations. Suppose that two decimal digits represented in ASCII code are located in the memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in *packed-BCD* format. Table 1.1 in Chapter 1 shows that the rightmost four bits of the ASCII code for a decimal digit correspond to the BCD code for the digit. Hence, the required task is to extract the low-order four bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED.

The instruction sequence shown in Figure 2.24 accomplishes the task using register R2 as a pointer to the ASCII characters in memory, and using registers R3 and R4 to develop the BCD digit codes. The program uses the LoadByte instruction, which loads a byte from the memory into the rightmost eight bit positions of a 32-bit processor register and clears the remaining higher-order bits to zero. The StoreByte instruction writes the rightmost byte in the source register into the specified destination location, but does not affect any other byte locations. The value 0xF in the And instruction is used to clear to zero all but the four rightmost bits in R4. Note that the immediate source operand is written as 0xF, which, interpreted as a 32-bit pattern, has 28 zeros in the most-significant bit positions.

### Arithmetic Shifts

In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number. A study of the 2's-complement binary number representation in Figure 1.3 reveals that shifting a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost when shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in



**Figure 2.23** Logical and arithmetic shift instructions.

bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmetic-shift-right instruction, AShiftR, is shown in Figure 2.23c. The Arithmetic-shift-left is exactly the same as the Logical-shift-left.

### Rotate Operations

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. For situations where it is desirable to preserve all of the bits, rotate instructions may be used instead. These are instructions that

---

Move	R2, #LOC	R2 points to data.
LoadByte	R3, (R2)	Load first byte into R3.
LShiftL	R3, R3, #4	Shift left by 4 bit positions.
Add	R2, R2, #1	Increment the pointer.
LoadByte	R4, (R2)	Load second byte into R4.
And	R4, R4, #0xF	Clear high-order bits to zero.
Or	R3, R3, R4	Concatenate the BCD digits.
StoreByte	R3, PACKED	Store the result.

---

**Figure 2.24** A routine that packs two BCD digits into a byte.

move the bits shifted out of one end of the operand into the other end. Two versions of both the Rotate-left and Rotate-right instructions are often provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. Figure 2.25 shows the left and right rotate operations with and without the C flag being included in the rotation. Note that when the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register. The OP codes RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations.

### 2.8.3 MULTIPLICATION AND DIVISION

Two signed integers can be multiplied or divided by machine instructions with the same format as we saw earlier for an Add instruction. The instruction

Multiply    Rk, Ri, Rj

performs the operation

$$Rk \leftarrow [Ri] \times [Rj]$$

The product of two  $n$ -bit numbers can be as large as  $2n$  bits. Therefore, the answer will not necessarily fit into register Rk. A number of instruction sets have a Multiply instruction that computes the low-order  $n$  bits of the product and places it in register Rk, as indicated. This is sufficient if it is known that all products in some particular application task will fit into  $n$  bits. To accommodate the general  $2n$ -bit product case, some processors produce the product in two registers, usually adjacent registers Rk and R(k + 1), with the high-order half being placed in register R(k + 1).

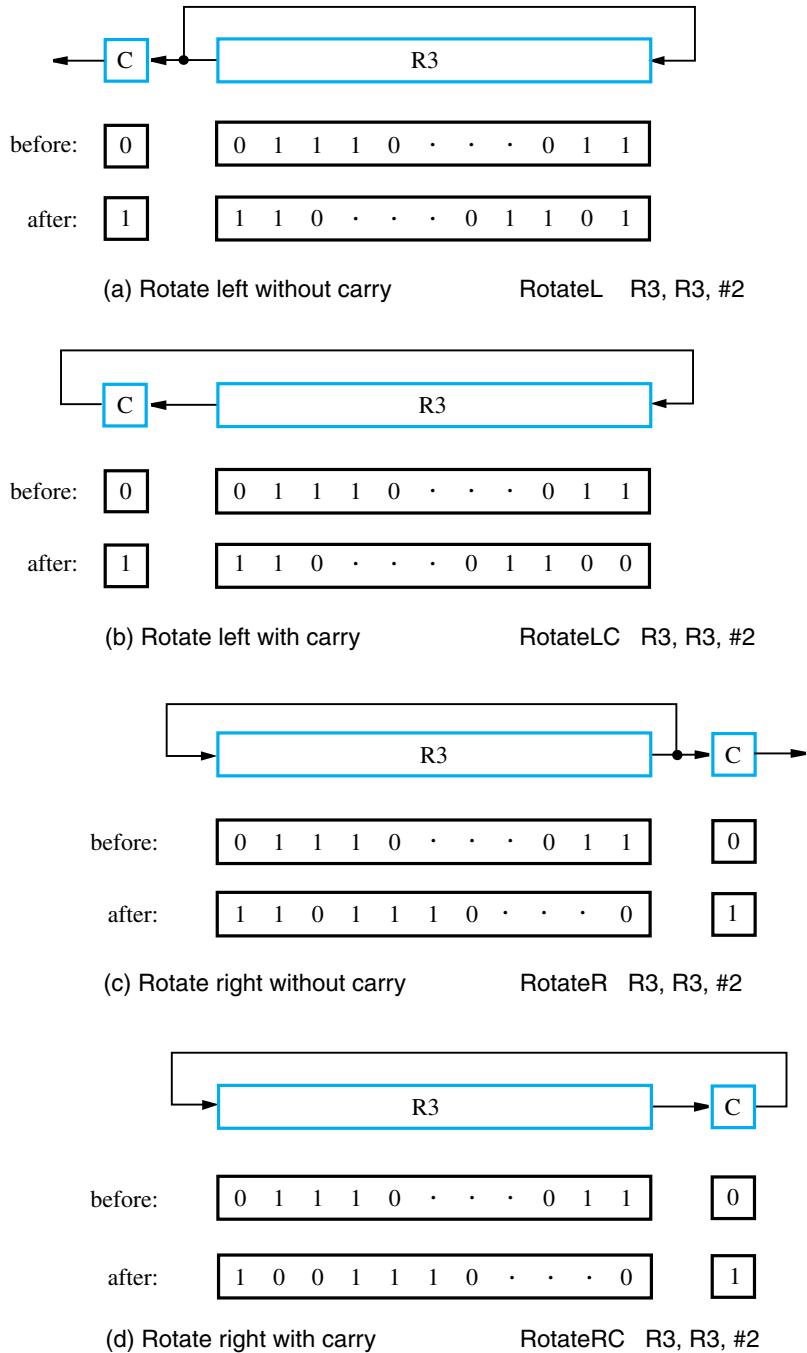
An instruction set may also provide a signed integer Divide instruction

Divide    Rk, Ri, Rj

which performs the operation

$$Rk \leftarrow [Rj]/[Ri]$$

placing the quotient in Rk. The remainder may be placed in R(k + 1), or it may be lost.



**Figure 2.25** Rotate instructions.

Computers that do not have Multiply and Divide instructions can perform these and other arithmetic operations by using sequences of more basic instructions such as Add, Subtract, Shift, and Rotate. This will become more apparent when we describe the implementation of arithmetic operations in Chapter 9.

---

## 2.9 DEALING WITH 32-BIT IMMEDIATE VALUES

In the discussion of addressing modes, in Section 2.4.1, we raised the question of how a 32-bit value that represents a constant or a memory address can be loaded into a processor register. The Immediate and Absolute modes in a RISC-style processor restrict the operand size to 16 bits. Therefore, a 32-bit value cannot be given explicitly in a single instruction that must fit in a 32-bit word.

A possible solution is to use two instructions for this purpose. One approach found in RISC-style processors uses instructions that perform two different logical-OR operations. The instruction

Or Rdst, Rsrc, #Value

extends the 16-bit immediate operand by placing zeros into the high-order bit positions to form a 32-bit value, which is then ORed with the contents of register Rsrc. If Rsrc contains zero, then Rdst will just be loaded with the extended 32-bit value. Another instruction

OrHigh Rdst, Rsrc, #Value

forms a 32-bit value by taking the 16-bit immediate operand as the high-order bits and appending zeros as the low-order bits. This value is then ORed with the contents of Rsrc. Using these instructions, and assuming that R0 contains the value 0, we can load the 32-bit value 0x20004FF0 into register R2 as follows:

OrHigh R2, R0, #0x2000  
Or R2, R2, #0x4FF0

To make it easier to write programs, a RISC-style instruction set may include pseudoinstructions that indicate an action that requires more than one machine instruction. Such pseudoinstructions are replaced with the corresponding machine-instruction sequence by the assembler program. For example, the pseudoinstruction

MoveImmediateAddress R2, LOC

could be used to load a 32-bit address represented by the symbol LOC into register R2. In the assembled program, it would be replaced with two instructions using 16-bit values as shown above.

An alternative to using two instructions to load a 32-bit address into a register is to use more than one word per instruction. In that case, a two-word instruction could give the OP code and register specification in the first word, and include a 32-bit value in the second word. This is the approach found in CISC-style processors.

Finally, note that in the previous sections we always assumed that single Load and Store instructions can be used to access memory locations represented by symbolic names. This makes the example programs simpler and easier to read. The programs will run correctly if the required memory addresses can be specified in 16 bits. If longer addresses are involved, then the approach described above to construct 32-bit addresses must be used.

## 2.10 CISC INSTRUCTION SETS

In preceding sections, we introduced the RISC style of instruction sets. Now we will examine some important characteristics of *Complex Instruction Set Computers* (CISC).

One key difference is that CISC instruction sets are not constrained to the *load/store architecture*, in which arithmetic and logic operations can be performed only on operands that are in processor registers. Another key difference is that instructions do not necessarily have to fit into a single word. Some instructions may occupy a single word, but others may span multiple words.

Instructions in modern CISC processors typically do not use a three-address format. Most arithmetic and logic instructions use the *two-address* format

Operation    destination, source

An Add instruction of this type is

Add    B, A

which performs the operation  $B \leftarrow [A] + [B]$  on memory operands. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that memory location B is both a source and a destination.

Consider again the task of adding two numbers

$C = A + B$

where all three operands may be in memory locations. Obviously, this cannot be done with a single two-address instruction. The task can be performed by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move    C, B

which performs the operation  $C \leftarrow [B]$ , leaving the contents of location B unchanged. The operation  $C \leftarrow [A] + [B]$  can now be performed by the two-instruction sequence

Move	C, B
Add	C, A

Observe that by using this sequence of instructions the contents of neither A nor B locations are overwritten.

In some CISC processors one operand may be in the memory but the other must be in a register. In this case, the instruction sequence for the required task would be

Move	R <sub>i</sub> , A
Add	R <sub>i</sub> , B
Move	C, R <sub>i</sub>

The general form of the Move instruction is

Move destination, source

where both the source and destination may be either a memory location or a processor register. The Move instruction includes the functionality of the Load and Store instructions we used previously in the discussion of RISC-style processors. In the Load instruction, the source is a memory location and the destination is a processor register. In the Store instruction, the source is a register and the destination is a memory location. While Load and Store instructions are restricted to moving operands between memory and processor registers, the Move instruction has a wider scope. It can be used to move immediate operands and to transfer operands between two memory locations or between two registers.

## 2.10.1 ADDITIONAL ADDRESSING MODES

Most CISC processors have all of the five basic addressing modes—Immediate, Register, Absolute, Indirect, and Index. Three additional addressing modes are often found in CISC processors.

### Autoincrement and Autodecrement Modes

There are two modes that are particularly convenient for accessing data items in successive locations in the memory and for implementation of stacks.

*Autoincrement mode*—The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory.

We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

$(R_i) +$

To access successive words in a byte-addressable memory with a 32-bit word length, the increment amount must be 4. Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands. Since the size of the operand is usually specified as part of the operation code of an instruction, it is sufficient to indicate the Autoincrement mode as  $(R_i) +$ .

As a companion for the Autoincrement mode, another useful mode accesses the memory locations in the reverse order:

*Autodecrement mode*—The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

$-(R_i)$

In this mode, operands are accessed in descending address order.

The reader may wonder why the address is decremented before it is used in the Autodecrement mode, and incremented after it is used in the Autoincrement mode. The main reason for this is to make it easy to use these modes together to implement a stack structure. Instead of needing two instructions

Subtract	SP, #4
Move	(SP), NEWITEM

to push a new item on the stack, we can use just one instruction

Move  $-(SP)$ , NEWITEM

Similarly, instead of needing two instructions

Move	ITEM, (SP)
Add	SP, #4

to pop an item from the stack, we can use just

Move ITEM, (SP)+

### Relative Mode

We have defined the Index mode by using general-purpose processor registers. Some computers have a version of this mode in which the program counter, PC, is used instead of a general-purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified *relative* to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

*Relative mode*—The effective address is determined by the Index mode using the program counter in place of the general-purpose register  $R_i$ .

## 2.10.2 CONDITION CODES

Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero. The processor can maintain the information about these results for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called *condition code flags*. These flags are usually grouped together in a special processor register called the *condition code register* or *status register*. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The N and Z flags record whether the result of an arithmetic or logic operation is negative or zero. In some computers, they may also be affected by the value of the operand of a Move instruction. This makes it possible for a later conditional branch instruction to cause a branch based on the sign and value of the operand that was moved. Some computers also provide a special Test instruction that examines a value in a register or in the memory without modifying it, and sets or clears the N and Z flags accordingly.

The V flag indicates whether overflow has taken place. As explained in Section 1.4, overflow occurs when the result of an arithmetic operation is outside the range of values that can be represented by the number of bits available for the operands. The processor sets the V flag to allow the programmer to test whether overflow has occurred and branch to an appropriate routine that deals with the problem. Instructions such as Branch\_if\_overflow are usually provided for this purpose.

The C flag is set to 1 if a carry occurs from the most-significant bit position during an arithmetic operation. This flag makes it possible to perform arithmetic operations on operands that are longer than the word length of the processor. Such operations are used in multiple-precision arithmetic, which is discussed in Chapter 9.

Consider the Branch instruction in Figure 2.6. If condition codes are used, then the Subtract instruction would cause both N and Z flags to be cleared to 0 if the contents of register R2 are still greater than 0. The desired branching could be specified simply as

Branch>0    LOOP

without indicating the register involved in the test. This instruction causes a branch if neither N nor Z is 1, that is, if the result produced by the Subtract instruction is neither negative nor equal to zero. Many conditional branch instructions are provided in the instruction set of a computer to enable a variety of conditions to be tested. The conditions are defined as logic expressions involving the condition code flags.

To illustrate the use of condition codes, consider again the program in Figure 2.8, which adds a list of numbers using RISC-style instructions. Using a CISC-style instruction set, this task can be implemented with fewer instructions, as shown in Figure 2.26. The

---

	Move	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Load address of the first number.
LOOP:	Add	R3, (R4)+	Add the next number to sum.
	Subtract	R2, #1	Decrement the counter.
	Branch>0	LOOP	Loop back if not finished.
	Move	SUM, R3	Store the final sum.

---

**Figure 2.26** A CISC version of the program of Figure 2.8.

Add instruction uses the pointer register (R4) to access successive numbers in the list and add them to the sum in register R3. After accessing the source operand, the processor automatically increments the pointer, because the Autoincrement addressing mode is used to specify the source operand. The Subtract instruction sets the condition codes, which are then used by the Branch instruction.

## 2.11 RISC AND CISC STYLES

RISC and CISC are two different styles of instruction sets. We introduced RISC first because it is simpler and easier to understand. Having looked at some basic features of both styles, we should summarize their main characteristics.

RISC style is characterized by:

- Simple addressing modes
- All instructions fitting in a single word
- Fewer instructions in the instruction set, as a consequence of simple addressing modes
- Arithmetic and logic operations that can be performed only on operands in processor registers
- Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
- Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining which is presented in Chapter 6
- Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

CISC style is characterized by:

- More complex addressing modes
- More complex instructions, where an instruction may span multiple words

- Many instructions that implement complex tasks
- Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers
- Transfers from one memory location to another by using a single Move instruction
- Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

Before the 1970s, all computers were of CISC type. An important objective was to simplify the development of software by making the hardware capable of performing fairly complex tasks, that is, to move the complexity from the software level to the hardware level. This is conducive to making programs simpler and shorter, which was important when computer memory was smaller and more expensive to provide. Today, memory is inexpensive and most computers have large amounts of it.

RISC-style designs emerged as an attempt to achieve very high performance by making the hardware very simple, so that instructions can be executed very quickly in pipelined fashion as will be discussed in Chapter 6. This results in moving complexity from the hardware level to the software level. Sophisticated compilers were developed to optimize the code consisting of simple instructions. The size of the code became less important as memory capacities increased.

While the RISC and CISC styles seem to define two significantly different approaches, today's processors often exhibit what may seem to be a compromise between these approaches. For example, it is attractive to add some non-RISC instructions to a RISC processor in order to reduce the number of instructions executed, as long as the execution of these new instructions is fast. We will deal with the performance issues in detail in Chapter 6 where we discuss the concept of pipelining.

---

## 2.12 EXAMPLE PROGRAMS

In this section we present two examples that further illustrate the use of machine instructions. The examples are representative of numeric and nonnumeric applications.

### 2.12.1 VECTOR DOT PRODUCT PROGRAM

The first example is a numerical application that is an extension of previous programs for adding numbers. In calculations that involve vectors and matrices, it is often necessary to compute the dot product of two vectors. Let  $A$  and  $B$  be two vectors of length  $n$ . Their dot product is defined as

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

Figures 2.27 and 2.28 show RISC- and CISC-style programs for computing the dot product and storing it in memory location DOTPROD. The first elements of each vector,  $A(0)$  and

---

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

---

**Figure 2.27** A RISC-style program for computing the dot product of two vectors.

---

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Move	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Move	R6, (R2)+	Compute the product of
	Multiply	R6, (R3)+	next components.
	Add	R5, R6	Add to previous sum.
	Subtract	R4, #1	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	DOTPROD, R5	Store dot product in memory.

---

**Figure 2.28** A CISC-style program for computing the dot product of two vectors.

$B(0)$ , are stored at memory locations AVEC and BVEC, with the remaining elements in the following word locations.

The task of accumulating a sum of products occurs in many signal-processing applications. In this case, one of the vectors consists of the most recent  $n$  signal samples in a continuing time sequence of inputs to a signal-processing unit. The other vector is a set of  $n$  weights. The  $n$  signal samples are multiplied by the weights, and the sum of these products constitutes an output signal sample.

Some computer instruction sets combine the operations of the Multiply and Add instructions used in the programs in Figures 2.27 and 2.28 into a single MultiplyAccumulate instruction. This is done in the ARM processor presented in Appendix D.

## 2.12.2 STRING SEARCH PROGRAM

As an example of a non-numerical application, let us consider the problem of string search. Given two strings of ASCII-encoded characters, a long string  $T$  and a short string  $P$ , we want to determine if the pattern  $P$  is contained in the target  $T$ . Since  $P$  may be found in  $T$  in several places, we will simplify our task by being interested only in the first occurrence of  $P$  in  $T$  when  $T$  is searched from left to right. Let  $T$  and  $P$  consist of  $n$  and  $m$  characters, respectively, where  $n > m$ . The characters are stored in memory in consecutive byte locations. Assume that the required data are located as follows:

- $T$  is the address of  $T(0)$ , which is the first character in string  $T$ .
- $N$  is the address of a 32-bit word that contains the value  $n$ .
- $P$  is the address of  $P(0)$ , which is the first character in string  $P$ .
- $M$  is the address of a 32-bit word that contains the value  $m$ .
- $\text{RESULT}$  is the address of a word in which the result of the search is to be stored. If the substring  $P$  is found in  $T$ , then the address of the corresponding location in  $T$  will be stored in  $\text{RESULT}$ ; otherwise, the value  $-1$  will be stored.

String search is an important and well-researched problem. Many algorithms have been developed. Since our main purpose is to illustrate the use of assembly-language instructions, we will use the simplest algorithm which is known as the *brute-force* algorithm. It is given in Figure 2.29.

In a RISC-style computer, the algorithm can be implemented as shown in Figure 2.30. The comments explain the use of various processor registers. Note that in the case of a failed search, the immediate value  $-1$  will cause the contents of R8 to become equal to  $0xFFFFFFFF$ , which represents  $-1$  in 2's complement.

Figure 2.31 shows how the algorithm may be implemented in a CISC-style computer. Observe that the first instruction in LOOP2 loads a character from string  $T$  into register R8, which is followed by an instruction that compares this character with a character in string  $P$ . The reader may wonder why it is not possible to use a single instruction

CompareByte (R6)+, (R7)+

to achieve the same effect. While CISC-style instruction sets allow operations that involve memory operands, they typically require that if one operand is in the memory, the other

---

```

for i ← 0 to n - m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return -1

```

---

**Figure 2.29** A brute-force string search algorithm.

---

	Move	R2, #T	R2 points to string $T$ .
	Move	R3, #P	R3 points to string $P$ .
	Load	R4, N	Get the value $n$ .
	Load	R5, M	Get the value $m$ .
	Subtract	R4, R4, R5	Compute $n - m$ .
	Add	R4, R2, R4	The address of $T(n - m)$ .
	Add	R5, R3, R5	The address of $P(m)$ .
LOOP1:	Move	R6, R2	Use R6 to scan through string $T$ .
	Move	R7, R3	Use R7 to scan through string $P$ .
LOOP2:	LoadByte	R8, (R6)	Compare a pair of characters in strings $T$ and $P$ .
	LoadByte	R9, (R7)	
	Branch_if_[R8]≠[R9]	NOMATCH	
	Add	R6, R6, #1	Point to next character in $T$ .
	Add	R7, R7, #1	Point to next character in $P$ .
	Branch_if_[R5]>[R7]	LOOP2	Loop again if not done.
	Store	R2, RESULT	Store the address of $T(i)$ .
	Branch	DONE	
NOMATCH:	Add	R2, R2, #1	Point to next character in $T$ .
	Branch_if_[R4]≥[R2]	LOOP1	Loop again if not done.
	Move	R8, #-1	Write $-1$ to indicate that no match was found.
	Store	R8, RESULT	
DONE:	next instruction		

---

**Figure 2.30** A RISC-style program for string search.

operand must be in a processor register. A common exception is the Move instruction, which may involve two memory operands. This provides a simple way of moving data between different memory locations.

## 2.13 ENCODING OF MACHINE INSTRUCTIONS

In this chapter, we have introduced a variety of useful instructions and addressing modes. We have used a generic form of assembly language to emphasize basic concepts without relying on processor-specific acronyms or mnemonics. Assembly-language instructions symbolically express the actions that must be performed by the processor circuitry. To be executed in a processor, assembly-language instructions must be converted by the assembler program, as described in Section 2.5, into machine instructions that are encoded in a compact binary pattern.

Let us now examine how machine instructions may be formed. The Add instruction

Add Rd<sub>st</sub>, R<sub>src1</sub>, R<sub>src2</sub>

---

	Move	R2, #T	R2 points to string $T$ .
	Move	R3, #P	R3 points to string $P$ .
	Move	R4, N	Get the value $n$ .
	Move	R5, M	Get the value $m$ .
	Subtract	R4, R5	Compute $n - m$ .
	Add	R4, R2	The address of $T(n - m)$ .
	Add	R5, R3	The address of $P(m)$ .
LOOP1:	Move	R6, R2	Use R6 to scan through string $T$ .
	Move	R7, R3	Use R7 to scan through string $P$ .
LOOP2:	MoveByte	R8, (R6)+	Compare a pair of characters in strings $T$ and $P$ .
	CompareByte	R8, (R7)+	
	Branch $\neq 0$	NOMATCH	
	Compare	R5, R7	Check if at $P(m)$ .
	Branch $> 0$	LOOP2	Loop again if not done.
	Move	RESULT, R2	Store the address of $T(i)$ .
	Branch	DONE	
NOMATCH:	Add	R2, #1	Point to next character in $T$ .
	Compare	R4, R2	Check if at $T(n - m)$ .
	Branch $\geq 0$	LOOP1	Loop again if not done.
	Move	RESULT, #-1	No match was found.
DONE:	next instruction		

---

**Figure 2.31** A CISC-style program for string search.

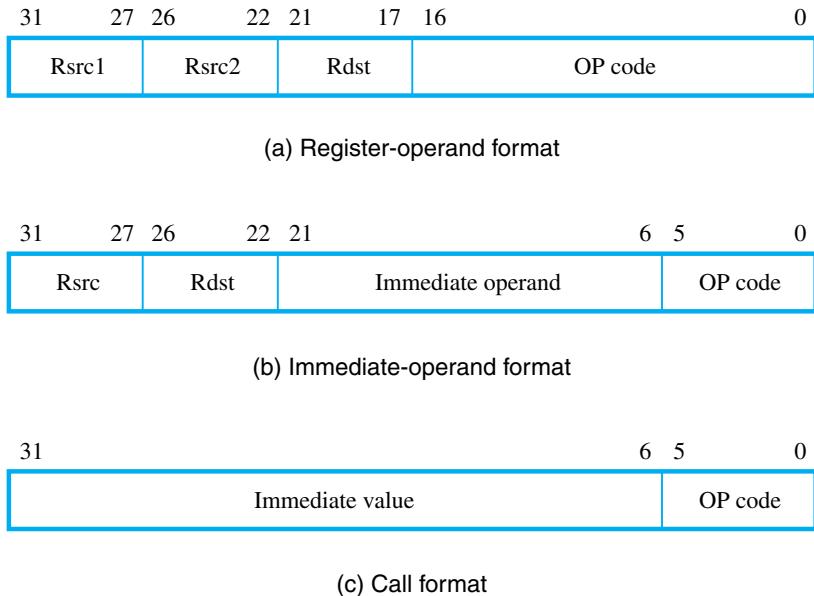
is representative of a class of three-operand instructions that use operands in processor registers. Registers Rdst, Rsrc1, and Rsrc2 hold the destination and two source operands. If a processor has 32 registers, then it is necessary to use five bits to specify each of the three registers in such instructions. If each instruction is implemented in a 32-bit word, the remaining 17 bits can be used to specify the OP code that indicates the operation to be performed. A possible format is shown in Figure 2.32a.

Now consider instructions in which one operand is given using the Immediate addressing mode, such as

Add Rdst, Rsrc, #Value

Of the 32 bits available, ten bits are needed to specify the two registers. The remaining 22 bits must give the OP code and the value of the immediate operand. The most useful sizes of immediate operands are 32, 16, and 8 bits. Since 32 bits are not available, a good choice is to allocate 16 bits for the immediate operand. This leaves six bits for specifying the OP code. A possible format is presented in Figure 2.32b. This format can also be used for Load and Store instructions, where the Index addressing mode uses the 16-bit field to specify the offset that is added to the contents of the index register.

The format in Figure 2.32b can also be used to encode the Branch instructions. Consider the program in Figure 2.12. The Branch-greater-than instruction at memory address 128



**Figure 2.32** Possible instruction formats.

could be written in a specific assembly language as

BGT R2, R0, LOOP

if the contents of register R0 are zero. The registers R2 and R0 can be specified in the two register fields in Figure 2.32b. The six-bit OP code has to identify the BGT operation. The 16-bit immediate field can be used to provide the information needed to determine the branch target address, which is the location of the instruction with the label LOOP. The target address generally comprises 32 bits. Since there is no space for 32 bits, the BGT instruction makes use of the immediate field to give an offset from the location of this instruction in the program to the required branch target. At the time the BGT instruction is being executed, the program counter, PC, has been incremented to point to the next instruction, which is the Store instruction at address 132. Therefore, the branch offset is  $132 - 112 = 20$ . Since the processor computes the target address by adding the current contents of the PC and the branch offset, the required offset in this example is negative, namely  $-20$ .

Finally, we should consider the Call instruction, which is used to call a subroutine. It only needs to specify the OP code and an immediate value that is used to determine the address of the first instruction in the subroutine. If six bits are used for the OP code, then the remaining 26 bits can be used to denote the immediate value. This gives the format shown in Figure 2.32c.

In this section, we introduced the basic concept of encoding the machine instructions. Different commercial processors have instruction sets that vary in the details of implementation. Appendices B to E present the instruction sets of four processors that we have chosen as examples.

## 2.14 CONCLUDING REMARKS

This chapter introduced the representation and execution of instructions and programs at the assembly and machine level as seen by the programmer. The discussion emphasized the basic principles of addressing techniques and instruction sequencing. The programming examples illustrated the basic types of operations implemented by the instruction set of any modern computer. Commonly used addressing modes were introduced. The subroutine concept and the instructions needed to implement it were discussed. In the discussion in this chapter, we provided the contrast between two different approaches to the design of machine instruction sets—the RISC and CISC approaches.

## 2.15 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

**Problem:** Assume that there is a string of ASCII-encoded characters stored in memory starting at address STRING. The string ends with the Carriage Return (CR) character. Write a RISC-style program to determine the length of the string and store it in location LENGTH.

### Example 2.1

**Solution:** Figure 2.33 presents a possible program. The characters in the string are compared to CR (ASCII code 0x0D), and a counter is incremented until the end of the string is reached.

	Move	R2, #STRING	R2 points to the start of the string.
	Clear	R3	R3 is a counter that is cleared to 0.
	Move	R4, #0x0D	ASCII code for Carriage Return.
LOOP:	LoadByte	R5, (R2)	Get the next character.
	Branch_if_[R5]==[R4]	DONE	Finished if character is CR.
	Add	R2, R2, #1	Increment the string pointer.
	Add	R3, R3, #1	Increment the counter.
	Branch	LOOP	Not finished, loop back.
DONE:	Store	R3, LENGTH	Store the count in location LENGTH.

**Figure 2.33** Program for Example 2.1.

---

LIST	EQU	1000	Starting address of the list.
	ORIGIN	400	
	Move	R2, #LIST	R2 points to the start of the list.
	Load	R3, 4(R2)	R3 is a counter, initialize it with $n$ .
	Add	R4, R2, #8	R4 points to the first number.
	Load	R5, (R4)	R5 holds the smallest number found so far.
LOOP:	Subtract	R3, R3, #1	Decrement the counter.
	Branch_if_[R3]=0	DONE	Finished if R3 is equal to 0.
	Add	R4, R4, #4	Increment the list pointer.
	Load	R6, (R4)	Get the next number.
	Branch_if_[R5]≤[R6]	LOOP	Check if smaller number found.
	Move	R5, R6	Update the smallest number found.
	Branch	LOOP	
DONE:	Store	R5, (R2)	Store the smallest number into SMALL.
	ORIGIN	1000	
SMALL:	RESERVE	4	Space for the smallest number found.
N:	DATAWORD	7	Number of entries in the list.
ENTRIES:	DATAWORD	4,5,3,6,1,8,2	Entries in the list.
	END		

---

**Figure 2.34** Program for Example 2.2.

**Example 2.2** **Problem:** We want to find the smallest number in a list of 32-bit positive integers. The word at address 1000 is to hold the value of the smallest number after it has been found. The next word contains the number of entries,  $n$ , in the list. The following  $n$  words contain the numbers in the list. The program is to start at address 400. Write a RISC-style program to find the smallest number and include the assembler directives needed to organize the program and data as specified. While the program has to be able to handle lists of different lengths, include in your code a small list of sample data comprising seven integers.

**Solution:** The program in Figure 2.34 accomplishes the required task. Comments in the program explain how this task is performed.

**Example 2.3** **Problem:** Write a RISC-style program that converts an  $n$ -digit decimal integer into a binary number. The decimal number is given as  $n$  ASCII-encoded characters, as would be the case if the number is entered by typing it on a keyboard. Memory location N contains  $n$ , the ASCII string starts at DECIMAL, and the converted number is stored at BINARY.

**Solution:** Consider a four-digit decimal number,  $D = d_3d_2d_1d_0$ . The value of this number is  $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$ . This representation of the number is the basis for the conversion technique used in the program in Figure 2.35. Note that each ASCII-encoded

---

	Load	R2, N	Initialize counter R2 with $n$ .
	Move	R3, #DECIMAL	R3 points to the ASCII digits.
	Clear	R4	R4 will hold the binary number.
LOOP:	LoadByte	R5, (R3)	Get the next ASCII digit.
	And	R5, R5, #0x0F	Form the BCD digit.
	Add	R4, R4, R5	Add to the intermediate result.
	Add	R3, R3, #1	Increment the digit pointer.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]==0	DONE	
	Multiply	R4, R4, #10	Multiply by 10.
	Branch	LOOP	Loop back if not done.
DONE:	Store	R4, BINARY	Store result in location BINARY.

---

**Figure 2.35** Program for Example 2.3.

character is converted into a Binary Coded Decimal (BCD) digit before it is used in the computation. It is assumed that the converted value can be represented in no more than 32 bits.

**Problem:** Consider an array of numbers  $A(i,j)$ , where  $i = 0$  through  $n - 1$  is the row index, and  $j = 0$  through  $m - 1$  is the column index. The array is stored in the memory of a computer one row after another, with elements of each row occupying  $m$  successive word locations. Assume that the memory is byte-addressable and that the word length is 32 bits. Write a RISC-style subroutine for adding column  $x$  to column  $y$ , element by element, leaving the sum elements in column  $y$ . The indices  $x$  and  $y$  are passed to the subroutine in registers R2 and R3. The parameters  $n$  and  $m$  are passed to the subroutine in registers R4 and R5, and the address of element  $A(0,0)$  is passed in register R6.

#### Example 2.4

**Solution:** A possible program is given in Figure 2.36. We have assumed that the values  $x$ ,  $y$ ,  $n$ , and  $m$  are stored in memory locations X, Y, N, and M. Also, the elements of the array are stored in successive words that begin at location ARRAY, which is the address of the element  $A(0,0)$ . Comments in the program indicate the purpose of individual instructions.

**Problem:** We want to sort a list of characters stored in memory. The list consists of  $n$  bytes, not necessarily distinct, and each byte contains the ASCII code for a character from the set of letters A through Z. In the ASCII code, presented in Chapter 1, the letters A, B, ..., Z, are represented by 7-bit patterns that have increasing values when interpreted as binary numbers. When an ASCII character is stored in a byte location, it is customary to set the most-significant bit position to 0. Using this code, we can sort a list of characters alphabetically by sorting their codes in increasing numerical order, considering them as positive numbers.

#### Example 2.5

	Load	R2, X	Load the value $x$ .
	Load	R3, Y	Load the value $y$ .
	Load	R4, N	Load the value $n$ .
	Load	R5, M	Load the value $m$ .
	Move	R6, #ARRAY	Load the address of A(0,0).
	Call	SUB	
	next instruction		
	:		
SUB:	Subtract	SP, SP, #4	
	Store	R7, (SP)	Save register R7.
	LShiftL	R5, R5, #2	Determine the distance in bytes between successive elements in a column.
	Subtract	R3, R3, R2	Form $y - x$ .
	LShiftL	R3, R3, #2	Form $4(y - x)$ .
	LShiftL	R2, R2, #2	Form $4x$ .
	Add	R6, R6, R2	R6 points to A(0, $x$ ).
	Add	R7, R6, R3	R7 points to A(0, $y$ ).
LOOP:	Load	R2, (R6)	Get the next number in column $x$ .
	Load	R3, (R7)	Get the next number in column $y$ .
	Add	R2, R2, R3	Add the numbers and store the sum.
	Store	R2, (R7)	
	Add	R6, R6, R5	Increment pointer to column $x$ .
	Add	R7, R7, R5	Increment pointer to column $y$ .
	Subtract	R4, R4, #1	Decrement the row counter.
	Branch_if_[R4]>0	LOOP	Loop back if not done.
	Load	R7, (SP)	Restore R7.
	Add	SP, SP, #4	
	Return		Return to the calling program.

---

**Figure 2.36** Program for Example 2.4.

Let the list be stored in memory locations LIST through LIST +  $n - 1$ , and let  $n$  be a 32-bit value stored at address N. The sorting is to be done in place, that is, the sorted list is to occupy the same memory locations as the original list.

We can sort the list using a straight-selection sort algorithm. First, the largest number is found and placed at the end of the list in location LIST +  $n - 1$ . Then the largest number in the remaining sublist of  $n - 1$  numbers is placed at the end of the sublist in location LIST +  $n - 2$ . The procedure is repeated until the list is sorted. A C-language program for this sorting algorithm is shown in Figure 2.37, where the list is treated as a one-dimensional array LIST(0) through LIST( $n - 1$ ). For each sublist LIST( $j$ ) through LIST(0), the number in LIST( $j$ ) is compared with each of the other numbers in the sublist. Whenever a larger number is found in the sublist, it is interchanged with the number in LIST( $j$ ).

---

```

for (j = n-1; j > 0; j = j - 1)
{ for (k = j-1; k >= 0; k = k - 1)
  { if (LIST[k] > LIST[j])
    { TEMP = LIST[k];
      LIST[k] = LIST[j];
      LIST[j] = TEMP;
    }
  }
}

```

---

**Figure 2.37** C-language program for sorting.

---

	Move	R2, #LIST	Load LIST into base register R2.
	Move	R3, N	Initialize outer loop index
	Subtract	R3, #1	register R3 to $j = n - 1$ .
OUTER:	Move	R4, R3	Initialize inner loop index
	Subtract	R4, #1	register R4 to $k = j - 1$ .
	MoveByte	R5, (R2,R3)	Load LIST( $j$ ) into R5, which holds current maximum in sublist.
INNER:	CompareByte	(R2,R4), R5	If $\text{LIST}(k) \leq [\text{R5}]$ ,
	Branch $\leq 0$	NEXT	do not exchange.
	MoveByte	R6, (R2,R4)	Otherwise, exchange LIST( $k$ )
	MoveByte	(R2,R4), R5	with LIST( $j$ ) and load
	MoveByte	(R2,R3), R6	new maximum into R5.
	MoveByte	R5, R6	Register R6 serves as TEMP.
NEXT:	Decrement	R4	Decrement index registers R4 and
	Branch $\geq 0$	INNER	R3, which also serve as
	Decrement	R3	loop counters, and branch
	Branch $> 0$	OUTER	back if loops not finished.

---

**Figure 2.38** A byte-sorting program.

Note that the C-language program traverses the list backwards. This order of traversal simplifies loop termination when a machine language program is written, because the loop is exited when an index is decremented to 0.

Write a CISC-style program that implements this sorting task.

**Solution:** A possible program is given in Figure 2.38.

## PROBLEMS

- 2.1** [E] Given a binary pattern in some memory location, is it possible to tell whether this pattern represents a machine instruction or a number?
- 2.2** [E] Consider a computer that has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word “Computer” has been entered.
- 2.3** [E] Repeat Problem 2.2 for the little-endian scheme.
- 2.4** [E] Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case?
- (a)  $12(R4)$
  - (b)  $(R4,R5)$
  - (c)  $28(R4,R5)$
  - (d)  $(R4)+$
  - (e)  $-(R4)$
- 2.5** [E] Write a RISC-style program that computes the expression  $SUM = 580 + 68400 + 80000$ .
- 2.6** [E] Write a CISC-style program for the task in Problem 2.5.
- 2.7** [E] Write a RISC-style program that computes the expression  $ANSWER = A \times B + C \times D$ .
- 2.8** [E] Write a CISC-style program for the task in Problem 2.7.
- 2.9** [M] Rewrite the addition loop in Figure 2.8 so that the numbers in the list are accessed in the reverse order; that is, the first number accessed is the last one in the list, and the last number accessed is at memory location NUM1. Try to achieve the most efficient way to determine loop termination. Would your loop execute faster than the loop in Figure 2.8?
- 2.10** [M] The list of student marks shown in Figure 2.10 is changed to contain  $j$  test scores for each student. Assume that there are  $n$  students. Write a RISC-style program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses  $SUM$ ,  $SUM + 4$ ,  $SUM + 8$ ,  $\dots$ . The number of tests,  $j$ , is larger than the number of registers in the processor, so the type of program shown in Figure 2.11 for the 3-test case cannot be used. Use two nested loops. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests,  $j$ . Assume that the memory area used to store the sums has been cleared to zero initially.
- 2.11** [M] Write a RISC-style program that finds the number of negative integers in a list of  $n$  32-bit integers and stores the count in location  $NEGNUM$ . The value  $n$  is stored in memory location  $N$ , and the first integer in the list is stored in location  $NUMBERS$ . Include the necessary assembler directives and a sample list that contains six numbers, some of which are negative.

- 2.12** [E] Both of the following statement segments cause the value 300 to be stored in location 1000, but at different times.

ORIGIN	1000
DATAWORD	300

and

Move	R2, #1000
Move	R3, #300
Store	R3, (R2)

Explain the difference.

- 2.13** [E] Write an assembly-language program in the style of Figure 2.13 for the program in Figure 2.11. Assume the data layout of Figure 2.10.
- 2.14** [E] Write a CISC-style program for the task in Example 2.1. At most one operand of an instruction can be in the memory.
- 2.15** [E] Write a CISC-style program for the task in Example 2.2. At most one operand of an instruction can be in the memory.
- 2.16** [M] Write a CISC-style program for the task in Example 2.3. At most one operand of an instruction can be in the memory.
- 2.17** [M] Write a CISC-style program for the task in Example 2.4. At most one operand of an instruction can be in the memory.
- 2.18** [M] Write a RISC-style program for the task in Example 2.5.
- 2.19** [E] Register R5 is used in a program to point to the top of a stack containing 32-bit numbers. Write a sequence of instructions using the Index, Autoincrement, and Autodecrement addressing modes to perform each of the following tasks:
- (a) Pop the top two items off the stack, add them, then push the result onto the stack.
  - (b) Copy the fifth item from the top into register R3.
  - (c) Remove the top ten items from the stack.
- For each case, assume that the stack contains ten or more elements.
- 2.20** [M] Show the processor stack contents and the contents of the stack pointer, SP, immediately after each of the following instructions in the program in Figure 2.18 is executed. Assume that  $[SP] = 1000$  at Level 1, before execution of the calling program begins.
- (a) The second Store instruction in the subroutine
  - (b) The last Load instruction in the subroutine
  - (c) The last Store instruction in the calling program
- 2.21** [M] Consider the following possibilities for saving the return address of a subroutine:
- (a) In a processor register
  - (b) In a memory location associated with the call, so that a different location is used when the subroutine is called from different places
  - (c) On a stack

Which of these possibilities supports subroutine nesting and which supports subroutine recursion (that is, a subroutine that calls itself)?

**2.22** [M] In addition to the processor stack, it may be convenient to use another stack in some programs. The second stack is usually allocated a fixed amount of space in the memory. In this case, it is important to avoid pushing an item onto the stack when the stack has reached its maximum size. Also, it is important to avoid attempting to pop an item off an empty stack, which could result from a programming error. Write two short RISC-style routines, called SAFEPUSH and SAFEPOP, for pushing onto and popping off this stack structure, while guarding against these two possible errors. Assume that the element to be pushed/popped is located in register R2, and that register R5 serves as the stack pointer for this user stack. The stack is full if its topmost element is stored in location TOP, and it is empty if the last element popped was stored in location BOTTOM. The routines should branch to FULLERROR and EMPTYERROR, respectively, if errors occur. All elements are of word size, and the stack grows toward lower-numbered address locations.

**2.23** [M] Repeat Problem 2.22 for CISC-style routines that can use Autoincrement and Autodecrement addressing modes.

**2.24** [D] Another useful data structure that is similar to the stack is called a *queue*. Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

A FIFO queue of bytes is to be implemented in the memory, occupying a fixed region of  $k$  bytes. The necessary pointers are an IN pointer and an OUT pointer. The IN pointer keeps track of the location where the next byte is to be appended to the back of the queue, and the OUT pointer keeps track of the location containing the next byte to be removed from the front of the queue.

(a) As data items are added to the queue, they are added at successively higher addresses until the end of the memory region is reached. What happens next, when a new item is to be added to the queue?

(b) Choose a suitable definition for the IN and OUT pointers, indicating what they point to in the data structure. Use a simple diagram to illustrate your answer.

(c) Show that if the state of the queue is described only by the two pointers, the situations when the queue is completely full and completely empty are indistinguishable.

(d) What condition would you add to solve the problem in part (c)?

(e) Propose a procedure for manipulating the two pointers IN and OUT to append and remove items from the queue.

- 2.25** [M] Consider the queue structure described in Problem 2.24. Write APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.
- 2.26** [M] The dot-product computation is discussed in Section 2.12.1. This type of computation can be used in the following signal-processing task. An input signal time sequence  $\text{IN}(0), \text{IN}(1), \text{IN}(2), \text{IN}(3), \dots$ , is processed by a 3-element weight vector  $(\text{WT}(0), \text{WT}(1), \text{WT}(2)) = (1/8, 1/4, 1/2)$  to produce an output signal time sequence  $\text{OUT}(0), \text{OUT}(1), \text{OUT}(2), \text{OUT}(3), \dots$ , as follows:

$$\begin{aligned}\text{OUT}(0) &= \text{WT}(0) \times \text{IN}(0) + \text{WT}(1) \times \text{IN}(1) + \text{WT}(2) \times \text{IN}(2) \\ \text{OUT}(1) &= \text{WT}(0) \times \text{IN}(1) + \text{WT}(1) \times \text{IN}(2) + \text{WT}(2) \times \text{IN}(3) \\ \text{OUT}(2) &= \text{WT}(0) \times \text{IN}(2) + \text{WT}(1) \times \text{IN}(3) + \text{WT}(2) \times \text{IN}(4) \\ \text{OUT}(3) &= \text{WT}(0) \times \text{IN}(3) + \text{WT}(1) \times \text{IN}(4) + \text{WT}(2) \times \text{IN}(5) \\ &\vdots\end{aligned}$$

All signal and weight values are 32-bit signed numbers. The weights, inputs, and outputs, are stored in the memory starting at locations WT, IN, and OUT, respectively. Write a RISC-style program to calculate and store the output values for the first  $n$  outputs, where  $n$  is stored at location N.

Hint: Arithmetic right shifts can be used to do the multiplications.

- 2.27** [M] Write a subroutine MEMCPY for copying a sequence of bytes from one area in the main memory to another area. The subroutine should accept three input parameters in registers representing the *from* address, the *to* address, and the *length* of the sequence to be copied. The two areas may overlap. In all but one case, the subroutine should copy the bytes in the order of increasing addresses. However, in the case where the *to* address falls within the sequence of bytes to be copied, i.e., when the *to* address is between *from* and *from+length*–1, the subroutine must copy the bytes in the order of decreasing addresses by starting at the end of the sequence of bytes to be copied in order to avoid overwriting bytes that have not yet been copied.
- 2.28** [M] Write a subroutine MEMCMP for performing a byte-by-byte comparison of two sequences of bytes in the main memory. The subroutine should accept three input parameters in registers representing the *first* address, the *second* address, and the *length* of the sequences to be compared. It should use a register to return the count of the number of comparisons that do not match.
- 2.29** [M] Write a subroutine called EXCLAIM that accepts a single parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes representing an arbitrary collection of sentences, with the NUL control character (value 0) at the end of the string. The subroutine should scan the string beginning at address STRNG and replace every occurrence of a period ('.') with an exclamation mark ('!').

- 2.30** [M] Write a subroutine called ALLCAPS that accepts a parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes, with the NUL control character (value 0) at the end of the string. The subroutine should scan the string beginning at address STRNG and replace every occurrence of a lower-case letter ('a'–'z') with the corresponding upper-case letter ('A'–'Z').
- 2.31** [M] Write a subroutine called WORDS that accepts a parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes, with the NUL control character (value 0) at the end of the string. The string represents English text with the space character between words. The subroutine has to determine the number of words in the string (excluding the punctuation characters). It must return the result to the calling program in a register.
- 2.32** [D] Write a subroutine called INSERT that places a number in the correct ordered position within a list of positive numbers that are stored in increasing order of value. Three input parameters should be passed to the subroutine in processor registers, representing the starting address of the ordered list of numbers, the length of the list, and the new value to be inserted into the list. The subroutine should locate the appropriate position for the new value in the list, then shift all of the larger numbers up by one position to create space for storing the new value in the list.
- 2.33** [D] Write a subroutine called INSERTSORT that repeatedly uses the INSERT subroutine in Problem 2.32 to take an unordered list of numbers and create a new list with the same numbers in increasing order. The subroutine should accept three input parameters in registers representing the starting address OLDLIST for the unordered sequence of numbers, the length of the list, and the starting address NEWLIST for the ordered sequence of numbers.



# CHAPTER 14

## PROCESSOR STRUCTURE AND FUNCTION

### **14.1 Processor Organization**

### **14.2 Register Organization**

- User-Visible Registers
- Control and Status Registers
- Example Microprocessor Register Organizations

### **14.3 Instruction Cycle**

- The Indirect Cycle
- Data Flow

### **14.4 Instruction Pipelining**

- Pipelining Strategy
- Pipeline Performance
- Pipeline Hazards
- Dealing with Branches
- Intel 80486 Pipelining

### **14.5 The x86 Processor Family**

- Register Organization
- Interrupt Processing

### **14.6 The Arm Processor**

- Processor Organization
- Processor Modes
- Register Organization
- Interrupt Processing

### **14.7 Key Terms, Review Questions, and Problems**

### LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Distinguish between **user-visible** and **control/status registers**, and discuss the purposes of registers in each category.
- ◆ Summarize the **instruction cycle**.
- ◆ Discuss the principle behind **instruction pipelining** and how it works in practice.
- ◆ Compare and contrast the various forms of pipeline hazards.
- ◆ Present an overview of the x86 processor structure.
- ◆ Present an overview of the ARM processor structure.

This chapter discusses aspects of the processor not yet covered in Part Three and sets the stage for the discussion of RISC and superscalar architecture in Chapters 15 and 16.

We begin with a summary of processor organization. Registers, which form the internal memory of the processor, are then analyzed. We are then in a position to return to the discussion (begun in Section 3.2) of the instruction cycle. A description of the instruction cycle and a common technique known as instruction pipelining complete our description. The chapter concludes with an examination of some aspects of the x86 and ARM organizations.

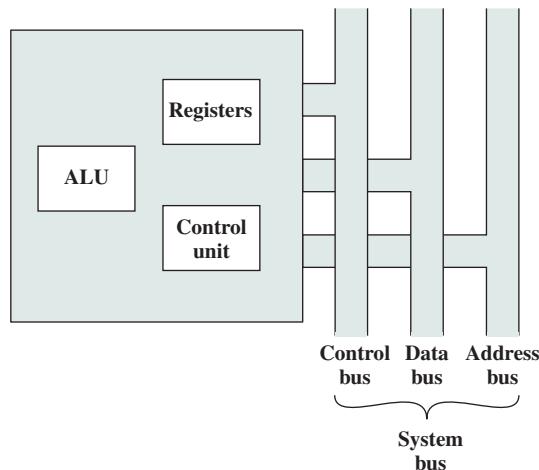
## 14.1 PROCESSOR ORGANIZATION

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
- **Write data:** The results of an execution may require writing data to memory or an I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs a small internal memory.

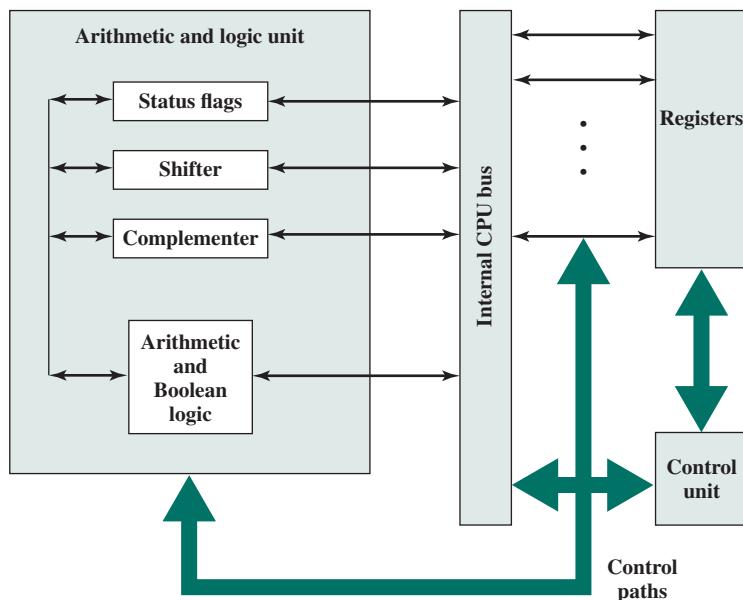
Figure 14.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. A similar interface would be needed for any



**Figure 14.1** The CPU with the System Bus

of the interconnection structures described in Chapter 3. The reader will recall that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers*.

Figure 14.2 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled *internal*



**Figure 14.2** Internal Structure of the CPU

*processor bus.* This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

## 14.2 REGISTER ORGANIZATION

As we discussed in Chapter 4, a computer system employs a memory hierarchy. At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor perform two roles:

- **User-visible registers:** Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers.
- **Control and status registers:** Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., x86), but on many it is not. For purposes of the following discussion, however, we will use these categories.

### User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

**General-purpose registers** can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers. **Data registers** may be used only to hold data and cannot be employed in the calculation of an operand address.

**Address registers** may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing (see Section 8.3), a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers:** These are used for indexed addressing and may be autoindexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

There are several design issues to be addressed here. An important issue is whether to use completely general-purpose registers or to specialize their use. We have already touched on this issue in the preceding chapter because it affects instruction set design. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The operand specifier must only identify one of a set of specialized registers rather than one out of all the registers, thus saving bits. On the other hand, this specialization limits the programmer's flexibility.

Another design issue is the number of registers, either general purpose or data plus address, to be provided. Again, this affects instruction set design because more registers require more operand specifier bits. As we previously discussed, somewhere between 8 and 32 registers appears optimum [LUND77]. Fewer registers result in more memory references; more registers do not noticeably reduce memory references (e.g., see [WILL90]). However, a new approach, which finds advantage in the use of hundreds of registers, is exhibited in some RISC systems and is discussed in Chapter 15.

Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds **condition codes** (also referred to as *flags*). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

Many processors, including those based on the IA-64 architecture and the MIPS processors, do not use condition codes at all. Rather, conditional branch instructions specify a comparison to be made and act on the result of the comparison, without storing a condition code. Table 14.1, based on [DERO87], lists key advantages and disadvantages of condition codes.

**Table 14.1** Condition Codes

Advantages	Disadvantages
<ol style="list-style-type: none"> <li>1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed.</li> <li>2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH.</li> <li>3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.</li> <li>4. Condition codes can be saved on the stack during subroutine calls along with other register information.</li> </ol>	<ol style="list-style-type: none"> <li>1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer.</li> <li>2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections.</li> <li>3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations.</li> <li>4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.</li> </ol>

In some machines, a subroutine call will result in the automatic saving of all user-visible registers, to be restored on return. The processor performs the saving and restoring as part of the execution of call and return instructions. This allows each subroutine to use the user-visible registers independently. On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call, by including instructions for this purpose in the program.

### Control and Status Registers

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

Four registers are essential to instruction execution:

- **Program counter (PC):** Contains the address of an instruction to be fetched.
- **Instruction register (IR):** Contains the instruction most recently fetched.
- **Memory address register (MAR):** Contains the address of a location in memory.
- **Memory buffer register (MBR):** Contains a word of data to be written to memory or the word most recently read.

Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred

to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the processor and memory. Within the processor, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

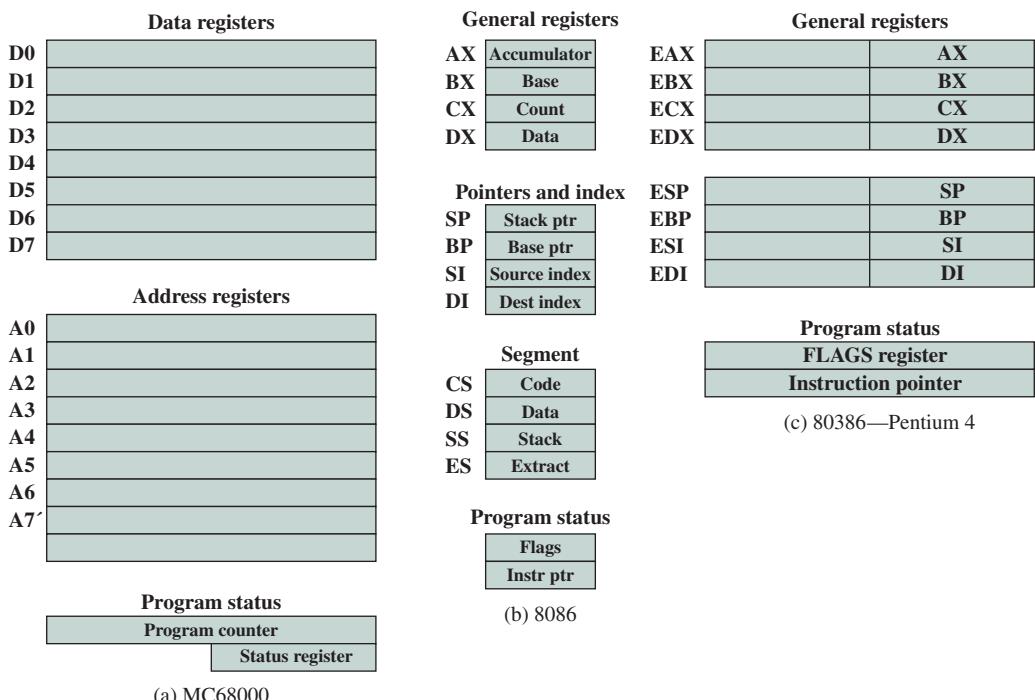
Many processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

A number of other registers related to status and control might be found in a particular processor design. There may be a pointer to a block of memory containing additional status information (e.g., process control blocks). In machines using vectored interrupts, an interrupt vector register may be provided. If a stack is used to implement certain functions (e.g., subroutine call), then a system stack pointer is needed. A page table pointer is used with a virtual memory system. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then the register organization can to some extent be tailored to the operating system.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or



**Figure 14.3** Example Microprocessor Register Organizations

thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost versus speed arises.

### Example Microprocessor Register Organizations

It is instructive to examine and compare the register organization of comparable systems. In this section, we look at two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 [STR79] and the Intel 8086 [MORS78]. Figures 14.3a and b depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations, determined by opcode. The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode. Both registers are numbered 7, because only one can be used at a time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

The Motorola team wanted a very regular instruction set, with no special-purpose registers. A concern for code efficiency led them to divide the registers into

two functional components, saving one bit on each register specifier. This seems a reasonable compromise between complete generality and code compactness.

The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. In others, the registers are used implicitly. For example, a multiply instruction always uses the accumulator. The four pointer registers are also used implicitly in a number of operations; each contains a segment offset. There are also four 16-bit segment registers. Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction (useful for branch instructions), a segment containing data, and a segment containing a stack, respectively. These dedicated and implicit uses provide for compact encoding at the cost of reduced flexibility. The 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

The point of this comparison should be clear. There is no universally accepted philosophy concerning the best way to organize processor registers [TOON81]. As with overall instruction set design and so many other processor design issues, it is still a matter of judgment and taste.

A second instructive point concerning register organization design is illustrated in Figure 14.3c. This figure shows the user-visible register organization for the Intel 80386 [ELAY85], which is a 32-bit microprocessor designed as an extension of the 8086.<sup>1</sup> The 80386 uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, the architects of the 32-bit processors had limited flexibility in designing the register organization.

## 14.3 INSTRUCTION CYCLE

In Section 3.2, we described the processor's instruction cycle (Figure 3.9). To recall, an instruction cycle includes the following stages:

- **Fetch:** Read the next instruction from memory into the processor.
- **Execute:** Interpret the opcode and perform the indicated operation.
- **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

We are now in a position to elaborate somewhat on the instruction cycle. First, we must introduce one additional stage, known as the indirect cycle.

---

<sup>1</sup>Because the MC68000 already uses 32-bit registers, the MC68020 [MACD84], which is a full 32-bit architecture, uses the same register organization.

## The Indirect Cycle

We have seen, in Chapter 13, that the execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required.

We can think of the fetching of indirect addresses as one more instruction stages. The result is shown in Figure 14.4. The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing. Following execution, an interrupt may be processed before the next instruction fetch.

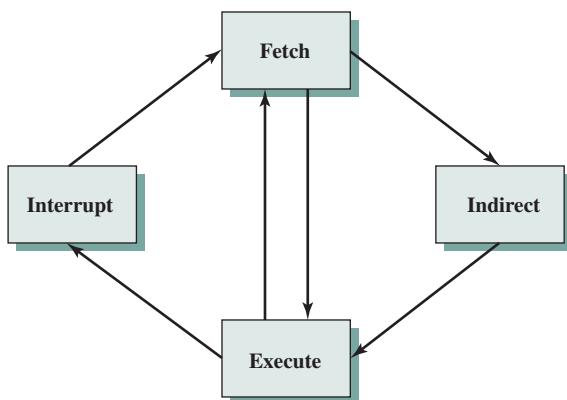
Another way to view this process is shown in Figure 14.5, which is a revised version of Figure 3.12. This illustrates more correctly the nature of the instruction cycle. Once an instruction is fetched, its operand specifiers must be identified. Each input operand in memory is then fetched, and this process may require indirect addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

## Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor. We can, however, indicate in general terms what must happen. Let us assume that a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the *fetch cycle*, an instruction is read from memory. Figure 14.6 shows the flow of data during this cycle. The PC contains the address of the next instruction to be fetched. This address is moved to the MAR and placed on the address bus. The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR. Meanwhile, the PC is incremented by 1, preparatory for the next fetch.

Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing. If so, an



**Figure 14.4** The Instruction Cycle

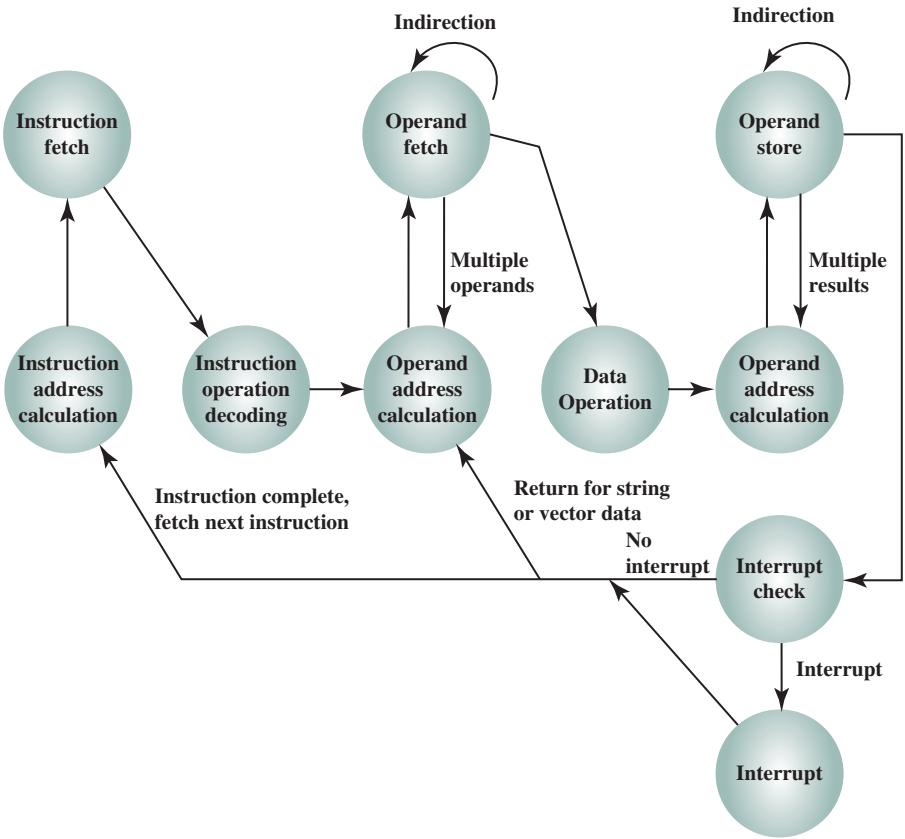
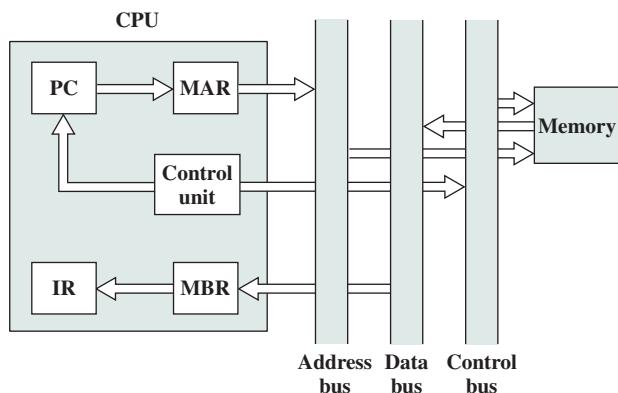


Figure 14.5 Instruction Cycle State Diagram



MBR = Memory buffer register

MAR = Memory address register

IR = Instruction register

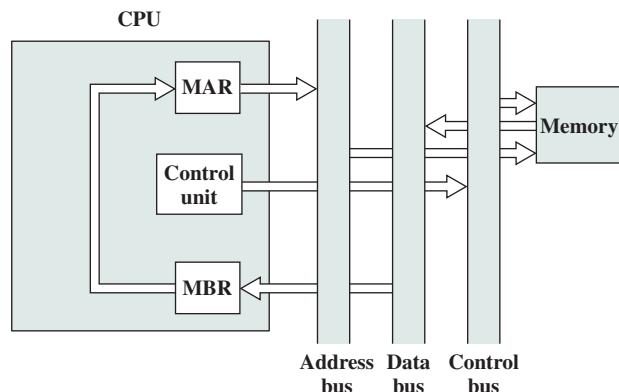
PC = Program counter

Figure 14.6 Data Flow, Fetch Cycle

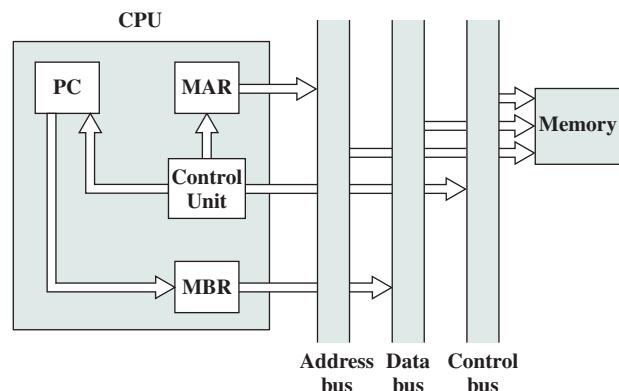
*indirect cycle* is performed. As shown in Figure 14.7, this is a simple cycle. The right-most  $N$  bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.

Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 14.8). The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine. As a result, the next instruction cycle will begin by fetching the appropriate instruction.



**Figure 14.7** Data Flow, Indirect Cycle



**Figure 14.8** Data Flow, Interrupt Cycle

## 14.4 INSTRUCTION PIPELINING

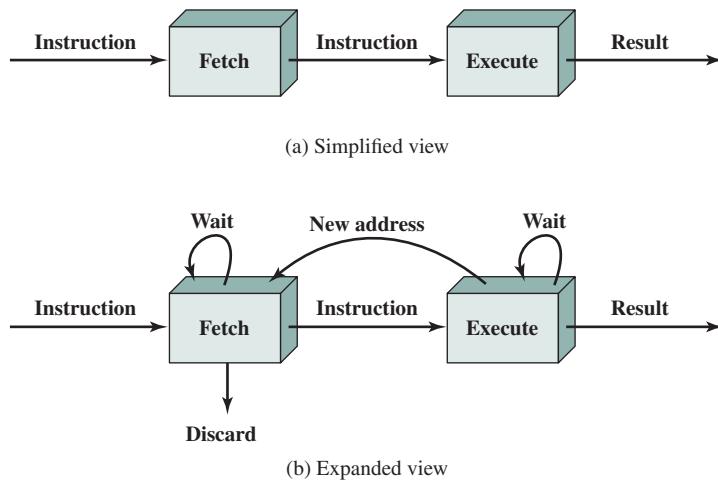
As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance. We have already seen some examples of this, such as the use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

### Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Figures 14.5, for example, breaks the instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Figure 14.9a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch



**Figure 14.9** Two-Stage Instruction Pipeline

and buffer the next instruction. This is called instruction prefetch or *fetch overlap*. Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 14.9b), we will see that this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. For the sake of illustration, let us assume equal duration. Using this assumption, Figure 14.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that.

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

**Figure 14.10** Timing Diagram for Instruction Pipeline Operation

However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 14.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 14.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 14.10 and 14.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 14.13 shows same sequence of events, with time progressing vertically down

	Time →														Branch penalty ←	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Instruction 1	FI	DI	CO	FO	EI	WO										
Instruction 2		FI	DI	CO	FO	EI	WO									
Instruction 3			FI	DI	CO	FO	EI	WO								
Instruction 4				FI	DI	CO	FO									
Instruction 5					FI	DI	CO									
Instruction 6						FI	DI									
Instruction 7							FI									
Instruction 15								FI	DI	CO	FO	EI	WO			
Instruction 16									FI	DI	CO	FO	EI	WO		

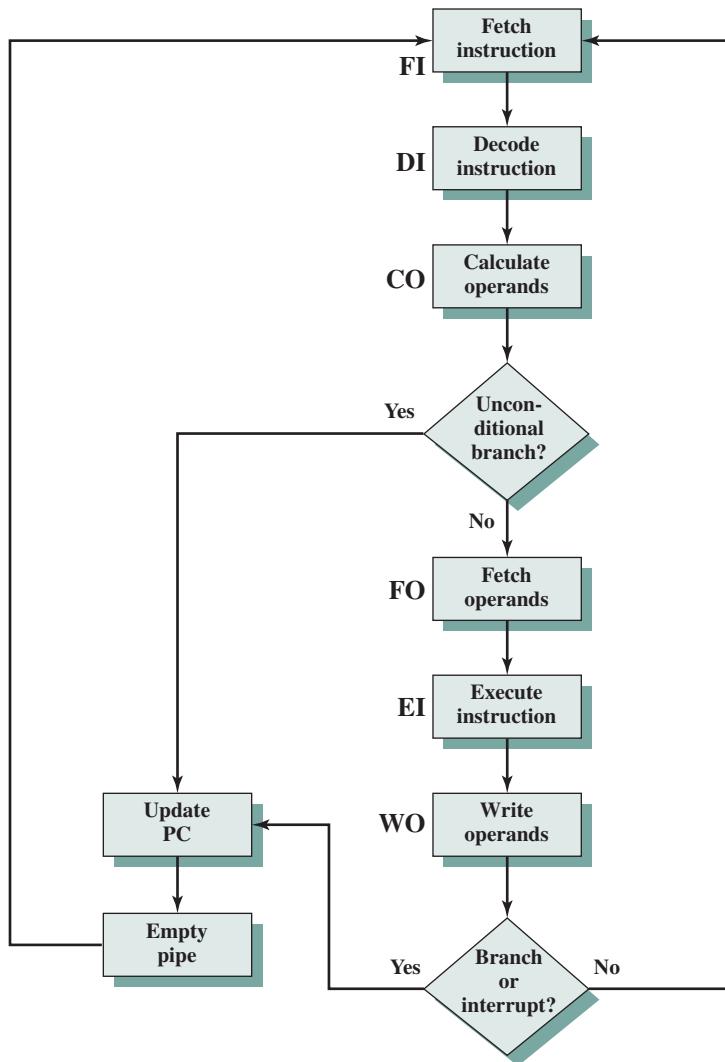
**Figure 14.11** The Effect of a Conditional Branch on Instruction Pipeline Operation

the figure, and each row showing the state of the pipeline at a given point in time. In Figure 14.13a (which corresponds to Figure 14.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. Some of the IBM S/360 designers pointed out two factors that frustrate this seemingly simple pattern for high-performance design [ANDE67a], and they remain elements that designer must still consider:

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction. This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies.
2. The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

Another consideration is latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.



**Figure 14.12** Six-Stage CPU Instruction Pipeline

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

### Pipeline Performance

In this subsection, we develop some simple measures of pipeline performance and relative speedup (based on a discussion in [HWAN93]). The cycle time  $\tau$  of an **instruction pipeline** is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 14.10 and 14.11 represents one cycle time. The cycle time can be determined as

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

(b) With conditional branch

**Figure 14.13** An Alternative Pipeline Depiction

where

$\tau_i$  = time delay of the circuitry in the  $i$ th stage of the pipeline

$\tau_m$  = maximum stage delay (delay through stage which experiences the largest delay)

$k$  = number of stages in the instruction pipeline

$d$  = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay  $d$  is equivalent to a clock pulse and  $\tau_m \gg d$ . Now suppose that  $n$  instructions are processed, with no branches. Let  $T_{k,n}$  be the total time required for a pipeline with  $k$  stages to execute  $n$  instructions. Then

$$T_{k,n} = [k + (n - 1)]\tau \quad (14.1)$$

A total of  $k$  cycles are required to complete the execution of the first instruction, and the remaining  $n - 1$  instructions require  $n - 1$  cycles.<sup>2</sup> This equation is easily verified from Figure 14.10. The ninth instruction completes at time cycle 14:

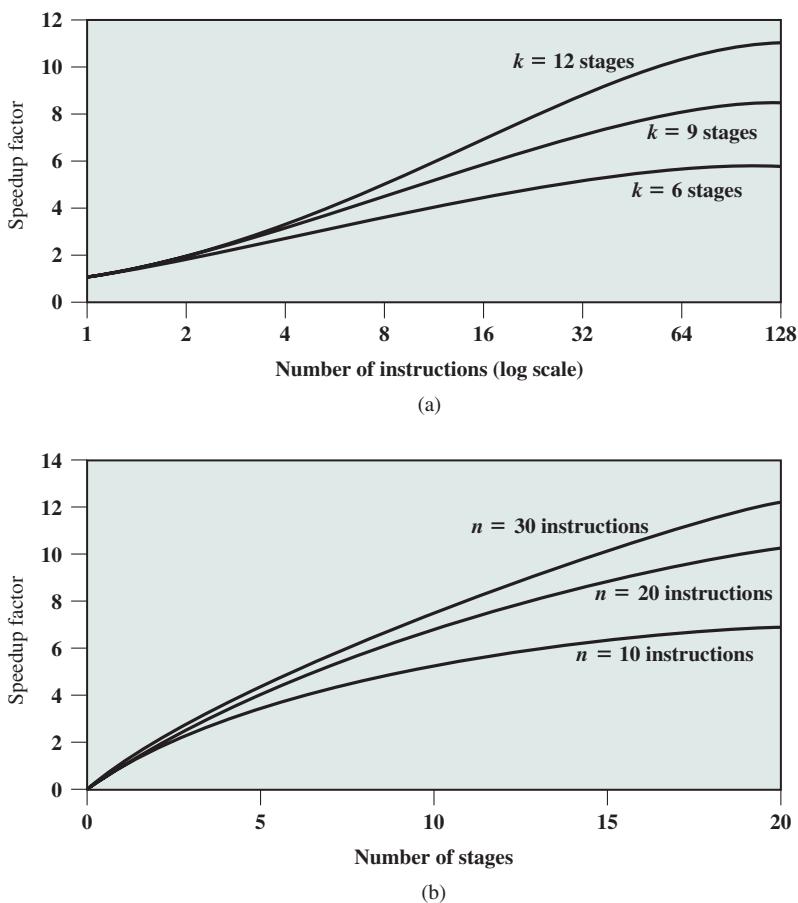
$$14 = [6 + (9 - 1)]$$

<sup>2</sup>We are being a bit sloppy here. The cycle time will only equal the maximum value of  $\tau$  when all the stages are full. At the beginning, the cycle time may be less for the first one or few cycles.

Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is  $k\tau$ . The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \quad (14.2)$$

Figure 14.14a plots the speedup factor as a function of the number of instructions that are executed without a branch. As might be expected, at the limit ( $n \rightarrow \infty$ ), we have a  $k$ -fold speedup. Figure 14.14b shows the speedup factor as a function of the number of stages in the instruction pipeline.<sup>3</sup> In this case, the speedup factor approaches the number of instructions that can be fed into the pipeline without branches. Thus, the larger the number of pipeline stages, the greater the potential for speedup. However, as a practical matter, the potential gains of additional



**Figure 14.14** Speedup Factors with Instruction Pipelining

<sup>3</sup>Note that the  $x$ -axis is logarithmic in Figure 14.14a and linear in Figure 14.14b.

pipeline stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

## Pipeline Hazards

In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in a more systematic way. Chapter 16 revisits this issue, in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

**RESOURCE HAZARDS** A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometimes referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Figure 14.15a shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel

	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			FI	DI	FO	EI	WO	
	I4				FI	DI	FO	EI	WO

(a) Five-stage pipeline, ideal case

	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			Idle	FI	DI	FO	EI	WO
	I4				FI	DI	FO	EI	WO

(b) I1 source operand in memory

**Figure 14.15** Example of Resource Hazard

with an instruction fetch. This is illustrated in Figure 14.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.



### Reservation Table Analyzer

One approach to analyzing resource conflicts and aiding in the design of pipelines is the reservation table. We examine reservation tables in Appendix N.

**DATA HAZARDS** A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

```
ADD EAX, EBX /* EAX = EAX + EBX
```

```
SUB ECX, EAX /* ECX = ECX - EAX
```

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. Figure 14.16 shows the pipeline behavior.

	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure 14.16 Example of Data Hazard

The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clock cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.

There are three types of data hazards:

- **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

The example of Figure 14.16 is a RAW hazard. The other two hazards are best discussed in the context of superscalar organization, discussed in Chapter 16.

**CONTROL HAZARDS** A control hazard, also known as a *branch hazard*, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. We discuss approaches to dealing with control hazards next.

### Dealing with Branches

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

**MULTIPLE STREAMS** A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.

- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

**PREFETCH BRANCH TARGET** When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

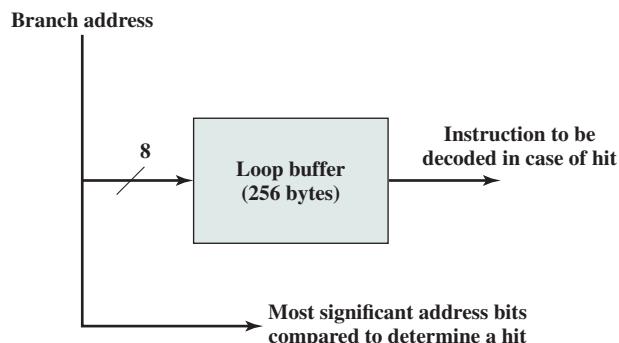
The IBM 360/91 uses this approach.

**LOOP BUFFER** A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the  $n$  most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. The loop buffer has three benefits:

- With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
- If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of IF-THEN and IF-THEN-ELSE sequences.
- This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

Figure 14.17 gives an example of a loop buffer. If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the



**Figure 14.17** Loop Buffer

buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

Among the machines using a loop buffer are some of the CDC machines (Star-100, 6600, 7600) and the CRAY-1. A specialized form of loop buffer is available on the Motorola 68010, for executing a three-instruction loop involving the DBcc (decrement and branch on condition) instruction (see Problem 14.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.



### Branch Prediction Simulator Branch Target Buffer

**BRANCH PREDICTION** Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods.

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path. However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence, and so this performance penalty should be taken into account. An avoidance mechanism may be employed to reduce this penalty.

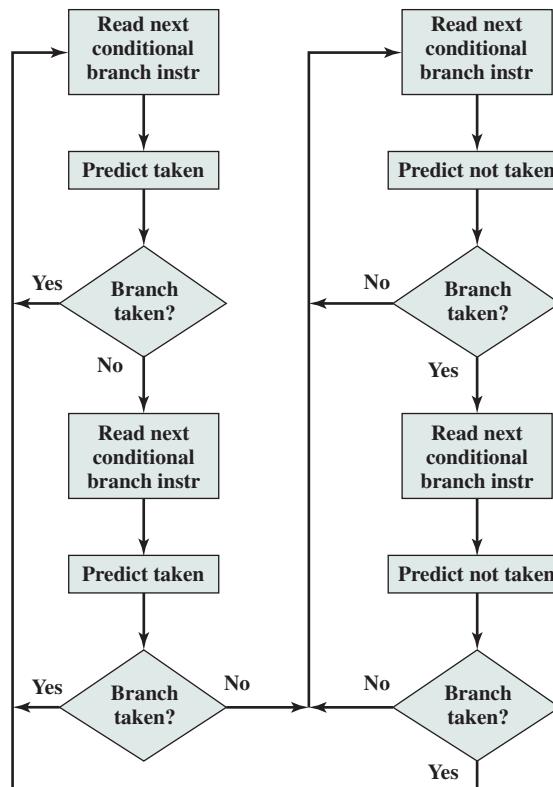
The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. For example, one or more bits can be associated with each conditional branch instruction that

reflect the recent history of the instruction. These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered. Typically, these history bits are not associated with the instruction in main memory. Rather, they are kept in temporary high-speed storage. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction's address.

With a single bit, all that can be recorded is whether the last execution of this instruction resulted in a branch or not. A shortcoming of using a single bit appears in the case of a conditional branch instruction that is almost always taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once on exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. Figure 14.18 shows a typical approach (see Problem 14.13 for other



**Figure 14.18** Branch Prediction Flowchart

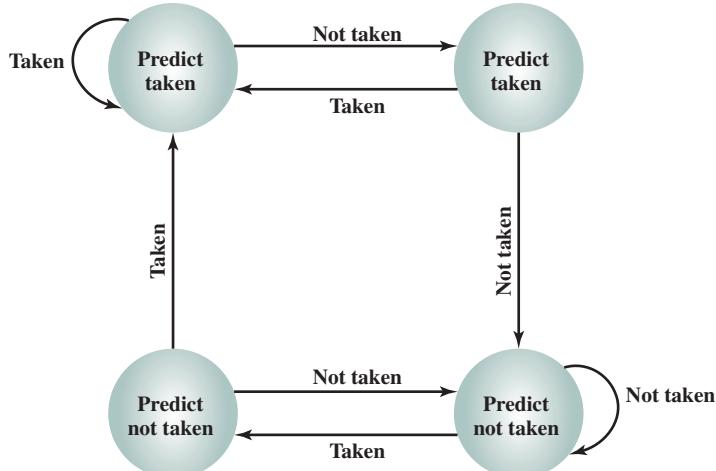
possibilities). Assume that the algorithm starts at the upper-left-hand corner of the flowchart. As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken. If a single prediction is wrong, the algorithm continues to predict that the next branch is taken. Only if two successive branches are not taken does the algorithm shift to the right-hand side of the flowchart. Subsequently, the algorithm will predict that branches are not taken until two branches in a row are taken. Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

The decision process can be represented more compactly by a finite-state machine, shown in Figure 14.19. The finite-state machine representation is commonly used in the literature.

The use of history bits, as just described, has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded. Greater efficiency could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, more information must be saved, in what is known as a branch target buffer, or a branch history table.

The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements: the address of a branch instruction, some number of history bits that record the state of use of that instruction, and information about the target instruction. In most proposals and implementations, this third field contains the address of the target instruction. Another possibility is for the third field to actually contain the target instruction. The trade-off is clear: Storing the target address yields a smaller table but a greater instruction fetch time compared with storing the target instruction [RECH98].

Figure 14.20 contrasts this scheme with a predict-never-taken strategy. With the former strategy, the instruction fetch stage always fetches the next sequential



**Figure 14.19** Branch Prediction State Diagram

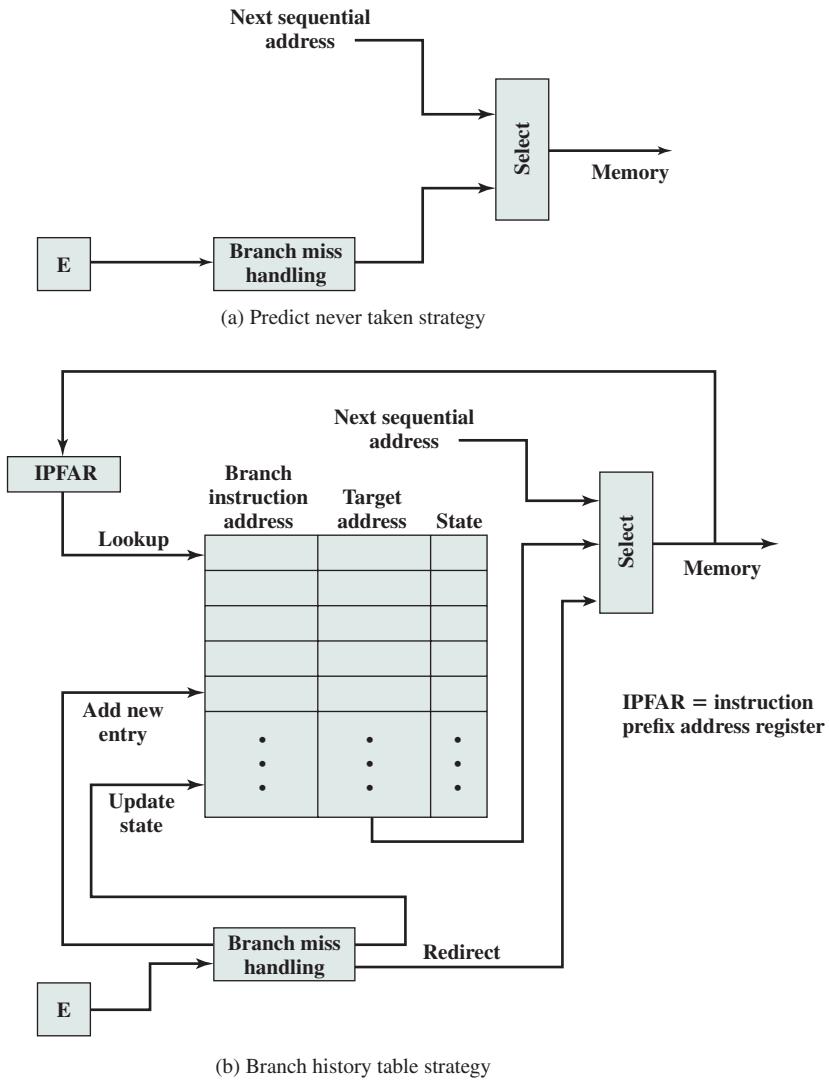


Figure 14.20 Dealing with Branches

address. If a branch is taken, some logic in the processor detects this and instructs that the next instruction be fetched from the target address (in addition to flushing the pipeline). The branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table. If no match is found, the next sequential address is used for the fetch. If a match is found, a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

When the branch instruction is executed, the execute stage signals the branch history table logic with the result. The state of the instruction is updated to reflect a correct or incorrect prediction. If the prediction is incorrect, the select logic is

redirected to the correct address for the next fetch. When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed in Chapter 4.

A refinement of the branch history approach is referred to as two-level or correlation-based branch history [YEH91]. This approach is based on the assumption that whereas in loop-closing branches, the past history of a particular branch instruction is a good predictor of future behavior, with more complex control-flow structures, the direction of a branch is frequently correlated with the direction of related branches. An example is an if-then-else or case structure. There are a number of strategies possible. Typically, recent global branch history (i.e., the history of the most recent branches not just of this branch instruction) is used in addition to the history of the current branch instruction. The general structure is defined as an  $(m, n)$  correlator, which uses the behavior of the last  $m$  branches to choose from  $2^m$   $n$ -bit branch predictors for the current branch instruction. In other words, an  $n$ -bit history is kept for a given branch for each possible combination of branches taken by the most recent  $m$  branches.

**DELAYED BRANCH** It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired. This intriguing approach is examined in Chapter 15.

## Intel 80486 Pipelining

An instructive example of an instruction pipeline is that of the Intel 80486. The 80486 implements a five-stage pipeline:

- **Fetch:** Instructions are fetched from the cache or from external memory and placed into one of the two 16-byte prefetch buffers. The objective of the fetch stage is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder. Because instructions are of variable length (from 1 to 11 bytes not counting prefixes), the status of the prefetcher relative to the other pipeline stages varies from instruction to instruction. On average, about five instructions are fetched with each 16-byte load [CRAW90]. The fetch stage operates independently of the other stages to keep the prefetch buffers full.
- **Decode stage 1:** All opcode and addressing-mode information is decoded in the D1 stage. The required information, as well as instruction-length information, is included in at most the first 3 bytes of the instruction. Hence, 3 bytes are passed to the D1 stage from the prefetch buffers. The D1 decoder can then direct the D2 stage to capture the rest of the instruction (displacement and immediate data), which is not involved in the D1 decoding.
- **Decode stage 2:** The D2 stage expands each opcode into control signals for the ALU. It also controls the computation of the more complex addressing modes.
- **Execute:** This stage includes ALU operations, cache access, and register update.

- **Write back:** This stage, if needed, updates registers and status flags modified during the preceding execute stage. If the current instruction updates memory, the computed value is sent to the cache and to the bus-interface write buffers at the same time.

With the use of two decode stages, the pipeline can sustain a throughput of close to one instruction per clock cycle. Complex instructions and conditional branches can slow down this rate.

Figure 14.21 shows examples of the operation of the pipeline. Figure 14.21a shows that there is no delay introduced into the pipeline when a memory access is required. However, as Figure 14.21b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle. In this example, the processor accesses the cache in the EX stage of the first instruction and stores the value retrieved in the register during the WB stage. However, the next instruction needs this register in its D2 stage. When the D2 stage lines up with the WB stage of the previous instruction, bypass signal paths allow the D2 stage to have access to the same data being used by the WB stage for writing, saving one pipeline stage.

Figure 14.21c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).

Fetch	D1	D2	EX	WB		MOV Reg1, Mem1
Fetch	D1	D2	EX	WB		MOV Reg1, Reg2
	Fetch	D1	D2	EX	WB	MOV Mem2, Reg1

(a) No data load delay in the pipeline

Fetch	D1	D2	EX	WB		MOV Reg1, Mem1
Fetch	D1		D2	EX		MOV Reg2, (Reg1)

(b) Pointer load delay

Fetch	D1	D2	EX	WB		CMP Reg1, Imm
Fetch	D1	D2	EX			Jcc Target
	Fetch	D1	D2	EX		Target

(c) Branch instruction timing

**Figure 14.21** 80486 Instruction Pipeline Examples

## 14.5 THE x86 PROCESSOR FAMILY

The x86 organization has evolved dramatically over the years. In this section we examine some of the details of the most recent processor organizations, concentrating on common elements in single processors. Chapter 16 looks at superscalar aspects of the x86, and Chapter 18 examines the multicore organization. An overview of the Pentium 4 processor organization is depicted in Figure 4.18.

### Register Organization

The register organization includes the following types of registers (Table 14.2):

- **General:** There are eight 32-bit general-purpose registers (see Figure 14.3c). These may be used for all types of x86 instructions; they can also hold operands for address calculations. In addition, some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction. As a result, a number of instructions can be

**Table 14.2** x86 Processor Registers

(a) Integer Unit in 32-bit Mode

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode

Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) Floating-Point Unit

Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

encoded more compactly. In 64-bit mode, there are sixteen 64-bit general-purpose registers.

- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The stack segment (SS) register references the segment containing a user-visible stack. The remaining segment registers (DS, ES, FS, GS) enable the user to reference up to four separate data segments at a time.
- **Flags:** The 32-bit EFLAGS register contains condition codes and various mode bits. In 64-bit mode, this register is extended to 64 bits and referred to as RFLAGS. In the current architecture definition, the upper 32 bits of RFLAGS are unused.
- **Instruction pointer:** Contains the address of the current instruction.

There are also registers specifically devoted to the floating-point unit:

- **Numeric:** Each register holds an extended-precision 80-bit floating-point number. There are eight registers that function as a stack, with push and pop operations available in the instruction set.
- **Control:** The 16-bit control register contains bits that control the operation of the floating-point unit, including the type of rounding control; single, double, or extended precision; and bits to enable or disable various exception conditions.
- **Status:** The 16-bit status register contains bits that reflect the current state of the floating-point unit, including a 3-bit pointer to the top of the stack; condition codes reporting the outcome of the last operation; and exception flags.
- **Tag word:** This 16-bit register contains a 2-bit tag for each floating-point numeric register, which indicates the nature of the contents of the corresponding register. The four possible values are valid, zero, special (NaN, infinity, denormalized), and empty. These tags enable programs to check the contents of a numeric register without performing complex decoding of the actual data in the register. For example, when a context switch is made, the processor need not save any floating-point registers that are empty.

The use of most of the aforementioned registers is easily understood. Let us elaborate briefly on several of the registers.

**EFLAGS REGISTER** The EFLAGS register (Figure 14.22) indicates the condition of the processor and helps to control its operation. It includes the six condition codes defined in Table 12.9 (carry, parity, auxiliary, zero, sign, overflow), which report the results of an integer operation. In addition, there are bits in the register that may be referred to as control bits:

- **Trap flag (TF):** When set, causes an interrupt after the execution of each instruction. This is used for debugging.
- **Interrupt enable flag (IF):** When set, the processor will recognize external interrupts.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	I D	V I P	V I F	A C	V M	R F	0	N T	I O P L	O D F F	D I F F	I T F F	S F F	Z F 0	A F 0	P F 1	C F						

X ID = Identification flag  
 X VIP = Virtual interrupt pending  
 X VIF = Virtual interrupt flag  
 X AC = Alignment check  
 X VM = Virtual 8086 mode  
 X RF = Resume flag  
 X NT = Nested task flag  
 X IOPL = I/O privilege level  
 S OF = Overflow flag

C DF = Direction flag  
 X IF = Interrupt enable flag  
 X TF = Trap flag  
 S SF = Sign flag  
 S ZF = Zero flag  
 S AF = Auxiliary carry flag  
 S PF = Parity flag  
 S CF = Carry flag

S indicates a status flag.  
 C indicates a control flag.  
 X indicates a system flag.  
 Shaded bits are reserved.

Figure 14.22 x86 EFLAGS Register

- **Direction flag (DF):** Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).
- **I/O privilege flag (IOPL):** When set, causes the processor to generate an exception on all accesses to I/O devices during protected-mode operation.
- **Resume flag (RF):** Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
- **Alignment check (AC):** Activates if a word or doubleword is addressed on a nonword or nondoubleword boundary.
- **Identification flag (ID):** If this bit can be set and cleared, then this processor supports the processorID instruction. This instruction provides information about the vendor, family, and model.

In addition, there are 4 bits that relate to operating mode. The Nested Task (NT) flag indicates that the current task is nested within another task in protected-mode operation. The Virtual Mode (VM) bit allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine. The Virtual Interrupt Flag (VIF) and Virtual Interrupt Pending (VIP) flag are used in a multitasking environment.

**CONTROL REGISTERS** The x86 employs four control registers (register CR1 is unused) to control various aspects of processor operation (Figure 14.23). All of the registers except CR0 are either 32 bits or 64 bits long, depending on whether the implementation supports the x86 64-bit architecture. The CR0 register contains system control flags, which control modes or indicate states that apply generally

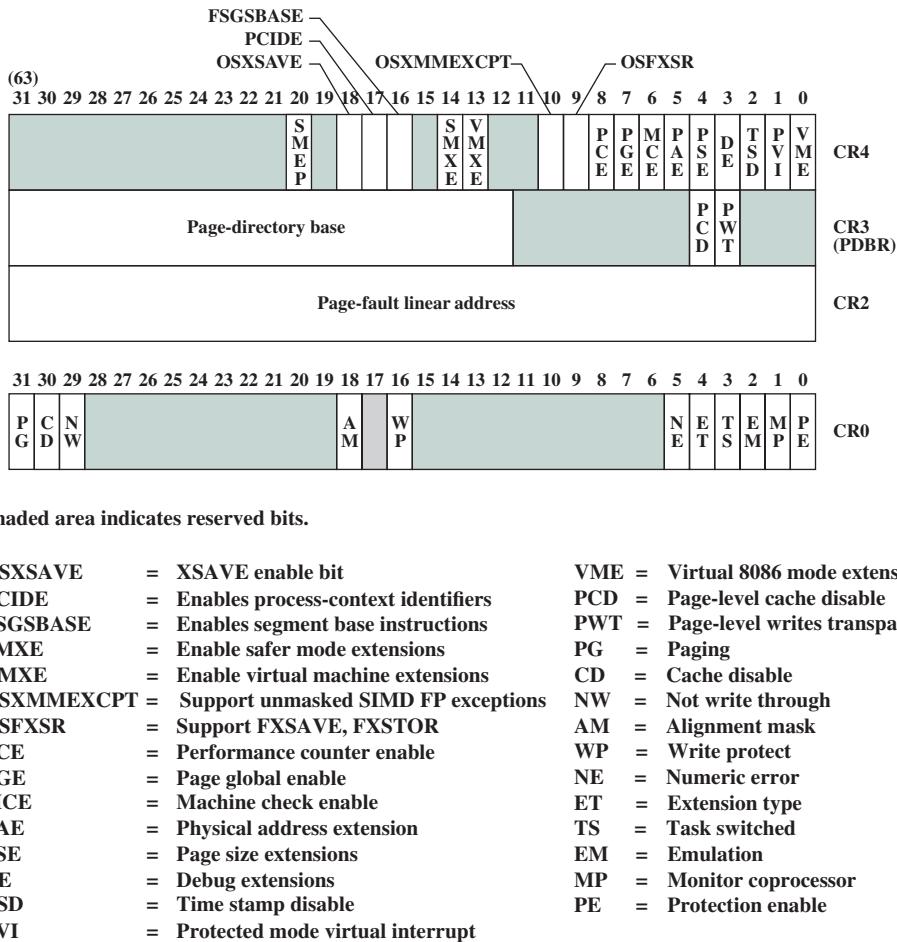


Figure 14.23 x86 Control Registers

to the processor rather than to the execution of an individual task. The flags are as follows:

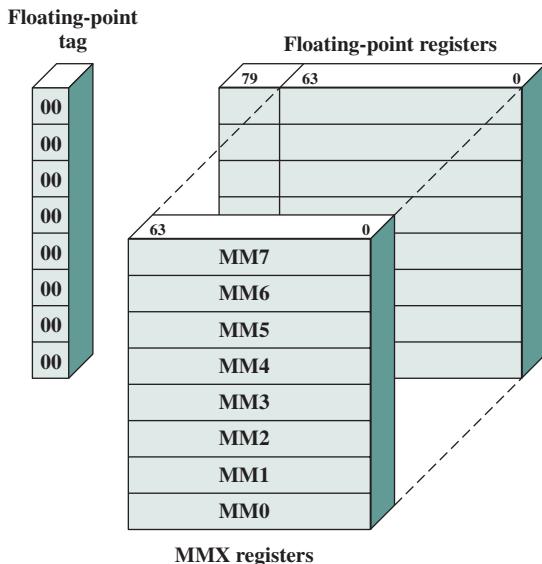
- **Protection Enable (PE):** Enable/disable protected mode of operation.
- **Monitor Coprocessor (MP):** Only of interest when running programs from earlier machines on the x86; it relates to the presence of an arithmetic coprocessor.
- **Emulation (EM):** Set when the processor does not have a floating-point unit, and causes an interrupt when an attempt is made to execute floating-point instructions.
- **Task Switched (TS):** Indicates that the processor has switched tasks.
- **Extension Type (ET):** Not used on the Pentium and later machines; used to indicate support of math coprocessor instructions on earlier machines.

- **Numeric Error (NE):** Enables the standard mechanism for reporting floating-point errors on external bus lines.
- **Write Protect (WP):** When this bit is clear, read-only user-level pages can be written by a supervisor process. This feature is useful for supporting process creation in some operating systems.
- **Alignment Mask (AM):** Enables/disables alignment checking.
- **Not Write Through (NW):** Selects mode of operation of the data cache. When this bit is set, the data cache is inhibited from cache write-through operations.
- **Cache Disable (CD):** Enables/disables the internal cache fill mechanism.
- **Paging (PG):** Enables/disables paging.

When paging is enabled, the CR2 and CR3 registers are valid. The CR2 register holds the 32-bit linear address of the last page accessed before a page fault interrupt. The leftmost 20 bits of CR3 hold the 20 most significant bits of the base address of the page directory; the remainder of the address contains zeros. Two bits of CR3 are used to drive pins that control the operation of an external cache. The page-level cache disable (PCD) enables or disables the external cache, and the page-level writes transparent (PWT) bit controls write through in the external cache. CR4 contains additional control bits.

**MMX REGISTERS** Recall from Section 10.3 that the x86 MMX capability makes use of several 64-bit data types. The MMX instructions make use of 3-bit register address fields, so that eight MMX registers are supported. In fact, the processor does not include specific MMX registers. Rather, the processor uses an aliasing technique (Figure 14.24). The existing floating-point registers are used to store MMX values. Specifically, the low-order 64 bits (mantissa) of each floating-point register are used to form the eight MMX registers. Thus, the older 32-bit x86 architecture is easily extended to support the MMX capability. Some key characteristics of the MMX use of these registers are as follows:

- Recall that the floating-point registers are treated as a stack for floating-point operations. For MMX operations, these same registers are accessed directly.
- The first time that an MMX instruction is executed after any floating-point operations, the FP tag word is marked valid. This reflects the change from stack operation to direct register addressing.
- The EMMS (Empty MMX State) instruction sets bits of the FP tag word to indicate that all registers are empty. It is important that the programmer insert this instruction at the end of an MMX code block so that subsequent floating-point operations function properly.
- When a value is written to an MMX register, bits [79:64] of the corresponding FP register (sign and exponent bits) are set to all ones. This sets the value in the FP register to NaN (not a number) or infinity when viewed as a floating-point value. This ensures that an MMX data value will not look like a valid floating-point value.



**Figure 14.24** Mapping of MMX Registers to Floating-Point Registers

## Interrupt Processing

Interrupt processing within a processor is a facility provided to support the operating system. It allows an application program to be suspended, in order that a variety of interrupt conditions can be serviced and later resumed.

**INTERRUPTS AND EXCEPTIONS** Two classes of events cause the x86 to suspend execution of the current instruction stream and respond to the event: interrupts and exceptions. In both cases, the processor saves the context of the current process and transfers to a predefined routine to service the condition. An *interrupt* is generated by a signal from hardware, and it may occur at random times during the execution of a program. An *exception* is generated from software, and it is provoked by the execution of an instruction. There are two sources of interrupts and two sources of exceptions:

### 1. Interrupts

- **Maskable interrupts:** Received on the processor's INTR pin. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.
- **Nonmaskable interrupts:** Received on the processor's NMI pin. Recognition of such interrupts cannot be prevented.

### 2. Exceptions

- **Processor-detected exceptions:** Results when the processor encounters an error while attempting to execute an instruction.

- **Programmed exceptions:** These are instructions that generate an exception (e.g., INTO, INT3, INT, and BOUND).

**INTERRUPT VECTOR TABLE** Interrupt processing on the x86 uses the interrupt vector table. Every type of interrupt is assigned a number, and this number is used to index into the interrupt vector table. This table contains 256 32-bit interrupt vectors, which is the address (segment and offset) of the interrupt service routine for that interrupt number.

Table 14.3 shows the assignment of numbers in the interrupt vector table; shaded entries represent interrupts, while nonshaded entries are exceptions. The NMI hardware interrupt is type 2. INTR hardware interrupts are assigned numbers in the range of 32 to 255; when an INTR interrupt is generated, it must be accompanied on the bus with the interrupt vector number for this interrupt. The remaining vector numbers are used for exceptions.

If more than one exception or interrupt is pending, the processor services them in a predictable order. The location of vector numbers within the table does not reflect priority. Instead, priority among exceptions and interrupts is organized into five classes. In descending order of priority, these are

- **Class 1:** Traps on the previous instruction (vector number 1)
- **Class 2:** External interrupts (2, 32–255)
- **Class 3:** Faults from fetching next instruction (3, 14)
- **Class 4:** Faults from decoding the next instruction (6, 7)
- **Class 5:** Faults on executing an instruction (0, 4, 5, 8, 10–14, 16, 17)

**INTERRUPT HANDLING** Just as with a transfer of execution using a CALL instruction, a transfer to an interrupt-handling routine uses the system stack to store the processor state. When an interrupt occurs and is recognized by the processor, a sequence of events takes place:

1. If the transfer involves a change of privilege level, then the current stack segment register and the current extended stack pointer (ESP) register are pushed onto the stack.
2. The current value of the EFLAGS register is pushed onto the stack.
3. Both the interrupt (IF) and trap (TF) flags are cleared. This disables INTR interrupts and the trap or single-step feature.
4. The current code segment (CS) pointer and the current instruction pointer (IP or EIP) are pushed onto the stack.
5. If the interrupt is accompanied by an error code, then the error code is pushed onto the stack.
6. The interrupt vector contents are fetched and loaded into the CS and IP or EIP registers. Execution continues from the interrupt service routine.

To return from an interrupt, the interrupt service routine executes an IRET instruction. This causes all of the values saved on the stack to be restored; execution resumes from the point of the interrupt.

**Table 14.3** x86 Exception and Interrupt Vector Table

Vector Number	Description
0	Divide error; division overflow or division by zero
1	Debug exception; includes various faults and traps related to debugging
2	NMI pin interrupt; signal on NMI pin
3	Breakpoint; caused by INT 3 instruction, which is a 1-byte instruction useful for debugging
4	INTO-detected overflow; occurs when the processor executes INTO with the OF flag set
5	BOUND range exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds
6	Undefined opcode
7	Device not available; attempt to use ESC or WAIT instruction fails due to lack of external device
8	Double fault; two interrupts occur during the same instruction and cannot be handled serially
9	Reserved
10	Invalid task state segment; segment describing a requested task is not initialized or not valid
11	Segment not present; required segment not present
12	Stack fault; limit of stack segment exceeded or stack segment not present
13	General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)
14	Page fault
15	Reserved
16	Floating-point error; generated by a floating-point arithmetic instruction
17	Alignment check; access to a word stored at an odd byte address or a doubleword stored at an address not a multiple of 4
18	Machine check; model specific
19–31	Reserved
32–255	User interrupt vectors; provided when INTR signal is activated

Unshaded: exceptions

Shaded: interrupts

## 14.6 THE ARM PROCESSOR

In this section, we look at some of the key elements of the ARM architecture and organization. We defer a discussion of more complex aspects of organization and pipelining until Chapter 16. For the discussion in this section and in Chapter 16, it is useful to keep in mind key characteristics of the ARM architecture. ARM is primarily a RISC system with the following notable attributes:

- A moderate array of uniform registers, more than are found on some CISC systems but fewer than are found on many RISC systems.
- A load/store model of data processing, in which operations only perform on operands in registers and not directly in memory. All data must be loaded into registers before an operation can be performed; the result can then be used for further processing or stored into memory.
- A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set.
- To make each data processing instruction more flexible, either a shift or rotation can preprocess one of the source registers. To efficiently support this feature, there are separate arithmetic logic unit (ALU) and shifter units.
- A small number of addressing modes with all load/store addressees determined from registers and instruction fields. Indirect or indexed addressing involving values in memory are not used.
- Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops.
- Conditional execution of instructions minimizes the need for conditional branch instructions, thereby improving pipeline efficiency, because pipeline flushing is reduced.

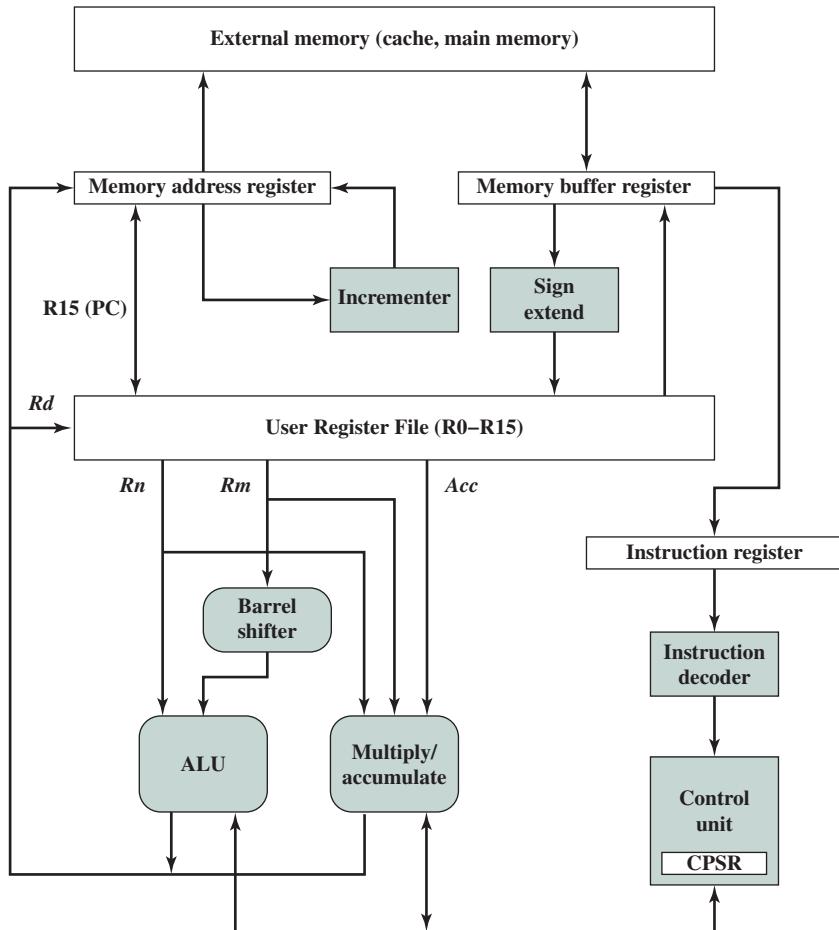
## Processor Organization

The ARM processor organization varies substantially from one implementation to the next, particularly when based on different versions of the ARM architecture. However, it is useful for the discussion in this section to present a simplified, generic ARM organization, which is illustrated in Figure 14.25. In this figure, the arrows indicate the flow of data. Each box represents a functional hardware unit or a storage unit.

Data are exchanged with the processor from external memory through a data bus. The value transferred is either a data item, as a result of a load or store instruction, or an instruction fetch. Fetched instructions pass through an instruction decoder before execution, under control of a control unit. The latter includes pipeline logic and provides control signals (not shown) to all the hardware elements of the processor. Data items are placed in the register file, consisting of a set of 32-bit registers. Byte or halfword items treated as twos-complement numbers are sign-extended to 32 bits.

ARM data processing instructions typically have two source registers,  $Rn$  and  $Rm$ , and a single result or destination register,  $Rd$ . The source register values feed into the ALU or a separate multiply unit that makes use of an additional register to accumulate partial results. The ARM processor also includes a hardware unit that can shift or rotate the  $Rm$  value before it enters the ALU. This shift or rotate occurs within the cycle time of the instruction and increases the power and flexibility of many data processing operations.

The results of an operation are fed back to the destination register. Load/store instructions may also use the output of the arithmetic units to generate the memory address for a load or store.



**Figure 14.25** Simplified ARM Organization

## Processor Modes

It is quite common for a processor to support only a small number of processor modes. For example, many operating systems make use of just two modes: a user mode and a kernel mode, with the latter mode used to execute privileged system software. In contrast, the ARM architecture provides a flexible foundation for operating systems to enforce a variety of protection policies.

The ARM architecture supports seven execution modes. Most application programs execute in **user mode**. While the processor is in user mode, the program being executed is unable to access protected system resources or to change mode, other than by causing an exception to occur.

The remaining six execution modes are referred to as privileged modes. These modes are used to run system software. There are two principal advantages to defining so many different privileged modes: (1) The OS can tailor the use of system software to a variety of circumstances, and (2) certain registers are dedicated for use for each of the privileged modes, allowing swifter changes in context.

The exception modes have full access to system resources and can change modes freely. Five of these modes are known as exception modes. These are entered when specific exceptions occur. Each of these modes has some dedicated registers that substitute for some of the user mode registers, and which are used to avoid corrupting User mode state information when the exception occurs. The exception modes are as follows:

- **Supervisor mode:** Usually what the OS runs in. It is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.
- **Abort mode:** Entered in response to memory faults.
- **Undefined mode:** Entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors.
- **Fast interrupt mode:** Entered whenever the processor receives an interrupt signal from the designated fast interrupt source. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.
- **Interrupt mode:** Entered whenever the processor receives an interrupt signal from any other interrupt source (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

The remaining privileged mode is the **System mode**. This mode is not entered by any exception and uses the same registers available in User mode. The System mode is used for running certain privileged operating system tasks. System mode tasks may be interrupted by any of the five exception categories.

## Register Organization

Figure 14.26 depicts the user-visible registers for the ARM. The ARM processor has a total of 37 32-bit registers, classified as follows:

- Thirty-one registers referred to in the ARM manual as general-purpose registers. In fact, some of these, such as the program counters, have special purposes.
- Six program status registers.

Registers are arranged in partially overlapping banks, with the current processor mode determining which bank is available. At any time, sixteen numbered registers and one or two program status registers are visible, for a total of 17 or 18 software-visible registers. Figure 14.26 is interpreted as follows:

- Registers R0 through R7, register R15 (the program counter) and the current program status register (CPSR) are visible in and shared by all modes.
- Registers R8 through R12 are shared by all modes except fast interrupt, which has its own dedicated registers R8\_fiq through R12\_fiq.
- All the exception modes have their own versions of registers R13 and R14.
- All the exception modes have a dedicated saved program status register (SPSR).

		Modes					
		Privileged modes					
		Exception modes					
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt	
R0	R0	R0	R0	R0	R0	R0	
R1	R1	R1	R1	R1	R1	R1	
R2	R2	R2	R2	R2	R2	R2	
R3	R3	R3	R3	R3	R3	R3	
R4	R4	R4	R4	R4	R4	R4	
R5	R5	R5	R5	R5	R5	R5	
R6	R6	R6	R6	R6	R6	R6	
R7	R7	R7	R7	R7	R7	R7	
R8	R8	R8	R8	R8	R8	R8_fiq	
R9	R9	R9	R9	R9	R9	R9_fiq	
R10	R10	R10	R10	R10	R10	R10_fiq	
R11	R11	R11	R11	R11	R11	R11_fiq	
R12	R12	R12	R12	R12	R12	R12_fiq	
R13(SP)	R13(SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	
R14(LR)	R14(LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer

CPSR = current program status register

LR = link register

SPSR = saved program status register

PC = program counter

**Figure 14.26** ARM Register Organization

**GENERAL-PURPOSE REGISTERS** Register R13 is normally used as a stack pointer and is also known as the SP. Because each exception mode has a separate R13, each exception mode can have its own dedicated program stack. R14 is known as the link register (LR) and is used to hold subroutine return addresses and exception mode returns. Register R15 is the program counter (PC).

**PROGRAM STATUS REGISTERS** The CPSR is accessible in all processor modes. Each exception mode also has a dedicated SPSR that is used to preserve the value of the CPSR when the associated exception occurs.

The 16 most significant bits of the CPSR contain user flags visible in User mode, and which can be used to affect the operation of a program (Figure 14.27). These are as follows:

- **Condition code flags:** The N, Z, C, and V flags, which are discussed in Chapter 12.
- **Q flag:** used to indicate whether overflow and/or saturation has occurred in some SIMD-oriented instructions.
- **J bit:** indicates the use of special 8-bit instructions, known as Jazelle instructions, which are beyond the scope of our discussion.
- **GE[3:0] bits:** SIMD instructions use bits [19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.

The 16 least significant bits of the CPSR contain system control flags that can only be altered when the processor is in a privileged mode. The fields are as follows:

- **E bit:** Controls load and store endianness for data; ignored for instruction fetches.
- **Interrupt disable bits:** The A bit disables imprecise data aborts when set; the I bit disables IRQ interrupts when set; and the F bit disables FIQ interrupts when set.
- **T bit:** Indicates whether instructions should be interpreted as normal ARM instructions or Thumb instructions.
- **Mode bits:** Indicates the processor mode.

## Interrupt Processing

As with any processor, the ARM includes a facility that enables the processor to interrupt the currently executing program to deal with exception conditions. Exceptions are generated by internal and external sources to cause the processor to handle an event. The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time. The ARM architecture supports seven types of exceptions. Table 14.4 lists the types of exception and the processor mode that is used to process each type. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors.

If more than one interrupt is outstanding, they are handled in priority order. Table 14.4 lists the exceptions in priority order, highest to lowest.

When an exception occurs, the processor halts execution after the current instruction. The state of the processor is preserved in the SPSR that corresponds to

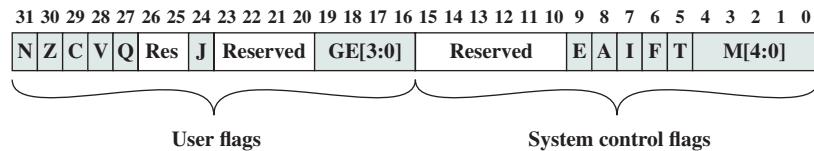


Figure 14.27 Format of ARM CPSR and SPSR

**Table 14.4** ARM Interrupt Vector

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.

the type of exception, so that the original program can be resumed when the exception routine has completed. The address of the instruction the processor was just about to execute is placed in the link register of the appropriate processor mode. To return after handling the exception, the SPSR is moved into the CPSR and R14 is moved into the PC.

## 14.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

### Key Terms

branch prediction condition code delayed branch	flag instruction cycle instruction pipeline	instruction prefetch program status word (PSW)
---	---	---

## Review Questions

- 14.1** What general roles are performed by processor registers?
- 14.2** What categories of data are commonly supported by user-visible registers?
- 14.3** What is the function of condition codes?
- 14.4** What is a program status word?
- 14.5** Why is a two-stage instruction pipeline unlikely to cut the instruction cycle time in half, compared with the use of no pipeline?
- 14.6** List and briefly explain various ways in which an instruction pipeline can deal with conditional branch instructions.
- 14.7** How are history bits used for branch prediction?

## Problems

- 14.1**
  - a. If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?
    - Carry
    - Zero
    - Overflow
    - Sign
    - Even Parity
    - Half-Carry
  - b. Repeat for the addition of  $-1$  (twos complement) and  $+1$ .
- 14.2** Repeat Problem 14.1 for the operation  $A - B$ , where  $A$  contains 11110000 and  $B$  contains 0010100.
- 14.3**
  - a. How long is a clock cycle?
  - b. What is the duration of a particular type of machine instruction consisting of three clock cycles?
- 14.4**
  - A microprocessor provides an instruction capable of moving a string of bytes from one area of memory to another. The fetching and initial decoding of the instruction takes 10 clock cycles. Thereafter, it takes 15 clock cycles to transfer each byte. The microprocessor is clocked at a rate of 10 GHz.
    - a. Determine the length of the instruction cycle for the case of a string of 64 bytes.
    - b. What is the worst-case delay for acknowledging an interrupt if the instruction is noninterruptible?
    - c. Repeat part (b) assuming the instruction can be interrupted at the beginning of each byte transfer.
- 14.5**
  - The Intel 8088 consists of a bus interface unit (BIU) and an execution unit (EU), which form a 2-stage pipeline. The BIU fetches instructions into a 4-byte instruction queue. The BIU also participates in address calculations, fetches operands, and writes results in memory as requested by the EU. If no such requests are outstanding and the bus is free, the BIU fills any vacancies in the instruction queue. When the EU completes execution of an instruction, it passes any results to the BIU (destined for memory or I/O) and proceeds to the next instruction.
    - a. Suppose the tasks performed by the BIU and EU take about equal time. By what factor does pipelining improve the performance of the 8088? Ignore the effect of branch instructions.
    - b. Repeat the calculation assuming that the EU takes twice as long as the BIU.
- 14.6** Assume an 8088 is executing a program in which the probability of a program jump is 0.1. For simplicity, assume that all instructions are 2 bytes long.

- a. What fraction of instruction fetch bus cycles is wasted?
  - b. Repeat if the instruction queue is 8 bytes long.
- 14.7** Consider the timing diagram of Figures 14.10. Assume that there is only a two-stage pipeline (fetch, execute). Redraw the diagram to show how many time units are now needed for four instructions.
- 14.8** Assume a pipeline with four stages: fetch instruction (FI), decode instruction and calculate addresses (DA), fetch operand (FO), and execute (EX). Draw a diagram similar to Figure 14.10 for a sequence of 7 instructions, in which the third instruction is a branch that is taken and in which there are no data dependencies.
- 14.9** A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has five stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instructions and out-of-sequence executions.
  - a. What is the speedup of this processor for this program compared to a nonpipelined processor, making the same assumptions used in Section 14.4?
  - b. What is throughput (in MIPS) of the pipelined processor?
- 14.10** A nonpipelined processor has a clock rate of 2.5 GHz and an average CPI (cycles per instruction) of 4. An upgrade to the processor introduces a five-stage pipeline. However, due to internal pipeline delays, such as latch delay, the clock rate of the new processor has to be reduced to 2 GHz.
  - a. What is the speedup achieved for a typical program?
  - b. What is the MIPS rate for each processor?
- 14.11** Consider an instruction sequence of length  $n$  that is streaming through the instruction pipeline. Let  $p$  be the probability of encountering a conditional or unconditional branch instruction, and let  $q$  be the probability that execution of a branch instruction I causes a jump to a nonconsecutive address. Assume that each such jump requires the pipeline to be cleared, destroying all ongoing instruction processing, when I emerges from the last stage. Revise Equations (14.1) and (14.2) to take these probabilities into account.
- 14.12** One limitation of the multiple-stream approach to dealing with branches in a pipeline is that additional branches will be encountered before the first branch is resolved. Suggest two additional limitations or drawbacks.
- 14.13** Consider the state diagrams of Figure 14.28.
  - a. Describe the behavior of each.
  - b. Compare these with the branch prediction state diagram in Section 14.4. Discuss the relative merits of each of the three approaches to branch prediction.
- 14.14** The Motorola 680x0 machines include the instruction Decrement and Branch According to Condition, which has the following form:

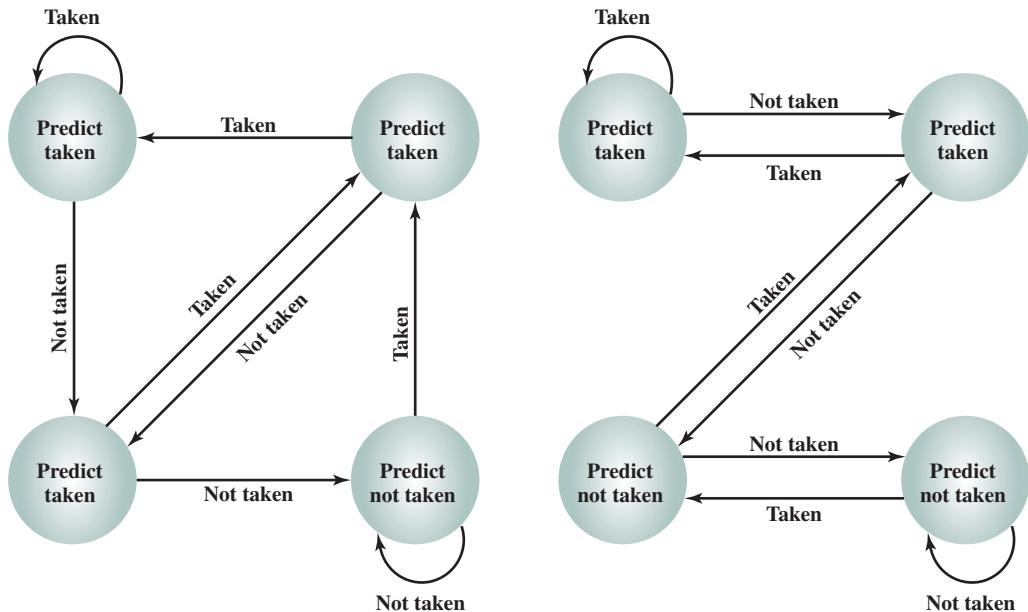
DBcc Dn, displacement

where cc is one of the testable conditions, Dn is a general-purpose register, and displacement specifies the target address relative to the current address. The instruction can be defined as follows:

```

if (cc = False)
then begin
    Dn := (Dn) -1;
    if Dn ≠ -1 then PC := (PC) + displacement end
else PC := (PC) + 2;
```

When the instruction is executed, the condition is first tested to determine whether the termination condition for the loop is satisfied. If so, no operation is performed and execution continues at the next instruction in sequence. If the condition is false, the specified data register is decremented and checked to see if it is less than zero. If it is



**Figure 14.28** Two Branch Prediction State Diagrams

less than zero, the loop is terminated and execution continues at the next instruction in sequence. Otherwise, the program branches to the specified location. Now consider the following assembly-language program fragment:

```

AGAIN      CMPM.L    (A0)+, (A1)+  

          DBNE      D1, AGAIN  

          NOP

```

Two strings addressed by A0 and A1 are compared for equality; the string pointers are incremented with each reference. D1 initially contains the number of longwords (4 bytes) to be compared.

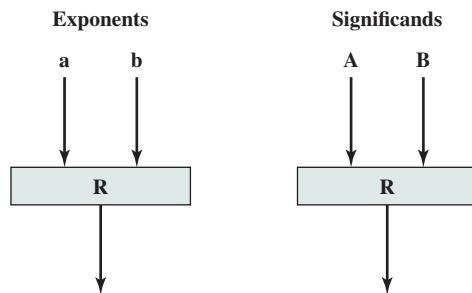
- a. The initial contents of the registers are A0 = \$00004000, A1 = \$00005000 and D1 = \$000000FF (the \$ indicates hexadecimal notation). Memory between \$4000 and \$6000 is loaded with words \$AAAA. If the foregoing program is run, specify the number of times the DBNE loop is executed and the contents of the three registers when the NOP instruction is reached.
- b. Repeat (a), but now assume that memory between \$4000 and \$4FEE is loaded with \$0000 and between \$5000 and \$6000 is loaded with \$AAA.

- 14.15** Redraw Figures 14.19c, assuming that the conditional branch is not taken.
- 14.16** Table 14.5 summarizes statistics from [MACD84] concerning branch behavior for various classes of applications. With the exception of type 1 branch behavior, there is no noticeable difference among the application classes. Determine the fraction of all branches that go to the branch target address for the scientific environment. Repeat for commercial and systems environments.
- 14.17** Pipelining can be applied within the ALU to speed up floating-point operations. Consider the case of floating-point addition and subtraction. In simplified terms, the pipeline could have four stages: (1) Compare the exponents; (2) Choose the exponent and align the significands; (3) Add or subtract significands; (4) Normalize the results. The

**Table 14.5** Branch Behavior in Sample Applications

<b>Occurrence of branch classes:</b>			
Type 1: Branch	72.5%	Scientific	Commercial
Type 2: Loop control	9.8%		
Type 3: Procedure call, return	17.7%		
<b>Type 1 branch: where it goes</b>			
Unconditional—100% go to target	20%	40%	35%
Conditional—went to target	43.2%	24.3%	32.5%
Conditional—did not go to target (inline)	36.8%	35.7%	32.5%
<b>Type 2 branch (all environments)</b>			
That go to target	91%		
That go inline	9%		
<b>Type 3 branch</b>			
100% go to target			

pipeline can be considered to have two parallel threads, one handling exponents and one handling significands, and could start out like this:



In this figure, the boxes labeled R refer to a set of registers used to hold temporary results. Complete the block diagram that shows at a top level the structure of the pipeline.

## BASIC PROCESSING UNIT

### CHAPTER OBJECTIVES

In this chapter you will learn about:

- Execution of instructions by a processor
- The functional units of a processor and how they are interconnected
- Hardware for generating control signals
- Microprogrammed control

In this chapter we focus on the processing unit, which executes machine-language instructions and coordinates the activities of other units in a computer. We examine its internal structure and show how it performs the tasks of fetching, decoding, and executing such instructions. The processing unit is often called the *central processing unit* (CPU). The term “central” is not as appropriate today as it was in the past, because today’s computers often include several processing units. We will use the term *processor* in this discussion.

The organization of processors has evolved over the years, driven by developments in technology and the desire to provide high performance. To achieve high performance, it is prudent to make various functional units of a processor operate in parallel as much as possible. Such processors have a *pipelined* organization where the execution of an instruction is started before the execution of the preceding instruction is completed. Another approach, known as *superscalar* operation, is to fetch and start the execution of several instructions at the same time. Pipelining and superscalar approaches are discussed in Chapter 6. In this chapter, we concentrate on the basic ideas that are common to all processors.

## 5.1 SOME FUNDAMENTAL CONCEPTS

A typical computing task consists of a series of operations specified by a sequence of machine-language instructions that constitute a program. The processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor uses the *program counter*, PC, to keep track of the address of the next instruction to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence. A branch instruction may cause a different value to be loaded into the PC.

When an instruction is fetched, it is placed in the *instruction register*, IR, from where it is interpreted, or decoded, by the processor’s control circuitry. The IR holds the instruction until its execution is completed.

Consider a 32-bit computer in which each instruction is contained in one word in the memory, as in RISC-style instruction set architecture. To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is

$$\text{IR} \leftarrow [\text{PC}]$$

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is

$$\text{PC} \leftarrow [\text{PC}] + 4$$

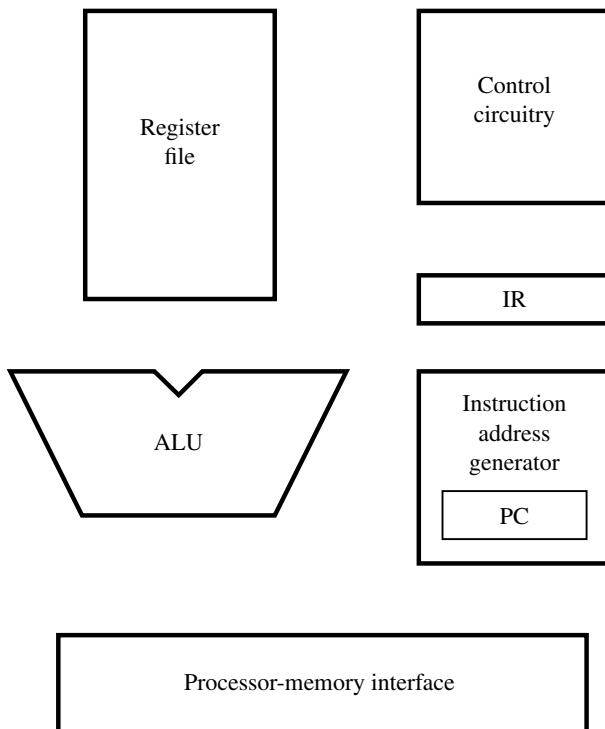
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the *instruction fetch phase*. Performing the operation specified in the instruction constitutes the *instruction execution phase*.

With few exceptions, the operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

The hardware components needed to perform these actions are shown in Figure 5.1. The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations. The instruction address generator updates the contents of the PC after every instruction is fetched. The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers. During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic



**Figure 5.1** Main hardware components of a processor.

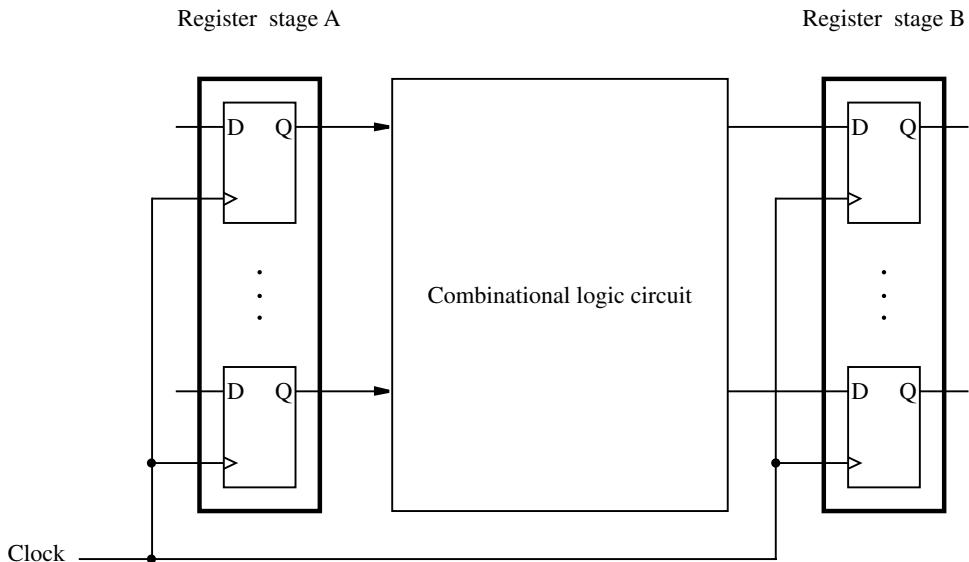
unit (ALU), which performs the required computation. The results of the computation are stored in a register in the register file.

Before we examine these units and their interaction in detail, it is helpful to consider the general structure of any data processing system.

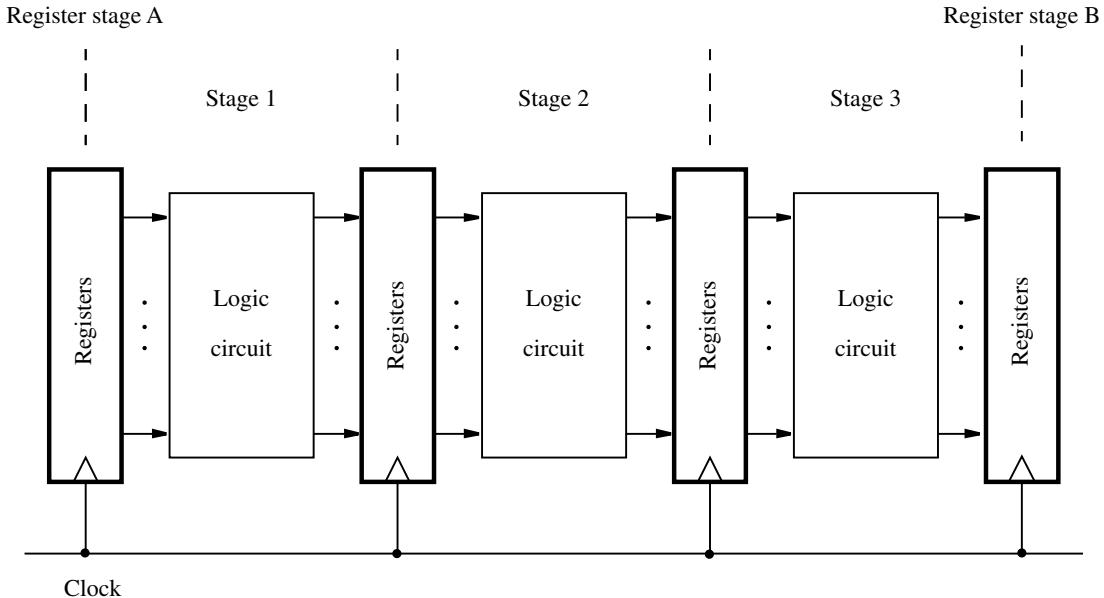
### Data Processing Hardware

A typical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. Figure 5.2 illustrates this structure. A clock signal is used to control the timing of data transfers. The registers comprise edge-triggered flip-flops into which new data are loaded at the active edge of the clock. In this chapter, we assume that the rising edge of the clock is the active edge. The clock period, which is the time between two successive rising edges, must be long enough to allow the combinational circuit to produce the correct result.

The operation performed by the combinational block in Figure 5.2 may be quite complex. It can often be broken down into several simpler steps, where each step is performed by a subcircuit of the original circuit. These subcircuits can then be cascaded into a multi-stage structure as shown in Figure 5.3. Then, if  $n$  stages are used, the operation will be completed in  $n$  clock cycles. Since these combinational subcircuits are smaller, they can complete their operation in less time, and hence a shorter clock period can be used. A key advantage of the multi-stage structure is that it is suitable for pipelined operation, as will be discussed in Chapter 6. Such a structure is particularly useful for implementing processors that have a RISC-style instruction set. The discussion in the remainder of this chapter focuses on processors that use a multi-stage structure of this type. In Section 5.7 we will consider a more traditional alternative that is suitable for CISC-style processors.



**Figure 5.2** Basic structure for data processing.



**Figure 5.3** A hardware structure with multiple stages.

## 5.2 INSTRUCTION EXECUTION

Let us now examine the actions involved in fetching and executing instructions. We illustrate these actions using a few representative RISC-style instructions.

### 5.2.1 LOAD INSTRUCTIONS

Consider the instruction

Load R5, X(R7)

which uses the Index addressing mode to load a word of data from memory location  $X + [R7]$  into register R5. Execution of this instruction involves the following actions:

- Fetch the instruction from the memory.
- Increment the program counter.
- Decode the instruction to determine the operation to be performed.
- Read register R7.
- Add the immediate value X to the contents of R7.
- Use the sum  $X + [R7]$  as the effective address of the source operand, and read the contents of that location in the memory.
- Load the data received from the memory into the destination register, R5.

Depending on how the hardware is organized, some of these actions can be performed at the same time. In the discussion that follows, we will assume that the processor has five hardware stages, which is a commonly used arrangement in RISC-style processors. Execution of each instruction is divided into five steps, such that each step is carried out by one hardware stage. In this case, fetching and executing the Load instruction above can be completed as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of register R7 in the register file.
3. Compute the effective address.
4. Read the memory source operand.
5. Load the operand into the destination register, R5.

### 5.2.2 ARITHMETIC AND LOGIC INSTRUCTIONS

Instructions that involve an arithmetic or logic operation can be executed using similar steps. They differ from the Load instruction in two ways:

- There are either two source registers, or a source register and an immediate source operand.
- No access to memory operands is required.

A typical instruction of this type is

Add R3, R4, R5

It requires the following steps:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of source registers R4 and R5.
3. Compute the sum  $[R4] + [R5]$ .
4. Load the result into the destination register, R3.

The Add instruction does not require access to an operand in the memory, and therefore could be completed in four steps instead of the five steps needed for the Load instruction. However, as we will see in the next chapter, it is advantageous to use the same multi-stage processing hardware for as many instructions as possible. This can be achieved if we arrange for all instructions to be executed in the same number of steps. To this end, the Add instruction should be extended to five steps, patterned along the steps of the Load instruction. Since no access to memory operands is required, we can insert a step in which no action takes place between steps 3 and 4 above. The Add instruction would then be performed as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R4 and R5.
3. Compute the sum  $[R4] + [R5]$ .

4. No action.
5. Load the result into the destination register, R3.

If the instruction uses an immediate operand, as in

Add R3, R4, #1000

the immediate value is given in the instruction word. Once the instruction is loaded into the IR, the immediate value is available for use in the addition operation. The same five-step sequence can be used, with steps 2 and 3 modified as:

2. Decode the instruction and read register R4.
3. Compute the sum [R4] + 1000.

### 5.2.3 STORE INSTRUCTIONS

The five-step sequence used for the Load and Add instructions is also suitable for Store instructions, except that the final step of loading the result into a destination register is not required. The hardware stage responsible for this step takes no action. For example, the instruction

Store R6, X(R8)

stores the contents of register R6 into memory location  $X + [R8]$ . It can be implemented as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R6 and R8.
3. Compute the effective address  $X + [R8]$ .
4. Store the contents of register R6 into memory location  $X + [R8]$ .
5. No action.

After reading register R8 in step 2, the memory address is computed in step 3 using the immediate value, X, in the IR. In step 4, the contents of R6 are sent to the memory to be stored. No action is taken in step 5.

In summary, the five-step sequence of actions given in Figure 5.4 is suitable for all instructions in a RISC-style instruction set. RISC-style instructions are one word long and only Load and Store instructions access operands in the memory, as explained in Chapter 2. Instructions that perform computations use data that are either stored in general-purpose registers or given as immediate data in the instruction.

The five-step sequence is suitable for all Load and Store instructions, because the addressing modes that can be used in these instructions are special cases of the Index mode. Most RISC-style processors provide one general-purpose register, usually register R0, that always contains the value zero. When R0 is used as the index register, the effective address of the operand is the immediate value X. This is the Absolute addressing mode. Alternatively, if the offset X is set to zero, the effective address is the contents of the index register,  $R_i$ . This is the Indirect addressing mode. Thus, only one addressing mode, the Index mode,

---

Step	Action
1	Fetch an instruction and increment the program counter.
2	Decode the instruction and read registers from the register file.
3	Perform an ALU operation.
4	Read or write memory data if the instruction involves a memory operand.
5	Write the result into the destination register, if needed.

---

**Figure 5.4** A five-step sequence of actions to fetch and execute an instruction.

needs to be implemented, resulting in a significant simplification of the processor hardware. The task of selecting R0 as the index register or setting X to zero is left to the assembler or the compiler. This is consistent with the RISC philosophy of aiming for simple and fast hardware at the expense of higher compiler complexity and longer compilation time. The result is a net gain in the time needed to perform various tasks on a computer, because programs are compiled much less frequently than they are executed.

---

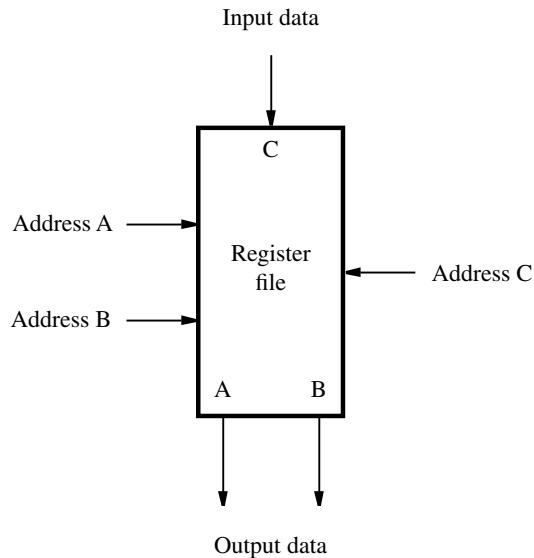
## 5.3 HARDWARE COMPONENTS

The discussion above indicates that all instructions of a RISC-style processor can be executed using the five-step sequence in Figure 5.4. Hence, the processor hardware may be organized in five stages, such that each stage performs the actions needed in one of the steps. We now examine the components in Figure 5.1 to see how they may be organized in the multi-stage structure of Figure 5.3.

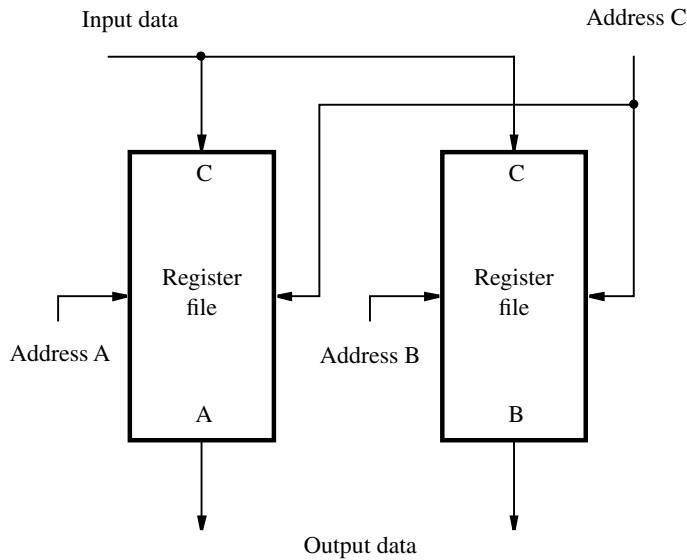
### 5.3.1 REGISTER FILE

General-purpose registers are usually implemented in the form of a register file, which is a small and fast memory block. It consists of an array of storage elements, with access circuitry that enables data to be read from or written into any register. The access circuitry is designed to enable two registers to be read at the same time, making their contents available at two separate outputs, A and B. The register file has two address inputs that select the two registers to be read. These inputs are connected to the fields in the IR that specify the source registers, so that the required registers can be read. The register file also has a data input, C, and a corresponding address input to select the register into which data are to be written. This address input is connected to the IR field that specifies the destination register of the instruction.

The inputs and outputs of any memory unit are often called input and output *ports*. A memory unit that has two output ports is said to be *dual-ported*. Figure 5.5 shows two ways



(a) Single memory block



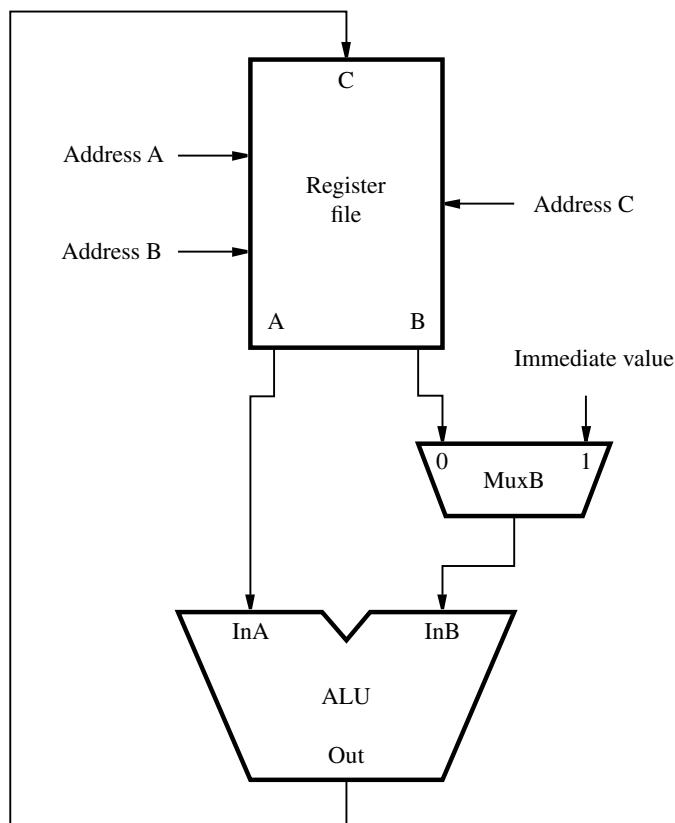
(b) Two memory blocks

**Figure 5.5** Two alternatives for implementing a dual-ported register file.

of realizing a dual-ported register file. One possibility is to use a single set of registers with duplicate data paths and access circuitry that enable two registers to be read at the same time. An alternative is to use two memory blocks, each containing one copy of the register file. Whenever data are written into a register, they are written into both copies of that register. Thus, the two files have identical contents. When an instruction requires data from two registers, one register is accessed in each file. In effect, the two register files together function as a single dual-ported register file.

### 5.3.2 ALU

The arithmetic and logic unit is used to manipulate data. It performs arithmetic operations such as addition and subtraction, and logic operations such as AND, OR, and XOR. Conceptually, the register file and the ALU may be connected as shown in Figure 5.6. When an instruction that performs an arithmetic or logic operation is being executed, the contents of the two registers specified in the instruction are read from the register file and become



**Figure 5.6** Conceptual view of the hardware needed for computation.

available at outputs A and B. Output A is connected directly to the first input of the ALU, InA, and output B is connected to a multiplexer, MuxB. The multiplexer selects either output B of the register file or the immediate value in the IR to be connected to the second ALU input, InB. The output of the ALU is connected to the data input, C, of the register file so that the results of a computation can be loaded into the destination register.

### 5.3.3 DATAPATH

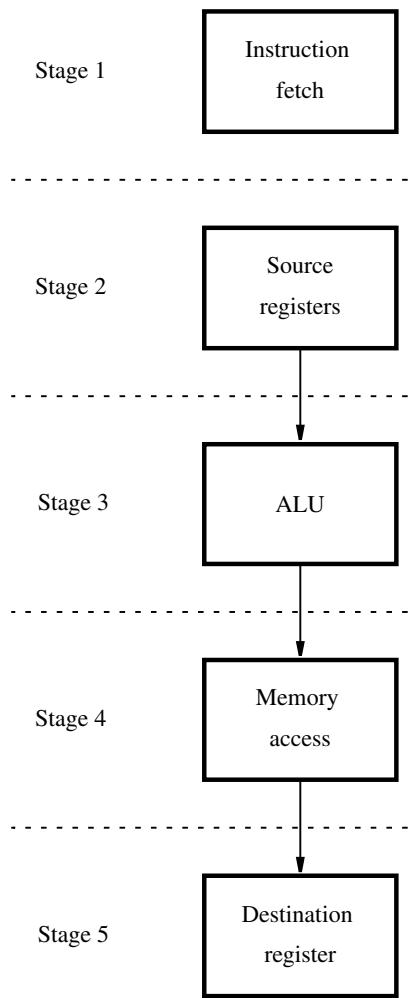
Instruction processing consists of two phases: the fetch phase and the execution phase. It is convenient to divide the processor hardware into two corresponding sections. One section fetches instructions and the other executes them. The section that fetches instructions is also responsible for decoding them and for generating the control signals that cause appropriate actions to take place in the execution section. The execution section reads the data operands specified in an instruction, performs the required computations, and stores the results.

We now need to organize the hardware into a multi-stage structure similar to that in Figure 5.3, with stages corresponding to the five steps in Figure 5.4. A possible structure is shown in Figure 5.7. The actions taken in each of the five stages are completed in one clock cycle. An instruction is fetched in step 1 by hardware stage 1 and placed into the IR. It is decoded, and its source registers are read in step 2. The information in the IR is used to generate the control signals for all subsequent steps. Therefore, the IR must continue to hold the instruction until its execution is completed.

It is necessary to insert registers between stages. Inter-stage registers hold the results produced in one stage so that they can be used as inputs to the next stage during the next clock cycle. This leads to the organization in Figure 5.8. The hardware in the figure is often referred to as the *datapath*. It corresponds to stages 2 to 5 in Figure 5.7. Data read from the register file are placed in registers RA and RB. Register RA provides the data to input InA of the ALU. Multiplexer MuxB forwards either the contents of RB or the immediate value in the IR to the ALU's second input, InB. The ALU constitutes stage 3, and the result of the computation it performs is placed in register RZ.

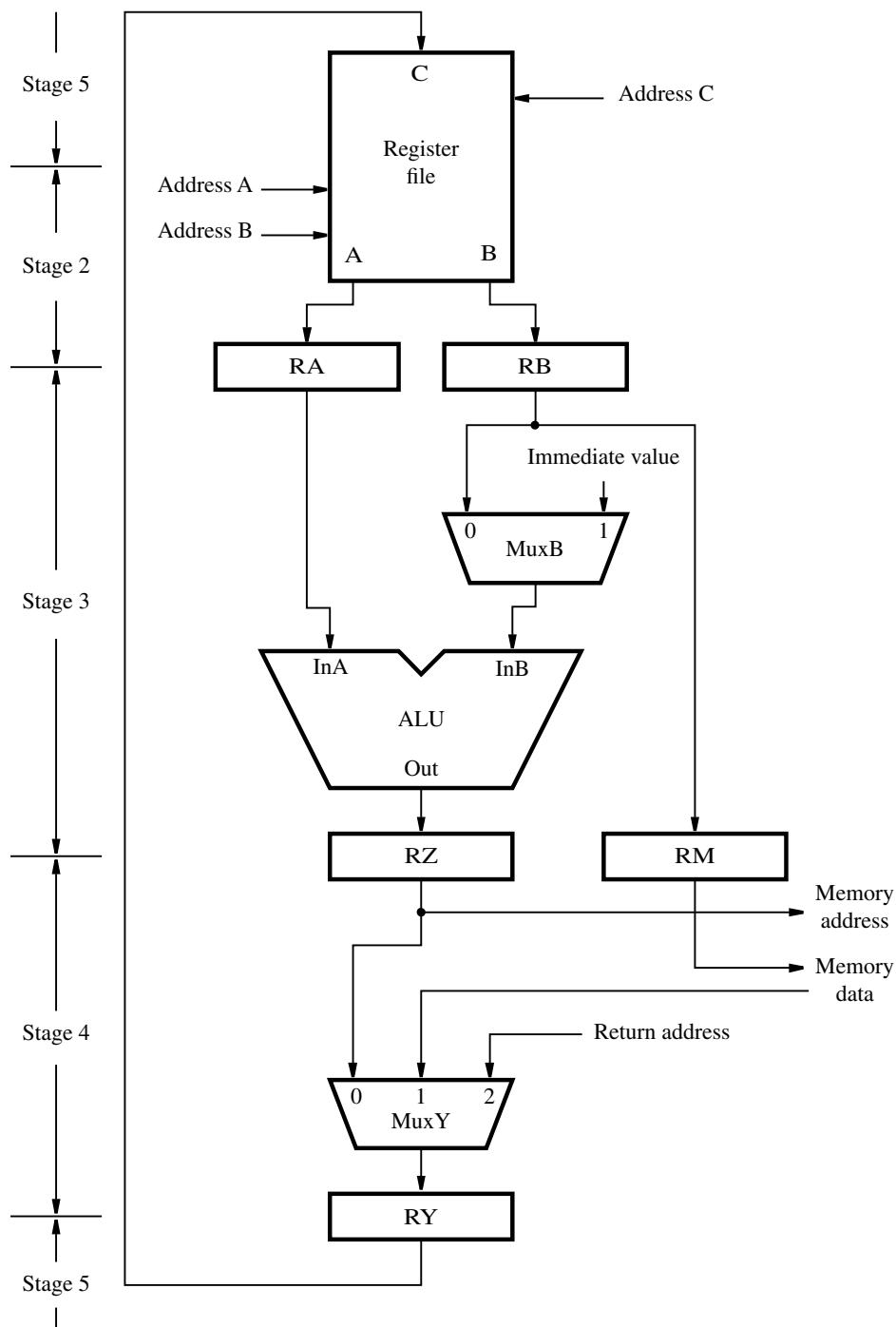
Recall that for computational instructions, such as an Add instruction, no processing actions take place in step 4. During that step, multiplexer MuxY in Figure 5.8 selects register RZ to transfer the result of the computation to RY. The contents of RY are transferred to the register file in step 5 and loaded into the destination register. For this reason, the register file is in both stages 2 and 5. It is a part of stage 2 because it contains the source registers and a part of stage 5 because it contains the destination register.

For Load and Store instructions, the effective address of the memory operand is computed by the ALU in step 3 and loaded into register RZ. From there, it is sent to the memory, which is stage 4. In the case of a Load instruction, the data read from the memory are selected by multiplexer MuxY and placed in register RY, to be transferred to the register file in the next clock cycle. For a Store instruction, data are read from the register file, which is part of stage 2, and placed in register RB. Since memory access is done in stage 4, another inter-stage register is needed to maintain correct data flow in the multi-stage structure. Register RM is introduced for this purpose. The data to be stored are moved from RB to RM in step 3, and from there to the memory in step 4. No action is taken in step 5 in this case.



**Figure 5.7** A five-stage organization.

The subroutine call instructions introduced in Section 2.7 save the return address in a general-purpose register, which we call LINK for ease of reference. Similarly, interrupt processing requires a return address to be saved, as described in Section 3.2. Assume that another general-purpose register, IRA, is used for this purpose. Both of these actions require the contents of the program counter to be sent to the register file. For this reason, multiplexer MuxY has a third input through which the return address can be routed to register RY, from where it can be sent to the register file. The return address is produced by the instruction address generator, as we will explain later.



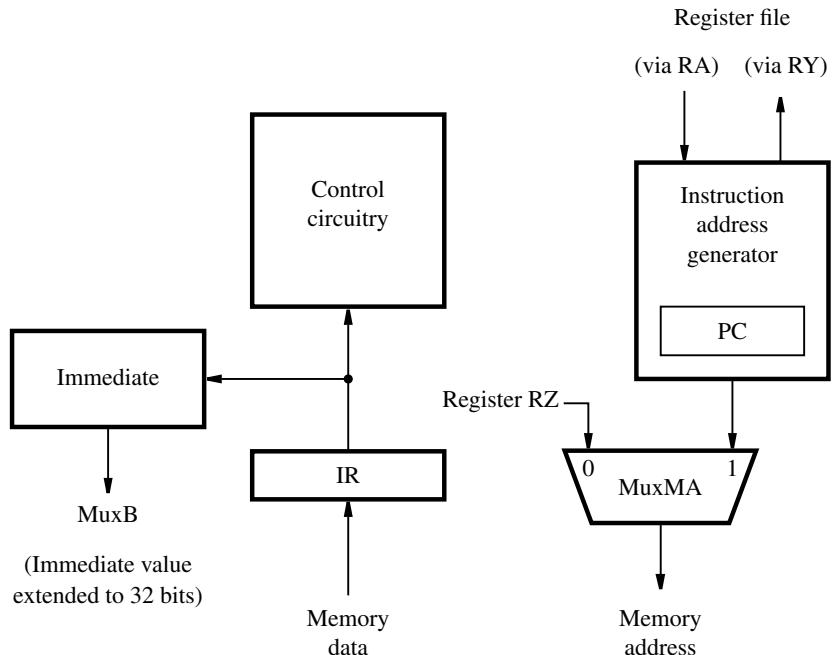
**Figure 5.8** Datapath in a processor.

### 5.3.4 INSTRUCTION FETCH SECTION

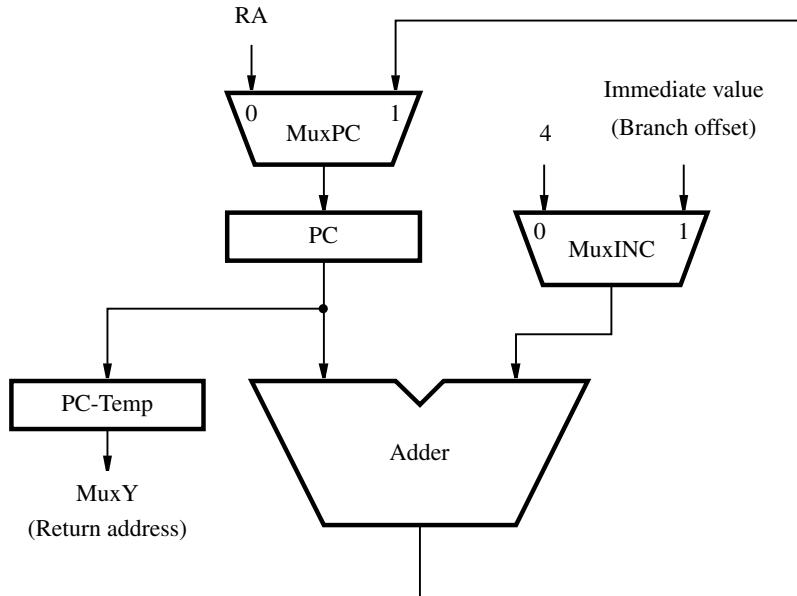
The organization of the instruction fetch section of the processor is illustrated in Figure 5.9. The addresses used to access the memory come from the PC when fetching instructions and from register RZ in the datapath when accessing instruction operands. Multiplexer MuxMA selects one of these two sources to be sent to the processor-memory interface. The PC is included in a larger block, the instruction address generator, which updates the contents of the PC after each instruction is fetched. The instruction read from the memory is loaded into the IR, where it stays until its execution is completed and the next instruction is fetched.

The contents of the IR are examined by the control circuitry to generate the signals needed to control all the processor's hardware. They are also used by the block labeled Immediate. As described in Chapter 2, an immediate value may be included in some instructions. A 16-bit immediate value is extended to 32 bits. The extended value is then used either directly as an operand or to compute the effective address of an operand. For some instructions, such as those that perform arithmetic operations, the immediate value is sign-extended; for others, such as logic instructions, it is padded with zeros. The Immediate block in Figure 5.9 generates the extended value and forwards it to MuxB in Figure 5.8 to be used in an ALU computation. It also generates the extended value to be used in computing the target address of branch instructions.

The address generator circuit is shown in Figure 5.10. An adder is used to increment the PC by 4 during straight-line execution. It is also used to compute a new value to be



**Figure 5.9** Instruction fetch section of Figure 5.7.



**Figure 5.10** Instruction address generator.

loaded into the PC when executing branch and subroutine call instructions. One adder input is connected to the PC. The second input is connected to a multiplexer, MuxINC, which selects either the constant 4 or the branch offset to be added to the PC. The branch offset is given in the immediate field of the IR and is sign-extended to 32 bits by the Immediate block in Figure 5.9. The output of the adder is routed to the PC via a second multiplexer, MuxPC, which selects between the adder and the output of register RA. The latter connection is needed when executing subroutine linkage instructions. Register PC-Temp is needed to hold the contents of the PC temporarily during the process of saving the subroutine or interrupt return address.

## 5.4 INSTRUCTION FETCH AND EXECUTION STEPS

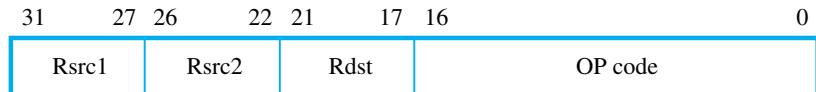
We now examine the process of fetching and executing instructions in more detail, using the datapath in Figure 5.8. Consider again the instruction

Add R3, R4, R5

The steps for fetching and executing this instruction are given in Figure 5.11. Assume that the instruction is encoded using the format in Figure 2.32, which is reproduced here as Figure 5.12. After the instruction has been fetched from the memory and placed in the IR, the source register addresses are available in fields  $IR_{31-27}$  and  $IR_{26-22}$ . These two fields

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction, RA $\leftarrow$ [R4], RB $\leftarrow$ [R5]
3	RZ $\leftarrow$ [RA] + [RB]
4	RY $\leftarrow$ [RZ]
5	R3 $\leftarrow$ [RY]

**Figure 5.11** Sequence of actions needed to fetch and execute the instruction: Add R3, R4, R5.



(a) Register-operand format



(b) Immediate-operand format



(c) Call format

**Figure 5.12** Instruction encoding.

are connected to the address inputs for ports A and B of the register file. As a result, registers R4 and R5 are read and their contents placed in registers RA and RB, respectively, at the end of step 2. In the next step, the control circuitry sets MuxB to select input 0, thus connecting register RB to input InB of the ALU. At the same time, it causes the ALU to perform an addition operation. Since register RA is connected to input InA, the ALU produces the required sum [RA] + [RB], which is loaded into register RZ at the end of step 3.

In step 4, multiplexer MuxY selects input 0, thus causing the contents of RZ to be transferred to RY. The control circuitry connects the destination address field of the Add instruction, IR<sub>21-17</sub>, to the address input for port C of the register file. In step 5, it issues

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction, RA $\leftarrow$ [R7]
3	RZ $\leftarrow$ [RA] + Immediate value X
4	Memory address $\leftarrow$ [RZ], Read memory, RY $\leftarrow$ Memory data
5	R5 $\leftarrow$ [RY]

**Figure 5.13** Sequence of actions needed to fetch and execute the instruction: Load R5, X(R7).

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction, RA $\leftarrow$ [R8], RB $\leftarrow$ [R6]
3	RZ $\leftarrow$ [RA] + Immediate value X, RM $\leftarrow$ [RB]
4	Memory address $\leftarrow$ [RZ], Memory data $\leftarrow$ [RM], Write memory
5	No action

**Figure 5.14** Sequence of actions needed to fetch and execute the instruction: Store R6, X(R8).

a Write command to the register file, causing the contents of register RY to be written into register R3.

Load and Store instructions are executed in a similar manner. In this case, the address of the destination register is given in bit field IR<sub>26–22</sub>. The control hardware connects this field to the address input corresponding to input C of the register file. The steps involved in executing these instructions are given in Figures 5.13 and 5.14. In both examples, the memory address is specified using the Index mode, in which the index value X is given as an immediate field of IR, extended as appropriate by the Immediate block in Figure 5.9, is selected by MuxB in step 3 and added to the contents of register RA. The resulting sum is the effective address of the operand.

### Some Observations

In the discussion above, we assumed that memory Read and Write operations can be completed in one clock cycle. Is this a realistic assumption? In general, accessing the main memory of a computer takes significantly longer than reading the contents of a register in the register file. However, most modern processors use cache memories, which will be discussed in detail in Chapter 8. A cache memory is much faster than the main memory.

It is usually implemented on the same chip as the processor, making it about as fast as the register file. Thus, a memory Read or Write operation can be completed in one clock cycle when the data involved are available in the cache. When the operation requires access to the main memory, the processor must wait for that operation to be completed. We will discuss how slower memory accesses are handled in Section 5.4.2.

We also assumed that the processor reads the source registers of the instruction in step 2, while it is still decoding the OP code of the instruction that has just been loaded into the IR. Can these two tasks be completed in the same step? How can the control hardware know which registers to read before it completes decoding the instruction? This is possible because source register addresses are specified using the same bit positions in all instructions. The hardware reads the registers whose addresses are in these bit positions once the instruction is loaded into the IR. Their contents are loaded into registers RA and RB at the end of step 2. If these data are needed by the instruction, they will be available for use in step 3. If not, they will be ignored by subsequent hardware stages.

Note that the actions described in Figures 5.11, 5.13, and 5.14 do not show two registers being read in step 2 in every case. To avoid confusion, only the registers needed by the specific instruction described in the figure are mentioned, even though two registers are always read.

### 5.4.1 BRANCHING

Instructions are fetched from sequential word locations in the memory during straight-line program execution. Whenever an instruction is fetched, the processor increments the PC by 4 to point to the next word. This execution pattern continues until a branch or subroutine call instruction loads a new address into the PC. Subroutine call instructions also save the return address, to be used when returning to the calling program. In this section we examine the actions needed to implement these instructions. Interrupts from I/O devices and software interrupt instructions are handled in a similar manner.

Branch instructions specify the branch target address relative to the PC. A branch offset given as an immediate value in the instruction is added to the current contents of the PC. The number of bits used for this offset is considerably less than the word length of the computer, because space is needed within the instruction to specify the OP code and the branch condition. Hence, the range of addresses that can be reached by a branch instruction is limited.

Subroutine call instructions can reach a larger range of addresses. Because they do not include a condition, more bits are available to specify the target address. Also, most RISC-style computers have Jump and Call instructions that use a general-purpose register to specify a full 32-bit address. The details vary from one computer to another, as the example processors introduced in Appendices B to E illustrate.

#### Branch Instructions

The sequence of steps for implementing an unconditional branch instruction is given in Figure 5.15. The instruction is fetched and the PC is incremented as usual in step 1. After the instruction has been decoded in step 2, multiplexer MuxINC selects the branch offset in

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction
3	PC $\leftarrow$ [PC] + Branch offset
4	No action
5	No action

**Figure 5.15** Sequence of actions needed to fetch and execute an unconditional branch instruction.

the IR to be added to the PC in step 3. This is the address that will be used to fetch the next instruction. Execution of a Branch instruction is completed in step 3. No action is taken in steps 4 and 5.

We explained in Section 2.13 that the branch offset is the distance between the branch target and the memory location following the branch instruction. The reason for this can be seen clearly in Figure 5.15. The PC is incremented by 4 in step 1, at the time the branch instruction is fetched. Then, the branch target address is computed in step 3 by adding the branch offset to the updated contents of the PC.

The sequence in Figure 5.15 can be readily modified to implement conditional branch instructions. In processors that do not use condition-code flags, the branch instruction specifies a compare-and-test operation that determines the branch condition. For example, the instruction

Branch\_if\_[R5]=[R6]      LOOP

results in a branch if the contents of registers R5 and R6 are identical. When this instruction is executed, the register contents are compared, and if they are equal, a branch is made to location LOOP.

Figure 5.16 shows how this instruction may be executed. Registers R5 and R6 are read in step 2, as usual, and compared in step 3. The comparison could be done by performing the subtraction operation  $[R5] - [R6]$  in the ALU. The ALU generates signals that indicate whether the result of the subtraction is positive, negative, or zero. The ALU may also generate signals to show whether arithmetic overflow has occurred and whether the operation produced a carry-out. The control circuitry examines these signals to test the condition given in the branch instruction. In the example above, it checks whether the result of the subtraction is equal to zero. If it is, the branch target address is loaded into the PC, to be used to fetch the next instruction. Otherwise, the contents of the PC remain at the incremented value computed in step 1, and straight-line execution continues.

According to the sequence of steps in Figure 5.16, the two actions of comparing the register contents and testing the result are both carried out in step 3. Hence, the clock cycle must be long enough for the two actions to be completed, one after the other. For this reason, it is desirable that the comparison be done as quickly as possible. A subtraction

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction, RA $\leftarrow$ [R5], RB $\leftarrow$ [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC $\leftarrow$ [PC] + Branch offset
4	No action
5	No action

**Figure 5.16** Sequence of actions needed to fetch and execute the instruction:  
Branch\_if\_[R5]=[R6] LOOP.

operation in the ALU is time consuming, and is not needed in this case. A simpler and faster comparator circuit can examine the contents of registers RA and RB and produce the required condition signals, which indicate the conditions greater than, equal, less than, etc. A comparator is not shown separately in Figure 5.8 as it can be a part of the ALU block. Example 5.3 shows how a comparator circuit can be designed.

#### Subroutine Call Instructions

Subroutine calls and returns are implemented in a similar manner to branch instructions. The address of the subroutine may either be computed using an immediate value given in the instruction or it may be given in full in one of the general-purpose registers. Figure 5.17 gives the sequence of actions for the instruction

Call\_Register R9

which calls a subroutine whose address is in register R9. The contents of that register are read and placed in RA in step 2. During step 3, multiplexer MuxPC selects its 0 input, thus transferring the data in register RA to be loaded into the PC.

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction, RA $\leftarrow$ [R9]
3	PC-Temp $\leftarrow$ [PC], PC $\leftarrow$ [RA]
4	RY $\leftarrow$ [PC-Temp]
5	Register LINK $\leftarrow$ [RY]

**Figure 5.17** Sequence of actions needed to fetch and execute the instruction:  
Call\_Register R9.

Assume that the return address of the subroutine, which is the previous contents of the PC, is to be saved in a general-purpose register called LINK in the register file. Data are written into the register file in step 5. Hence, it is not possible to send the return address directly to the register file in step 3. To maintain correct data flow in the five-stage structure, the processor saves the return address in a temporary register, PC-Temp. From there, the return address is transferred to register RY in step 4, then to register LINK in step 5. The address LINK is built into the control circuitry.

Subroutine return instructions transfer the value saved in register LINK back to the PC. The encoding of the Return-from-subroutine instruction is such that the address of register LINK appears in bits IR<sub>31–27</sub>. This is the field connected to Address A of the register file. Hence, once the instruction is fetched, register LINK is read and its contents are placed in RA, from where they can be transferred to the PC via MuxPC in Figure 5.10. Return-from-interrupt instructions are handled in a similar manner, except that a different register is used to hold the return address.

## 5.4.2 WAITING FOR MEMORY

The role of the processor-memory interface circuit is to control data transfers between the processor and the memory. We pointed out earlier that modern processors use fast, on-chip cache memories. Most of the time, the instruction or data referenced in memory Read and Write operations are found in the cache, in which case the operation is completed in one clock cycle. When the requested information is not in the cache and has to be fetched from the main memory, several clock cycles may be needed. The interface circuit must inform the processor's control circuitry about such situations, to delay subsequent execution steps until the memory operation is completed.

Assume that the processor-memory interface circuit generates a signal called Memory Function Completed (MFC). It asserts this signal when a requested memory Read or Write operation has been completed. The processor's control circuitry checks this signal during any processing step in which it issues a memory Read or Write request, to determine when it can proceed to the next step. When the requested data are found in the cache, the interface circuit asserts the MFC signal before the end of the same clock cycle in which the memory request is issued. Hence, instruction execution continues uninterrupted. If access to the main memory is required, the interface circuit delays asserting MFC until the operation is completed. In this case, the processor's control circuitry must extend the duration of the execution step for as many clock cycles as needed, until MFC is asserted. We will use the command Wait for MFC to indicate that a given execution step must be extended, if necessary, until a memory operation is completed. When MFC is received, the actions specified in the step are completed, and the processor proceeds to the next step in the execution sequence.

Step 1 of the execution sequence of any instruction involves fetching the instruction from the memory. Therefore, it must include a Wait for MFC command, as follows:

Memory address  $\leftarrow$  [PC], Read memory, Wait for MFC,  
IR  $\leftarrow$  Memory data, PC  $\leftarrow$  [PC] + 4

The Wait for MFC command is also needed in step 4 of Load and Store instructions in Figures 5.13 and 5.14. Most of the time, the requested information is found in the cache, so the MFC signal is generated quickly, and the step is completed in one clock cycle. When an access involves the main memory, the MFC response is delayed, and the step is extended to several clock cycles.

---

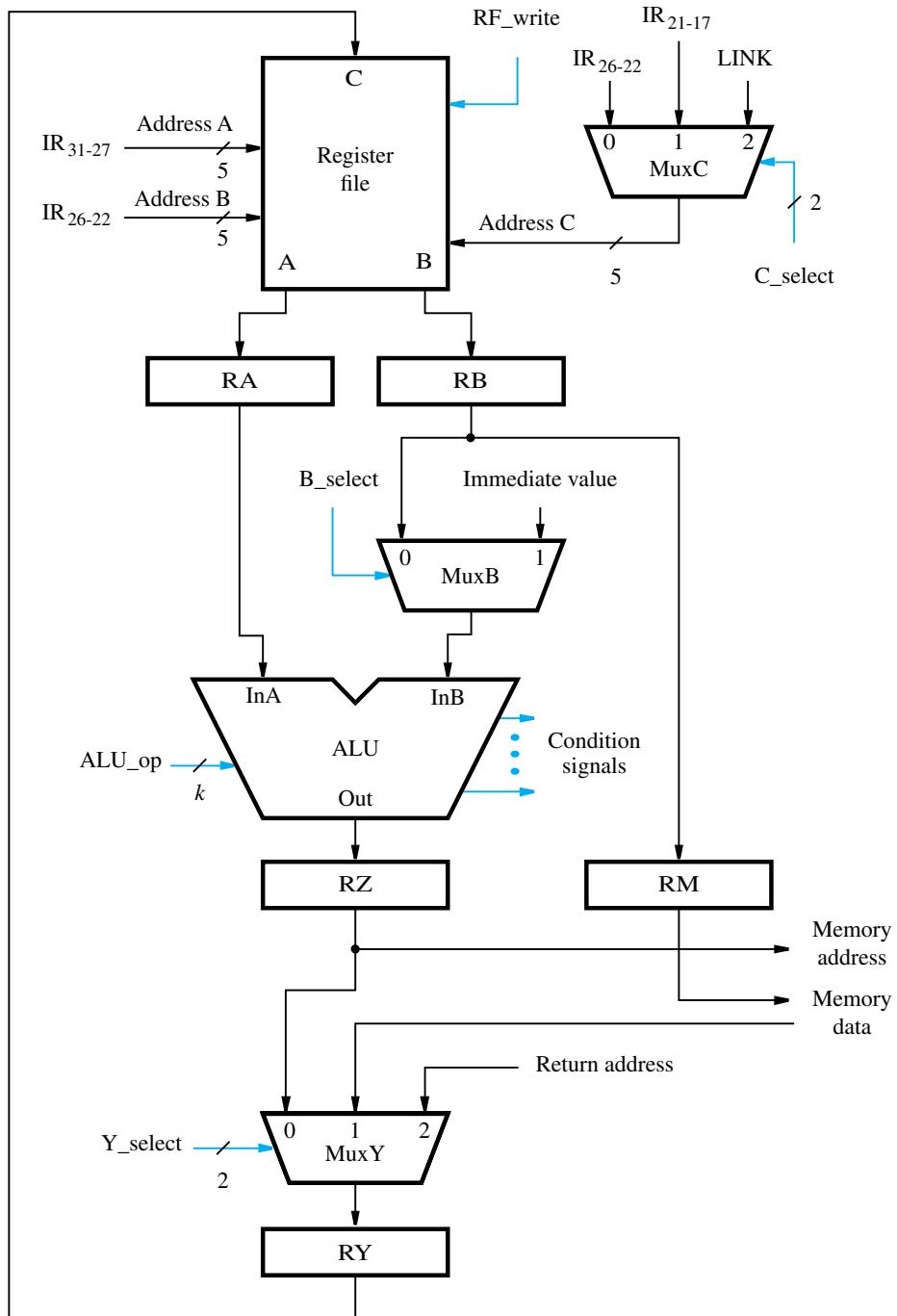
## 5.5 CONTROL SIGNALS

The operation of the processor's hardware components is governed by *control signals*. These signals determine which multiplexer input is selected, what operation is performed by the ALU, and so on. In this section we discuss the signals needed to control the operation of the components in Figures 5.8 to 5.10.

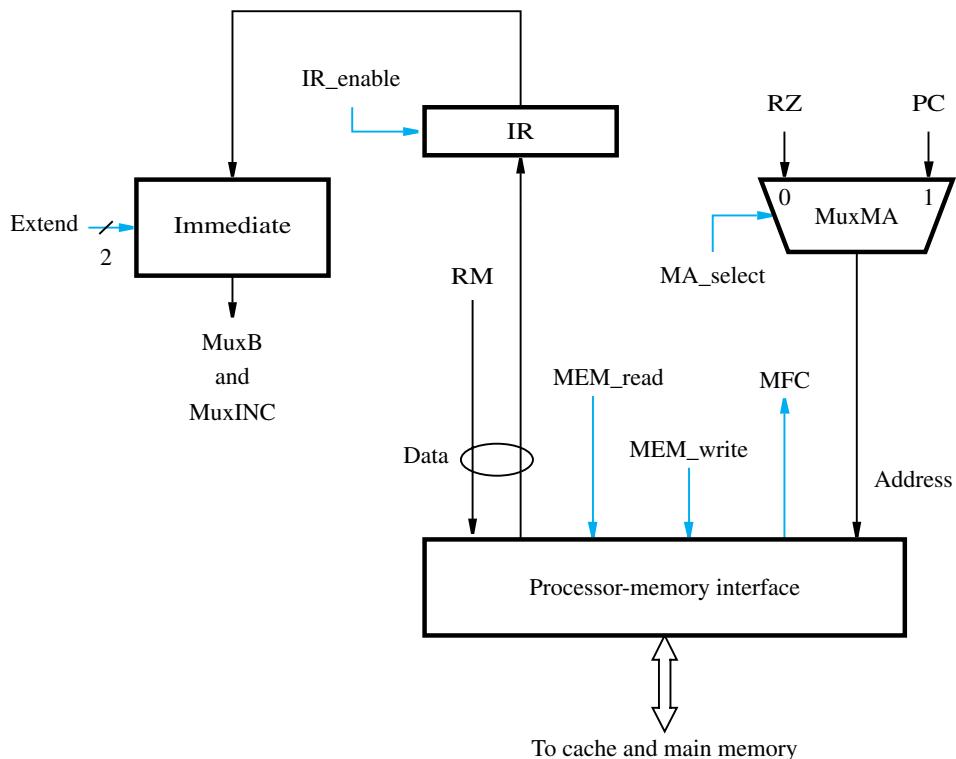
It is instructive to begin by recalling how data flow through the four stages of the datapath, as described in Section 5.3.3. In each clock cycle, the results of the actions that take place in one stage are stored in inter-stage registers, to be available for use by the next stage in the next clock cycle. Since data are transferred from one stage to the next in every clock cycle, inter-stage registers are always enabled. This is the case for registers RA, RB, RZ, RY, RM, and PC-Temp. The contents of the other registers, namely, the PC, the IR, and the register file, must not be changed in every clock cycle. New data are loaded into these registers only when called for in a particular processing step. They must be enabled only at those times.

The role of the multiplexers is to select the data to be operated on in any given stage. For example, MuxB in stage 3 of Figure 5.8 selects the immediate field in the IR for instructions that use an immediate source operand. It also selects that field for instructions that use immediate data as an offset when computing the effective address of a memory operand. Otherwise, it selects register RB. The data selected by the multiplexer are used by the ALU. Examination of Figures 5.11, 5.13, and 5.14 shows that the ALU is used only in step 3, and hence the selection made by MuxB matters only during that step. To simplify the required control circuit, the same selection can be maintained in all execution steps. A similar observation can be made about MuxY. However, MuxMA in Figure 5.9 must change its selection in different execution steps. It selects the PC as the source of the memory address during step 1, when a new instruction is being fetched. During step 4 of Load and Store instructions, it selects register RZ, which contains the effective address of the memory operand.

Figures 5.18, 5.19, and 5.20 show the required control signals. The register file has three 5-bit address inputs, allowing access to 32 general-purpose registers. Two of these inputs, Address A and Address B, determine which registers are to be read. They are connected to fields  $IR_{31-27}$  and  $IR_{26-22}$  in the instruction register. The third address input, Address C, selects the destination register, into which the input data at port C are to be written. Multiplexer MuxC selects the source of that address. We have assumed that three-register instructions use bits  $IR_{21-17}$  and other instructions use  $IR_{26-22}$  to specify the destination register, as in Figure 5.12. The third input of the multiplexer is the address of the link register used in subroutine linkage instructions. New data are loaded into the selected register only when the control signal RF\_write is asserted.



**Figure 5.18** Control signals for the datapath.

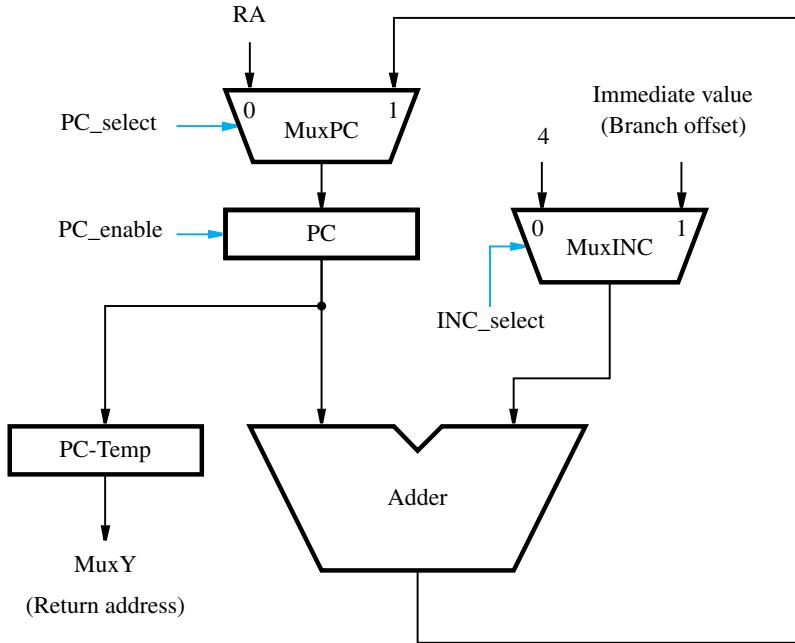


**Figure 5.19** Processor-memory interface and IR control signals.

Multiplexers are controlled by signals that select which input data appear at the multiplexer's output. For example, when B\_select is equal to 0, MuxB selects the contents of register RB to be available at input InB of the ALU. Note that two bits are needed to control MuxC and MuxY, because each multiplexer selects one of three inputs.

The operation performed by the ALU is determined by a  $k$ -bit control code, ALU\_op, which can specify up to  $2^k$  distinct operations, such as Add, Subtract, AND, OR, and XOR. When an instruction calls for two values to be compared, a comparator performs the comparison specified, as mentioned earlier. The comparator generates condition signals that indicate the result of the comparison. These signals are examined by the control circuitry during the execution of conditional branch instructions to determine whether the branch condition is true or false.

The interface between the processor and the memory and the control signals associated with the instruction register are presented in Figure 5.19. Two signals, MEM\_read and MEM\_write are used to initiate a memory Read or a memory Write operation. When the requested operation has been completed, the interface asserts the MFC signal. The instruction register has a control signal, IR\_enable, which enables a new instruction to be loaded into the register. During a fetch step, it must be activated only after the MFC signal is asserted.



**Figure 5.20** Control signals for the instruction address generator.

We have assumed that the Immediate block handles three possible formats for the immediate value: a sign-extended 16-bit value, a zero-extended 16-bit value, and a 26-bit value that is handled in a special way (see Problem 5.14). Hence, its control signal, Extend, comprises two bits.

The signals that control the operation of the instruction address generator are shown in Figure 5.20. The INC\_select signal selects the value to be added to the PC, either the constant 4 or the branch offset specified in the instruction. The PC\_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC\_enable control signal is activated.

## 5.6 HARDWIRED CONTROL

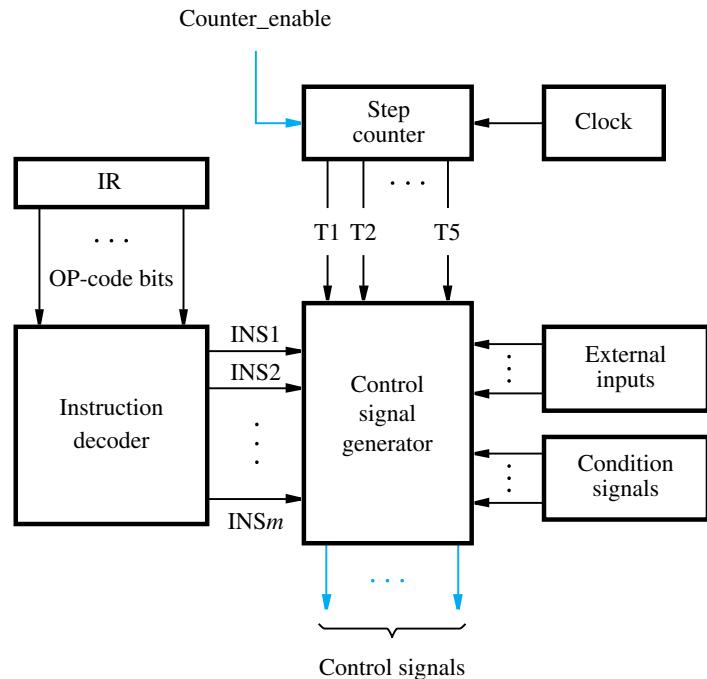
Previous sections described the actions needed to fetch and execute instructions. We now examine how the processor generates the control signals that cause these actions to take place in the correct sequence and at the right time. There are two basic approaches: hardwired control and microprogrammed control. Hardwired control is discussed in this section.

An instruction is executed in a sequence of steps, where each step requires one clock cycle. Hence, a step counter may be used to keep track of the progress of execution. Several

actions are performed in each step, depending on the instruction being executed. In some cases, such as for branch instructions, the actions taken depend on tests applied to the result of a computation or a comparison operation. External signals, such as interrupt requests, may also influence the actions to be performed. Thus, the setting of the control signals depends on:

- Contents of the step counter
- Contents of the instruction register
- The result of a computation or a comparison operation
- External input signals, such as interrupt requests

The circuitry that generates the control signals may be organized as shown in Figure 5.21. The instruction decoder interprets the OP-code and addressing mode information in the IR and sets to 1 the corresponding  $INS_i$  output. During each clock cycle, one of the outputs  $T_1$  to  $T_5$  of the step counter is set to 1 to indicate which of the five steps involved in fetching and executing instructions is being carried out. Since all instructions are completed in five steps, a modulo-5 counter may be used. The control signal generator is a combinational circuit that produces the necessary control signals based on all its inputs. The required settings of the control signals can be determined from the action sequences that implement each of the instructions represented by the signals  $INS_1$  to  $INS_m$ .



**Figure 5.21** Generation of the control signals.

As an example, consider step 1 in the instruction execution process. This is the step in which a new instruction is fetched from the memory. It is identified by signal T1 being asserted. During that clock period, the MA\_select signal in Figure 5.19 is set to 1 to select the PC as the source of the memory address, and MEM\_read is activated to initiate a memory Read operation. The data received from the memory are loaded into the IR by activating IR\_enable when the memory's response signal, MFC, is asserted. At the same time, the PC is incremented by 4, by setting the INC\_select signal in Figure 5.20 to 0 and PC\_select to 1. The PC\_enable signal is activated to cause the new value to be loaded into the PC at the positive edge of the clock marking the end of step T1.

### 5.6.1 DATAPATH CONTROL SIGNALS

Instructions that handle data include Load, Store, and all computational instructions. They perform various data movement and manipulation operations using the processor's datapath, whose control signals are shown in Figures 5.18 and 5.19. Once an instruction is loaded into the IR, the instruction decoder interprets its contents to determine the actions needed. At the same time, the source registers are read and their contents become available at the A and B outputs of the register file. As mentioned earlier, inter-stage registers RA, RB, RZ, RM, and RY are always enabled. This means that data flow automatically from one datapath stage to the next on every active edge of the clock signal.

The desired setting of various control signals can be determined by examining the actions taken in each execution step of every instruction. For example, the RF\_write signal is set to 1 in step T5 during execution of an instruction that writes data into the register file. It may be generated by the logic expression

$$\text{RF\_write} = \text{T5} \cdot (\text{ALU} + \text{Load} + \text{Call})$$

where ALU stands for all instructions that perform arithmetic or logic operations, Load stands for all Load instructions, and Call stands for all subroutine-call and software-interrupt instructions. The RF\_write signal is a function of both the instruction and the timing signals. But, as mentioned earlier, the setting of some of the multiplexers need not change from one timing step to another. In this case, the multiplexer's select signal can be implemented as a function of the instruction only. For example,

$$\text{B\_select} = \text{Immediate}$$

where Immediate stands for all instructions that use an immediate value in the IR. We encourage the reader to examine other control signals and derive the appropriate logic expressions for them, based on the execution steps of various instructions.

### 5.6.2 DEALING WITH MEMORY DELAY

The timing signals T1 to T5 are asserted in sequence as the step counter is advanced. Most of the time, the step counter is incremented at the end of every clock cycle. However, a step

in which a MEM\_read or a MEM\_write command is issued does not end until the MFC signal is asserted, indicating that the requested memory operation has been completed.

To extend the duration of an execution step to more than one clock cycle, we need to disable the step counter. Assume that the counter is incremented when enabled by a control signal called Counter\_enable. Let the need to wait for a memory operation to be completed be indicated by a control signal called WMFC, which is activated during any execution step in which the Wait for MFC command is issued. Counter\_enable should be set to 1 in any step in which WMFC is not asserted. Otherwise, it should be set to 1 when MFC is asserted. This means that

$$\text{Counter\_enable} = \overline{\text{WMFC}} + \text{MFC}$$

A new value is loaded into the PC at the end of any clock cycle in which the PC\_enable signal in Figure 5.20 is activated. We must ensure that the PC is incremented only once when an execution step is extended for more than one clock cycle. Hence, when fetching an instruction, the PC should be enabled only when MFC is received. It is also enabled in step 3 of instructions that cause branching. Let BR denote all instructions in this group. Then, PC\_enable may be realized as

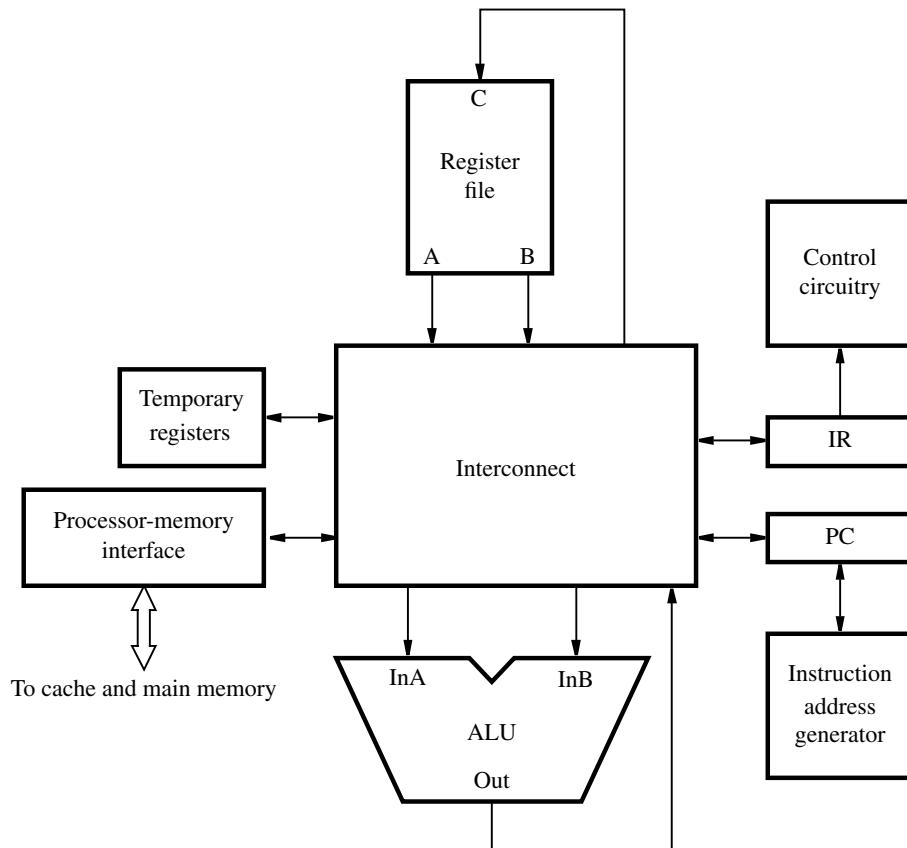
$$\text{PC\_enable} = \text{T1} \cdot \text{MFC} + \text{T3} \cdot \text{BR}$$

## 5.7 CISC-STYLE PROCESSORS

We saw in the previous sections that a RISC-style instruction set is conducive to a multi-stage implementation of the processor. All instructions can be executed in a uniform manner using the same five-stage hardware. As a result, the hardware is simple and well suited to pipelined operation. Also, the control signals are easy to generate.

CISC-style instruction sets are more complex because they allow much greater flexibility in accessing instruction operands. Unlike RISC-style instruction sets, where only Load and Store instructions access data in the memory, CISC instructions can operate directly on memory operands. Also, they are not restricted to one word in length. An instruction may use several words to specify operand addresses and the actions to be performed, as explained in Section 2.10. Therefore, CISC-style instructions require a different organization of the processor hardware.

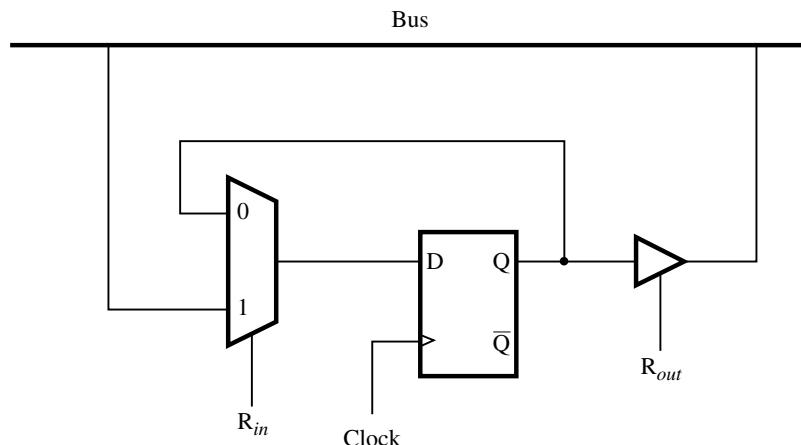
Figure 5.22 shows a possible processor organization. The main difference between this organization and the five-stage structure discussed earlier is that the Interconnect block, which provides interconnections among other blocks, does not prescribe any particular structure or pattern of data flow. It provides paths that make it possible to transfer data between any two components, as needed to implement instructions. The multi-stage structure of Figure 5.8 uses inter-stage registers, such as RZ and RY. These are not needed in the organization of Figure 5.22. Instead, some registers are needed to hold intermediate results during instruction execution. The temporary registers block in the figure is provided for this purpose. It includes two temporary registers, Temp1 and Temp2. The need for these registers will become apparent from the examples given later.



**Figure 5.22** Organization of a CISC-style processor.

A traditional approach to the implementation of the Interconnect is to use buses. A *bus* consists of a set of lines to which several devices may be connected, enabling data to be transferred from any one device to any other. A logic gate that sends a signal over a bus line is called a *bus driver*. Since all devices connected to the bus have the ability to send data, we must ensure that only one of them is driving the bus at any given time. For this reason, the bus driver is a special type of logic gate called a *tri-state gate*. It has a control input that turns it on or off. When turned on, the gate places a logic signal of 0 or 1 on the bus, according to the value of its input. When turned off, the gate is electrically disconnected from the bus, as explained in Appendix A.

Figure 5.23 shows how a flip-flop that forms one bit of a data register can be connected to a bus line. There are two control signals,  $R_{in}$  and  $R_{out}$ . When  $R_{in}$  is equal to 1 the multiplexer selects the data on the bus line to be loaded into the flip-flop. Setting  $R_{in}$  to 0 causes the flip-flop to maintain its present value. The output of the flip-flop is connected to the bus line through a tri-state gate, which is turned on when  $R_{out}$  is asserted. At other times, the tri-state gate is turned off, allowing other components to drive the bus line.



**Figure 5.23** Input and output gating for one register bit.

### 5.7.1 AN INTERCONNECT USING BUSES

The Interconnect in Figure 5.22 may be implemented using one or more buses. Figure 5.24 shows a three-bus implementation. All registers are assumed to be edge-triggered. That is, when a register is enabled, data are loaded into it on the active edge of the clock at the end of the clock period. Addresses for the three ports of the register file are provided by the Control block. These connections are not shown to keep the figure simple. Also not shown is the Immediate block through which the IR is connected to bus B. This is the circuit that extends an immediate operand in the IR to 32 bits.

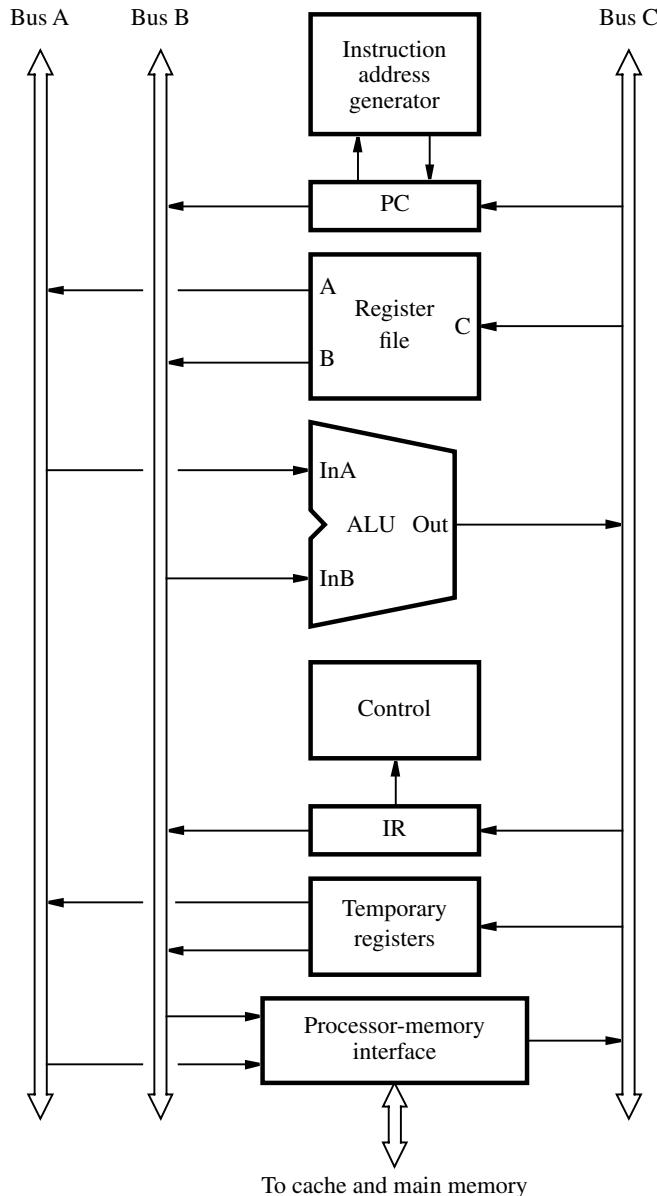
Consider the two-operand instruction

Add R5, R6

which performs the operation

$$R5 \leftarrow [R5] + [R6]$$

Fetching and executing this instruction using the hardware in Figure 5.24 can be performed in three steps, as shown in Figure 5.25. Each step, except for the step involving access to the memory, is completed in one clock cycle. In step 1, bus B is used to send the contents of the PC to the processor-memory interface, which sends them on the memory address lines and initiates a memory Read operation. The data received from the memory, which represent an instruction to be executed, are sent to the IR over bus C. The command Wait for MFC is included to accommodate the possibility that memory access may take more than one clock cycle, as explained in Section 5.4.2. The instruction is decoded in step 2 and the control circuitry begins reading the source registers, R5 and R6. However, the contents of the registers do not become available at the A and B outputs of the register file until step 3. They are sent to the ALU using buses A and B. The ALU performs the addition operation, and the sum is sent back to the ALU over bus C, to be written into register R5 at the end of the clock cycle.



**Figure 5.24** Three-bus CISC-style processor organization.

Note that reading the source registers is completed in step 2 in Figure 5.11. In that case, the action of reading the registers proceeds in parallel with the action of decoding the instruction, because the location of the bit fields containing register addresses in a RISC-style instruction is known. Since CISC-style instructions do not always use the same

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, Wait for MFC, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction
3	R5 $\leftarrow$ [R5] + [R6]

**Figure 5.25** Sequence of actions needed to fetch and execute the instruction: Add R5, R6.

instruction fields to specify register addresses, the action of reading the source registers does not begin until the instruction has been at least partially decoded. Hence, it may not be possible to complete reading the source registers in step 2.

Next, consider the instruction

And X(R7), R9

which performs the logical AND operation on the contents of register R9 and memory location X + [R7] and stores the result back in the same memory location. Assume that the index offset X is a 32-bit value given as the second word of the instruction. To execute this instruction, it is necessary to access the memory four times. First, the OP-code word is fetched. Then, when the instruction decoding circuit recognizes the Index addressing mode, the index offset X is fetched. Next, the memory operand is fetched and the AND operation is performed. Finally, the result is stored back into the memory.

Figure 5.26 gives the steps needed to execute the instruction. After decoding the instruction in step 2, the second word of the instruction is read in step 3. The data received,

Step	Action
1	Memory address $\leftarrow$ [PC], Read memory, Wait for MFC, IR $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
2	Decode instruction
3	Memory address $\leftarrow$ [PC], Read memory, Wait for MFC, Temp1 $\leftarrow$ Memory data, PC $\leftarrow$ [PC] + 4
4	Temp2 $\leftarrow$ [Temp1] + [R7]
5	Memory address $\leftarrow$ [Temp2], Read memory, Wait for MFC, Temp1 $\leftarrow$ Memory data
6	Temp1 $\leftarrow$ [Temp1] AND [R9]
7	Memory address $\leftarrow$ [Temp2], Memory data $\leftarrow$ [Temp1], Write memory, Wait for MFC

**Figure 5.26** Sequence of actions needed to fetch and execute the instruction: And X(R7), R9.

which represent the offset X, are stored temporarily in register Temp1, to be used in the next step for computing the effective address of the memory operand. In step 4, the contents of registers Temp1 and R7 are sent to the ALU inputs over buses A and B. The effective address is computed and placed into register Temp2, then used to read the operand in step 5. Register Temp1 is used again during step 5, this time to hold the data operand received from the memory. The computation is performed in step 6, and the result is placed back in register Temp1. In the final step, the result is sent to be stored in the memory at the operand address, which is still available in register Temp2.

The two examples in Figures 5.25 and 5.26 illustrate the variability in the number of execution steps in CISC-style instructions. There is no uniform sequence of actions that can be followed for all instructions in the same way as was demonstrated for RISC instructions in Section 5.2.

### 5.7.2 MICROPROGRAMMED CONTROL

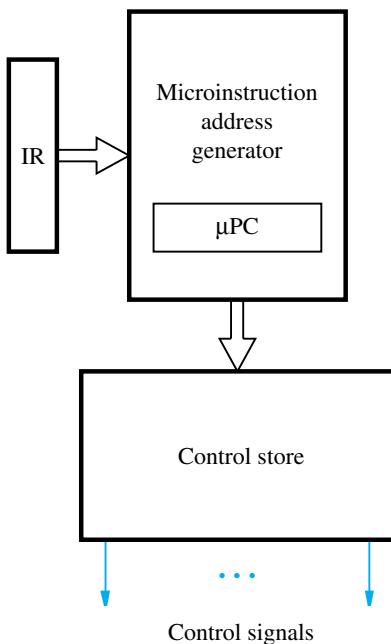
The control signals needed to control the operation of the components in Figures 5.22 and 5.24 can be generated using the hardwired approach described in Section 5.6. But, there is an interesting alternative that was popular in the past, which we describe next.

Control signals are generated for each execution step based on the instruction in the IR. In hardwired control, these signals are generated by circuits that interpret the contents of the IR as well as the timing signals derived from a step counter. Instead of employing such circuits, it is possible to use a "software" approach, in which the desired setting of the control signals in each step is determined by a program stored in a special memory. The control program is called a *microprogram* to distinguish it from the program being executed by the processor. The microprogram is stored on the processor chip in a small and fast memory called the *microprogram memory* or the *control store*.

Suppose that  $n$  control signals are needed. Let each control signal be represented by a bit in an  $n$ -bit word, which is often referred to as a *control word* or a *microinstruction*. Each bit in that word specifies the setting of the corresponding signal for a particular step in the execution flow. One control word is stored in the microprogram memory for each step in the execution sequence of an instruction. For example, the action of reading an instruction or a data operand from the memory requires use of the MEM\_read and WMFC signals introduced in Sections 5.5 and 5.6.2, respectively. These signals are asserted by setting the corresponding bits in the control word to 1 for steps 1, 3, and 5 in Figure 5.26. When a microinstruction is read from the control store, each control signal takes on the value of its corresponding bit.

The sequence of microinstructions corresponding to a given machine instruction constitutes the *microroutine* that implements that instruction. The first two steps in Figures 5.25 and 5.26 specify the actions for fetching and decoding an instruction. They are common to all instructions. The microroutine that is specific to a given machine instruction starts with step 3.

Figure 5.27 depicts a typical organization of the hardware needed for microprogrammed control. It consists of a microinstruction address generator, which generates the address



**Figure 5.27** Microprogrammed control unit organization.

to be used for reading microinstructions from the control store. The address generator uses a *microprogram counter*,  $\mu$ PC, to keep track of control store addresses when reading microinstructions from successive locations. During step 2 in Figures 5.25 and 5.26, the microinstruction address generator decodes the instruction in the IR to obtain the starting address of the corresponding microroutine and loads that address into the  $\mu$ PC. This is the address that will be used in the following clock cycle to read the control word corresponding to step 3. As execution proceeds, the microinstruction address generator increments the  $\mu$ PC to read microinstructions from successive locations in the control store. One bit in the microinstruction, which we will call End, is used to mark the last microinstruction in a given microroutine. When End is equal to 1, as would be the case in step 3 in Figure 5.25 and step 7 in Figure 5.26, the address generator returns to the microinstruction corresponding to step 1, which causes a new machine instruction to be fetched.

Microprogrammed control can be viewed as having a control processor within the main processor. Microinstructions are fetched and executed much like machine instructions. Their function is to direct the actions of the main processor's hardware components, by indicating which control signals need to be active during each execution step.

Microprogrammed control is simple to implement and provides considerable flexibility in controlling the execution of machine instructions. But, it is slower than hardwired control. Also, the flexibility it provides is not needed in RISC-style processors. As the discussion in this chapter illustrates, the control signals needed to implement RISC-style instructions are

quite simple to generate. Since the cost of logic circuitry is no longer a significant factor, hardwired control has become the preferred choice.

## 5.8 CONCLUDING REMARKS

This chapter explained the basic structure of a processor and how it executes instructions. Modern processors have a multi-stage organization because this is a structure that is well-suited to pipelined operation. Each stage implements the actions needed in one of the execution steps of an instruction. A five-step sequence in which each step is completed in one clock cycle has been demonstrated. Such an approach is commonly used in processors that have a RISC-style instruction set.

The discussion in this chapter assumed that the execution of one instruction is completed before the next instruction is fetched. Only one of the five hardware stages is used at any given time, as execution moves from one stage to the next in each clock cycle. We will show in the next chapter that it is possible to overlap the execution steps of successive instructions, resulting in much better performance. This leads to a pipelined organization.

## 5.9 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

**Problem:** Figure 5.11 shows an Add instruction being executed in five steps, but no processing actions take place in step 4. If it is desired to eliminate that step, what changes have to be made in the datapath in Figure 5.8 to make this possible?

**Example 5.1**

**Solution:** Step 4 can be skipped by sending the output of the ALU in Figure 5.8 directly to register RY. This can be accomplished by adding one more input to multiplexer Mux Y and connecting that input to the output of the ALU. Thus, the result of a computation at the output of the ALU is loaded into both registers RZ and RY at the end of step 3. For an Add instruction, or any other computational instruction, the register file control signal RF\_write can be enabled in step 4 to load the contents of RY into the register file.

**Problem:** Assume that all memory access operations are completed in one clock cycle in a processor that has a 1-GHz clock. What is the frequency of memory access operations if Load and Store instructions constitute 20 percent of the dynamic instruction count in a program? (The dynamic count is the number of instruction executions, including the effect of program loops, which may cause some instructions to be executed more than once.) Assume that all instructions are executed in 5 clock cycles.

**Example 5.2**

**Solution:** There is one memory access to fetch each instruction. Then, 20 percent of the instructions have a second memory access to read or write a memory operand. On average, each instruction has 1.2 memory accesses in 5 clock cycles. Therefore, the frequency of memory accesses is  $(1.2/5) \times 10^9$ , or 240 million accesses per second.

**Example 5.3**

**Problem:** Derive the logic expressions for a circuit that compares two unsigned numbers:  $X = x_2x_1x_0$  and  $Y = y_2y_1y_0$  and generates three outputs:  $XGY$ ,  $XEQ$ , and  $XLY$ . One of these outputs is set to 1 to indicate that  $X$  is greater than, equal to, or less than  $Y$ , respectively.

**Solution:** To compare two unsigned numbers, we need to compare individual bit locations, starting with the most significant bit. If  $x_2 = 1$  and  $y_2 = 0$ , then  $X$  is greater than  $Y$ . If  $x_2 = y_2$ , then we need to compare the next lower bit location, and so on. Thus, the logic expressions for the three outputs may be written as follows:

$$\begin{aligned} XGY &= x_2\bar{y}_2 + \overline{(x_2 \oplus y_2)} \cdot (x_1\bar{y}_1 + \overline{(x_1 \oplus y_1)} x_0\bar{y}_0) \\ XEQ &= \overline{(x_2 \oplus y_2)} \cdot \overline{(x_1 \oplus y_1)} \cdot \overline{(x_0 \oplus y_0)} \\ XLY &= \overline{XGY + XEQ} \end{aligned}$$

**Example 5.4**

**Problem:** Give the sequence of actions for a Return-from-subroutine instruction in a RISC processor. Assume that the address LINK of the general-purpose register in which the subroutine return address is stored is given in the instruction field connected to address A of the register file ( $IR_{31-27}$ ).

**Solution:** Whenever an instruction is loaded into the IR, the contents of the general-purpose register whose address is given in bits  $IR_{31-27}$  are read and placed into register RA (see Figure 5.18). Hence, a Return-from-subroutine instruction will cause the contents of register LINK to be read and placed in register RA. Execution proceeds as follows:

1. Memory address  $\leftarrow [PC]$ , Read memory, Wait for MFC,  $IR \leftarrow$  Memory data,  $PC \leftarrow [PC] + 4$
2. Decode instruction,  $RA \leftarrow [LINK]$
3.  $PC \leftarrow [RA]$
4. No action
5. No action

**Example 5.5**

**Problem:** A processor has the following interrupt structure. When an interrupt is received, the interrupt return address is saved in a general-purpose register called IRA. The current contents of the processor status register, PS, are saved in a special register called IPS, which is not a general-purpose register. The interrupt-service routine starts at address ILOC.

Assume that the processor checks for interrupts in the last execution step of every instruction. If an interrupt request is present and interrupts are enabled, the request is accepted. Instead of fetching the next instruction, the processor saves the PC and the PS and branches to ILOC. Give a suitable sequence of steps for performing these actions. What additional hardware is needed in Figures 5.18 to 5.20 to support interrupt processing?

**Solution:** The first two steps of instruction execution, in which an instruction is fetched and decoded, are not needed in the case of an interrupt. They may be skipped, or they would take no action if it is desired to maintain a 5-step sequence. Saving the PC can be done in exactly the same manner as for a subroutine call instruction. Another input to MuxC in Figure 5.18 is needed to which the address of register IRA should be connected. To load the starting address of the interrupt-service routine into the PC, an additional input to MuxPC in Figure 5.20 is needed, to which the value ILOC should be connected. Registers PS and IPS should be connected directly to each other to enable data to be transferred between them. The execution steps required are:

3.  $\text{PC-Temp} \leftarrow [\text{PC}]$ ,  $\text{PC} \leftarrow \text{ILOC}$ ,  $\text{IPS} \leftarrow [\text{PS}]$ , Disable interrupts
4.  $\text{RY} \leftarrow [\text{PC-Temp}]$
5.  $\text{IRA} \leftarrow [\text{RY}]$

These actions are reversed by a Return-from-interrupt instruction. See Problem 5.8.

**Problem:** Example 5.5 illustrates how the contents of the PC and the PS are saved when an interrupt request is accepted. In order to support interrupt nesting, it is necessary for the interrupt-service routine to save these registers on the processor stack, as described in Section 3.2. To do so, the contents of the PS, which are saved in register IPS at the time the interrupt is accepted, need to be moved to one of the general-purpose registers, from where they can be saved on the stack. Assume that two special instructions

MoveControl     $R_i, \text{IPS}$

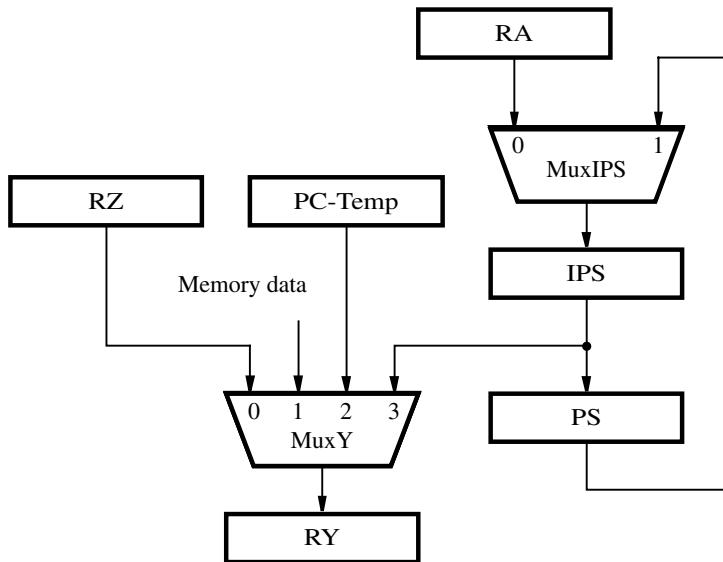
and

MoveControl     $\text{IPS}, R_i$

are available to save and restore the contents of IPS, respectively. Suggest changes to the hardware in Figures 5.8 and 5.10 to implement these instructions.

### Example 5.6

**Solution:** A possible organization is shown in Figure 5.28. To save the contents of IPS, its output is connected to an additional input on MuxY. When restoring its contents, MuxIPS selects register RA.



**Figure 5.28** Connection of IPS for Example 5.6.

## PROBLEMS

- 5.1** [M] The propagation delay through the combinational circuit in Figure 5.2 is 600 ps (picoseconds). The registers have a setup time requirement of 50 ps, and the maximum propagation delay from the clock input to the Q outputs is 70 ps.
- What is the minimum clock period required for correct operation of this circuit?
  - Assume that the circuit is reorganized into three stages as in Figure 5.3, such that the combinational circuit in each stage has a delay of 200 ps. What is the minimum clock period in this case?

- 5.2** [M] At the time the instruction

Load R6, 1000(R9)

is fetched, R6 and R9 contain the values 4200 and 85320, respectively. Memory location 86320 contains 75900. Show the contents of the interstage registers in Figure 5.8 during each of the 5 execution steps of this instruction.

- 5.3** [E] Figure 5.12 shows the bit fields assigned to register addresses for different groups of instructions. Why is it important to use the same field locations for all instructions?
- 5.4** [M] At some point in the execution of a program, registers R4, R6, and R7 contain the values 1000, 7500, and 2500, respectively. Show the contents of registers RA, RB, RZ,

RY, and R6 in Figure 5.8 during steps 3 to 5 as the instruction

Subtract R6, R4, R7

is fetched and executed, and also during step 1 of the instruction that is fetched next.

**5.5** [M] The instruction

And R4, R4, R8

is stored in location 0x37C00 in the memory. At the time this instruction is fetched, registers R4 and R8 contain the values 0x1000 and 0xB2500, respectively. Give the values in registers PC, R4, RA, RM, RZ, and RY of Figures 5.8 and 5.10 in each clock cycle as this instruction is executed, and also in the first clock cycle of the next instruction.

**5.6** [D] Modify the expressions given in Example 5.3 to compare two, 4-bit, signed numbers in 2's-complement representation.

**5.7** [E] The subroutine-call instructions described in Chapter 2 always use the same general-purpose register, LINK, to store the return address. Hence, the return register address is not included in the instruction. However, the address LINK is included in bits IR<sub>31–27</sub> of subroutine-return instructions (see Section 5.4.1 and Example 5.4). Why are the two instructions treated differently?

**5.8** [M] Give the execution sequence for the Return-from-interrupt instruction for a processor that has the interrupt structure given in Example 5.5. Assume that the address of register IRA is given in bits IR<sub>31–27</sub> of the instruction.

**5.9** [D] Consider an instruction set in which instruction encoding is such that register addresses for different instructions are not always in the same bit locations. What effect would that have on the execution steps of the instructions? What would you do to maintain a five-step execution sequence in this case? Assume the same hardware structure as in Figure 5.8.

**5.10** [M] Assume that immediate operands occupy bits IR<sub>21–6</sub> of the instruction. The immediate value is sign-extended to 32 bits in arithmetic instructions, such as Add, and padded with zeros in logic instructions, such as Or. Design a suitable implementation for the Immediate block in Figure 5.9.

**5.11** [M] A RISC processor that uses the five-step sequence in Figure 5.4 is driven by a 1-GHz clock. Instruction statistics in a large program are as follows:

Branch	20%
Load	20%
Store	10%
Computational instructions	50%

Estimate the rate of instruction execution in each of the following cases:

(a) Access to the memory is always completed in 1 clock cycle.

(b) 90% of instruction fetch operations are completed in one clock cycle and 10% are completed in 4 clock cycles. On average, access to the data operands of a Load or Store instruction is completed in 3 clock cycles.

**5.12** [E] The execution of computational instructions follows the pattern given in Figure 5.11 for the Add instruction, in which no processing actions are performed in step 4. Consider a program that has the instruction statistics given in Problem 5.11. Estimate the increase in instruction execution rate if this step is eliminated, assuming that all execution steps are completed in one clock cycle.

**5.13** [D] Figure 5.16 shows that step 3 of a conditional branch instruction may result in a new value being loaded into the PC. In pipelined processors, it is desirable to determine the outcome of a conditional branch as early as possible in the execution sequence. What hardware changes would be needed to make it possible to move the actions in step 3 to step 2? Examine all the actions involved in these two steps and show which actions can be carried out in parallel and which must be completed sequentially.

**5.14** [M] The instructions of a computer are encoded as shown in Figure 5.12. When an immediate value is given in an instruction, it has to be extended to a 32-bit value. Assume that the immediate value is used in three different ways:

- (a) A 16-bit value is sign-extended for use in arithmetic operations.
- (b) A 16-bit value is padded with zeros to the left for use in logic operations.
- (c) A 26-bit value is padded with 2 zeros to the right and the 4 high-order bits of the PC are appended to the left for use in subroutine-call instructions.

Show an implementation for the Immediate block in Figure 5.19 that would perform the required extensions.

**5.15** [E] We have seen how all RISC-style instructions can be executed using the steps in Figure 5.4 on the multi-stage hardware of Figure 5.8. Autoincrement and Autodecrement addressing modes are not included in RISC-style instruction sets. Explain why the instruction

Load R3, (R5)+

cannot be executed on the hardware in Figure 5.8.

**5.16** [E] Section 2.9 describes how the two instructions Or and OrHigh can be used to load a 32-bit value into a register. What additional functionality is needed in the processor's datapath to implement the OrHigh instruction? Give the sequence of actions needed to fetch and execute the instruction.

**5.17** [E] During step 1 of instruction processing, a memory Read operation is started to fetch an instruction at location 0x46000. However, as the instruction is not found in the cache, the Read operation is delayed, and the MFC signal does not become active until the fourth clock cycle. Assume that the delay is handled as described in Section 5.6.2. Show the contents of the PC during each of the four clock cycles of step 1, and also during step 2.

**5.18** [M] Give the sequence of steps needed to fetch and execute the two special instructions

MoveControl Ri, IPS

and

MoveControl IPS, Ri

used in Example 5.6.

- 5.19** [D] What are the essential differences between the hardware structures in Figures 5.8 and 5.22? Illustrate your answer by identifying the difficulties that would be encountered if one attempts to execute the instruction

Subtract LOC, R5

on the hardware in Figure 5.8. This instruction performs the operation

$$\text{LOC} \leftarrow [\text{LOC}] - [\text{R5}]$$

where LOC is a memory location whose address is given as the second word of a two-word instruction.

- 5.20** [M] Consider the actions needed to execute the instructions given in Section 5.4.1. Derive the logic expressions to generate the signals C\_select, MA\_select, and Y\_select in Figures 5.18 and 5.19 for these instructions.

- 5.21** [E] Why is it necessary to include both WMFC and MFC in the logic expression for Counter\_enable given in Section 5.6.2?

- 5.22** [E] Explain what would happen if the MFC variable is omitted from the expression for PC\_enable given in Section 5.6.2.

- 5.23** [M] Derive the logic expressions to generate the signals PC\_select and INC\_select shown in Figure 5.20, taking into account the actions needed when executing the following instructions:

Branch: All branch instructions, with a 16-bit branch offset given in the instruction

Call\_register: A subroutine-call instruction with the subroutine address given in a general-purpose register

Other: All other instructions that do not involve branching

- 5.24** [M] A microprogrammed processor has the following parameters. Generating the starting address of the microroutine of an instruction takes 2.1 ns, and reading a microinstruction from the control store takes 1.5 ns. Performing an operation in the ALU requires a maximum of 2.2 ns, and access to the cache memory requires 1.7 ns. Assume that all instructions and data are in the cache.

(a) Determine the minimum time needed for each of the steps in Figure 5.26.

(b) Ignoring all other delays, what is the minimum clock cycle that can be used for this processor?

- 5.25** [M] Give the sequence of steps needed to fetch and execute the instruction

Load R3, (R5)+

on the processor of Figure 5.24. Assume 32-bit operands.

- 5.26** [M] Consider a CISC-style processor that saves the return address of a subroutine on the processor stack instead of in the predefined register LINK. Give the sequence of actions needed to execute a Call\_Register instruction on the processor of Figure 5.24.

*This page intentionally left blank*



**PART SIX** THE CONTROL  
UNIT

**CHAPTER** **20**

# CONTROL UNIT OPERATION

## **20.1 Micro-Operations**

- The Fetch Cycle
- The Indirect Cycle
- The Interrupt Cycle
- The Execute Cycle
- The Instruction Cycle

## **20.2 Control of the Processor**

- Functional Requirements
- Control Signals
- A Control Signals Example
- Internal Processor Organization
- The Intel 8085

## **20.3 Hardwired Implementation**

- Control Unit Inputs
- Control Unit Logic

## **20.4 Key Terms, Review Questions, and Problems**

### LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Explain the concept of micro-operations and define the principal instruction cycle phases in terms of micro-operations.
- ◆ Discuss how micro-operations are organized to control a processor.
- ◆ Understand hardwired control unit organization.

In Chapter 12, we pointed out that a machine instruction set goes a long way toward defining the processor. If we know the machine instruction set, including an understanding of the effect of each opcode and an understanding of the addressing modes, and if we know the set of user-visible registers, then we know the functions that the processor must perform. This is not the complete picture. We must know the external interfaces, usually through a bus, and how interrupts are handled. With this line of reasoning, the following list of those things needed to specify the function of a processor emerges:

1. Operations (opcodes)
2. Addressing modes
3. Registers
4. I/O module interface
5. Memory module interface
6. Interrupts

This list, though general, is rather complete. Items 1 through 3 are defined by the instruction set. Items 4 and 5 are typically defined by specifying the system bus. Item 6 is defined partially by the system bus and partially by the type of support the processor offers to the operating system.

This list of six items might be termed the functional requirements for a processor. They determine what a processor must do. This is what occupied us in Parts Two and Four. Now, we turn to the question of how these functions are performed or, more specifically, how the various elements of the processor are controlled to provide these functions. Thus, we turn to a discussion of the control unit, which controls the operation of the processor.

## 20.1 MICRO-OPERATIONS

We have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. Of course, we must remember that this sequence of instruction cycles is not necessarily the same as the *written sequence* of instructions that make up the program, because of the existence of branching instructions. What we are referring to here is the execution *time sequence* of instructions.

We have further seen that each instruction cycle is made up of a number of smaller units. One subdivision that we found convenient is fetch, indirect, execute, and interrupt, with only fetch and execute cycles always occurring.

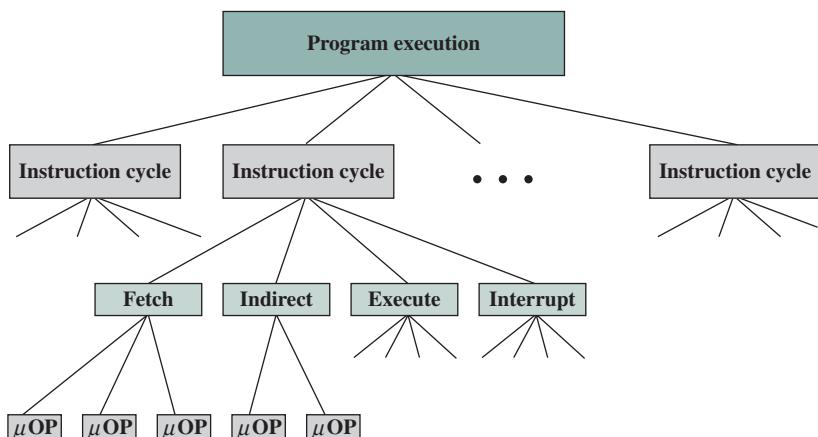
To design a control unit, however, we need to break down the description further. In our discussion of pipelining in Chapter 14, we began to see that a further decomposition is possible. In fact, we will see that each of the smaller cycles involves a series of steps, each of which involves the processor registers. We will refer to these steps as **micro-operations**. The prefix *micro* refers to the fact that each step is very simple and accomplishes very little. Figure 20.1 depicts the relationship among the various concepts we have been discussing. To summarize, the execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (e.g., fetch, indirect, execute, interrupt). The execution of each subcycle involves one or more shorter operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations. A simple example will be used. In the remainder of this chapter, we then show how the concept of micro-operations serves as a guide to the design of the control unit.

## The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. For purposes of discussion, we assume the organization depicted in Figure 14.6 (*Data Flow, Fetch Cycle*). Four registers are involved:

- **Memory address register (MAR):** Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory buffer register (MBR):** Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.



**Figure 20.1** Constituent Elements of a Program Execution

- **Program counter (PC):** Holds the address of the next instruction to be fetched.
- **Instruction register (IR):** Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 20.2. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100. The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by the instruction length to get ready for the next instruction. Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

$$\begin{aligned} t_1: \text{MAR} &\leftarrow (\text{PC}) \\ t_2: \text{MBR} &\leftarrow \text{Memory} \\ \text{PC} &\leftarrow (\text{PC}) + I \\ t_3: \text{IR} &\leftarrow (\text{MBR}) \end{aligned}$$

where  $I$  is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are

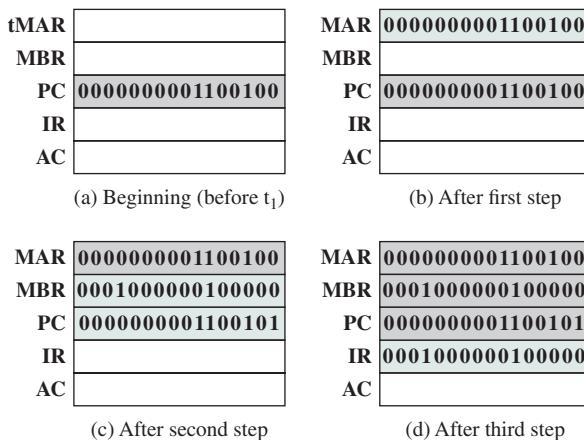


Figure 20.2 Sequence of Events, Fetch Cycle

of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation  $(t_1, t_2, t_3)$  represents successive time units. In words, we have

- **First time unit:** Move contents of PC to MAR.
- **Second time unit:** Move contents of memory location specified by MAR to MBR. Increment by  $I$  the contents of the PC.
- **Third time unit:** Move contents of MBR to IR.

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

$$\begin{aligned} t_1: \text{MAR} &\leftarrow (\text{PC}) \\ t_2: \text{MBR} &\leftarrow \text{Memory} \\ t_3: \text{PC} &\leftarrow (\text{PC}) + I \\ \text{IR} &\leftarrow (\text{MBR}) \end{aligned}$$

The groupings of micro-operations must follow two simple rules:

1. The proper sequence of events must be followed. Thus  $(\text{MAR} \leftarrow (\text{PC}))$  must precede  $(\text{MBR} \leftarrow \text{Memory})$  because the memory read operation makes use of the address in the MAR.
2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations  $(\text{MBR} \leftarrow \text{Memory})$  and  $(\text{IR} \leftarrow \text{MBR})$  should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor. We defer a discussion of this point until later in this chapter.

It is useful to compare events described in this and the following subsections to Figure 3.5 (*Example of Program Execution*). Whereas micro-operations are ignored in that figure, this discussion shows the micro-operations needed to perform the subcycles of the instruction cycle.

### The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. The data flow differs somewhat from that indicated in Figure 14.7 (*Data Flow, Indirect Cycle*) and includes the following micro-operations:

$$\begin{aligned} t_1: \text{MAR} &\leftarrow (\text{IR}(\text{Address})) \\ t_2: \text{MBR} &\leftarrow \text{Memory} \\ t_3: \text{IR}(\text{Address}) &\leftarrow (\text{MBR}(\text{Address})) \end{aligned}$$

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

### The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, as illustrated in Figure 14.8 (*Data Flow, Indirect Cycle*). We have

$t_1:$	MBR	$\leftarrow$	(PC)
$t_2:$	MAR	$\leftarrow$	Save_Address
	PC	$\leftarrow$	Routine_Address
$t_3:$	Memory	$\leftarrow$	(MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the Save\_Address and the Routine\_Address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

### The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. Because of the variety of opcodes, there are a number of different sequences of micro-operations that can occur. The control unit examines the opcode and generates a sequence of micro-operations based on the value of the opcode. This is referred to as instruction decoding.

Let us consider several hypothetical examples.

First, consider an add instruction:

ADD R1, X

which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

$t_1:$	MAR	$\leftarrow$	(IR (address))
$t_2:$	MBR	$\leftarrow$	Memory
$t_3:$	R1	$\leftarrow$	(R1) + (MBR)

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract

the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skip if zero:

ISZ X

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

```

 $t_1: MAR \leftarrow (IR \text{ (address)})$ 
 $t_2: MBR \leftarrow \text{Memory}$ 
 $t_3: MBR \leftarrow (MBR) + 1$ 
 $t_4: \text{Memory} \leftarrow (MBR)$ 
    If  $((MBR) = 0)$  then  $(PC \leftarrow (PC) + I)$ 
```

The new feature introduced here is the conditional action. The PC is incremented if  $(MBR) = 0$ . This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

BSA X

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location  $X + I$ . The saved address will later be used for return. This is a straightforward technique for supporting subroutine calls. The following micro-operations suffice:

```

 $t_1: MAR \leftarrow (IR \text{ (address)})$ 
        MBR  $\leftarrow (PC)$ 
 $t_2: PC \leftarrow (IR \text{ (address)})$ 
        Memory  $\leftarrow (MBR)$ 
 $t_3: PC \leftarrow (PC) + I$ 
```

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

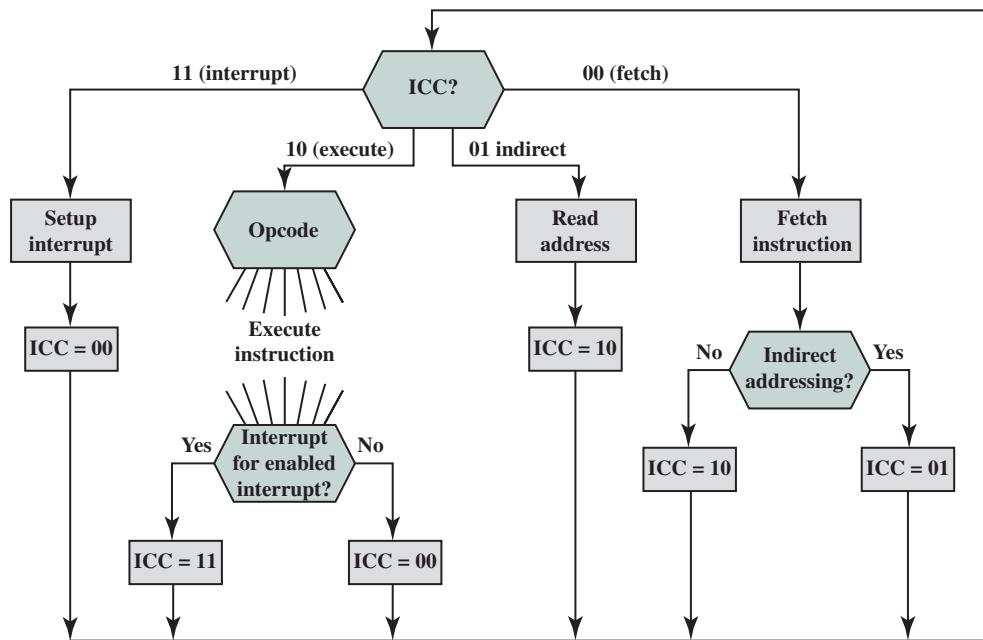
### The Instruction Cycle

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence each for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode.

To complete the picture, we need to tie sequences of micro-operations together, and this is done in Figure 20.3. We assume a new 2-bit register called the *instruction cycle code* (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch

01: Indirect



**Figure 20.3** Flowchart for Instruction Cycle

10: Execute

11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle (see Figure 14.4, *The Instruction Cycle*). For both the fetch and execute cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 20.3 defines the complete sequence of micro-operations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in which the operation of the processor is defined as the performance of a sequence of micro-operations. We can now consider how the control unit causes this sequence to occur.

## 20.2 CONTROL OF THE PROCESSOR

### Functional Requirements

As a result of our analysis in the preceding section, we have decomposed the behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the *functional requirements* for the control unit: those functions that the control unit must perform. A definition of these functional requirements is the basis for the design and implementation of the control unit.

With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

We have already performed steps 1 and 2. Let us summarize the results. First, the basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths
- External data paths
- Control unit

Some thought should convince you that this is a complete list. The ALU is the functional essence of the computer. Registers are used to store data internal to the processor. Some registers contain status information needed to manage instruction sequencing (e.g., a program status word). Others contain data that go to or come from the ALU, memory, and I/O modules. Internal data paths are used to move data between registers and between register and ALU. External data paths link registers to memory and I/O modules, often by means of a system bus. The control unit causes operations to happen within the processor.

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. Upon review of Section 20.1, the reader should see that all micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface to a register.
- Perform an arithmetic or logic operation, using registers for input and output.

All of the micro-operations needed to perform one instruction cycle, including all of the micro-operations to execute every instruction in the instruction set, fall into one of these categories.

We can now be somewhat more explicit about the way in which the control unit functions. The control unit performs two basic tasks:

- **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
- **Execution:** The control unit causes each micro-operation to be performed.

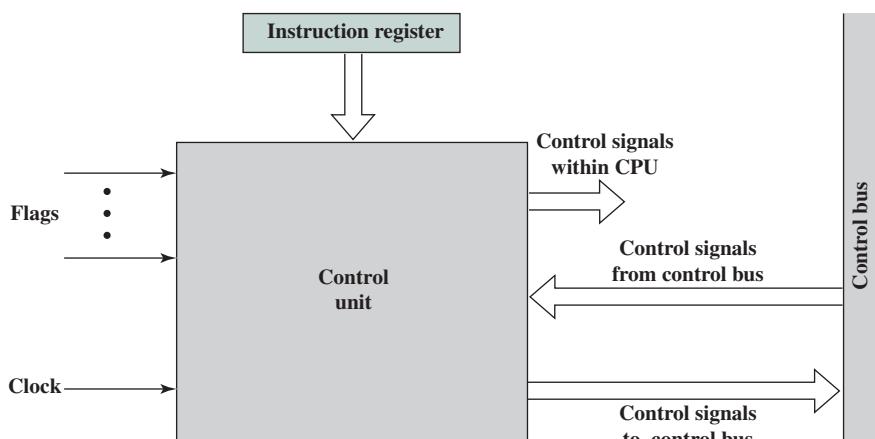
The preceding is a functional description of what the control unit does. The key to how the control unit operates is the use of control signals.

## Control Signals

We have defined the elements that make up the processor (ALU, registers, data paths) and the micro-operations that are performed. For the control unit to perform its function, it must have inputs that allow it to determine the state of the system and outputs that allow it to control the behavior of the system. These are the external specifications of the control unit. Internally, the control unit must have the logic required to perform its sequencing and execution functions. We defer a discussion of the internal operation of the control unit to Section 20.3 and Chapter 21. The remainder of this section is concerned with the interaction between the control unit and the other elements of the processor.

Figure 20.4 is a general model of the control unit, showing all of its inputs and outputs. The inputs are:

- **Clock:** This is how the control unit “keeps time.” The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time.
- **Instruction register:** The opcode and addressing mode of the current instruction are used to determine which micro-operations to perform during the execute cycle.
- **Flags:** These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.
- **Control signals from control bus:** The control bus portion of the system bus provides signals to the control unit.



**Figure 20.4** Block Diagram of the Control Unit

The outputs are as follows:

- **Control signals within the processor:** These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
- **Control signals to control bus:** These are also of two types: control signals to memory, and control signals to the I/O modules.

Three types of control signals are used: those that activate an ALU function; those that activate a data path; and those that are signals on the external system bus or other external interface. All of these signals are ultimately applied directly as binary inputs to individual logic gates.

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

- A control signal that opens gates, allowing the contents of the MAR onto the address bus;
- A memory read control signal on the control bus;
- A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR;
- Control signals to logic that add 1 to the contents of the PC and store the result back to the PC.

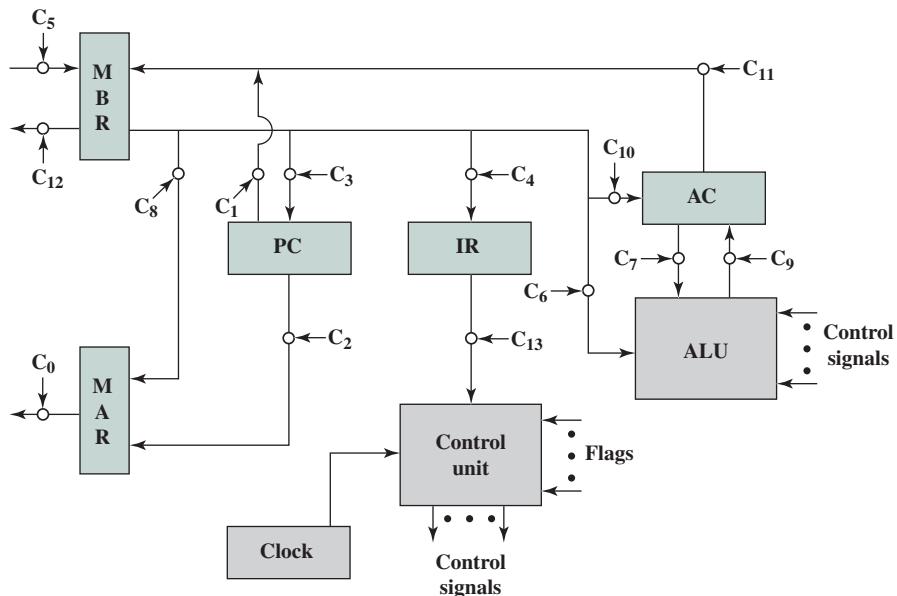
Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

This completes the fetch cycle except for one thing: The control unit must decide whether to perform an indirect cycle or an execute cycle next. To decide this, it examines the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit begins by examining the opcode and, on the basis of that, decides which sequence of micro-operations to perform for the execute cycle.

## A Control Signals Example

To illustrate the functioning of the control unit, let us examine a simple example. Figure 20.5 illustrates the example. This is a simple processor with a single accumulator (AC). The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled  $C_i$  and indicated by a circle. The control unit receives inputs from the clock, the IR, and flags. With each clock cycle, the control unit



**Figure 20.5** Data Paths and Control Signals

reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

- **Data paths:** The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the IR. For each path to be controlled, there is a switch (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.
- **ALU:** The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic circuits and gates within the ALU.
- **System bus:** The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize. Table 20.1 indicates the control signals that are needed for some of the micro-operation sequences described earlier. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical operations (e.g., positive, overflow, etc.). It never gets to see the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

**Table 20.1** Micro-operations and Control Signals

	Micro-operations	Active Control Signals
Fetch:	$t_1: MAR \leftarrow (PC)$	$C_2$
	$t_2: MBR \leftarrow \text{Memory}$ $PC \leftarrow (PC) + 1$	$C_5, C_R$
	$t_3: IR \leftarrow (MBR)$	$C_4$
Indirect:	$t_1: MAR \leftarrow (IR(\text{Address}))$	$C_8$
	$t_2: MBR \leftarrow \text{Memory}$	$C_5, C_R$
	$t_3: IR(\text{Address}) \leftarrow (MBR(\text{Address}))$	$C_4$
Interrupt:	$t_1: MBR \leftarrow (PC)$	$C_1$
	$t_2: MAR \leftarrow \text{Save-address}$ $PC \leftarrow \text{Routine-address}$	
	$t_3: \text{Memory} \leftarrow (MBR)$	$C_{12}, C_W$

$C_R$  = Read control signal to system bus.

$C_W$  = Write control signal to system bus.

### Internal Processor Organization

Figure 20.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear. More typically, some sort of internal bus arrangement, as was suggested in Figure 14.2 (*Internal Structure of the CPU*), will be used.

Using an internal processor bus, Figure 20.5 can be rearranged as shown in Figure 20.6. A single internal bus connects the ALU and all processor registers. Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z, have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit (see Chapter 11) with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Therefore, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage. With this arrangement, an operation to add a value from memory to the AC would have the following steps:

- $t_1: MAR \leftarrow (IR(\text{address}))$
- $t_2: MBR \leftarrow \text{Memory}$
- $t_3: Y \leftarrow (MBR)$
- $t_4: Z \leftarrow (AC) + (Y)$
- $t_5: AC \leftarrow (Z)$

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the

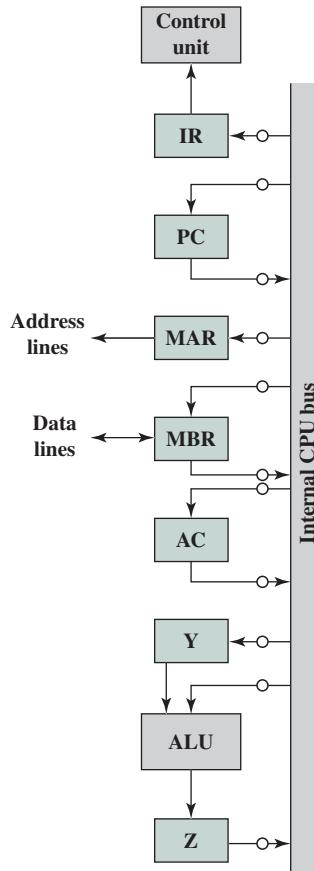


Figure 20.6 CPU with Internal Bus

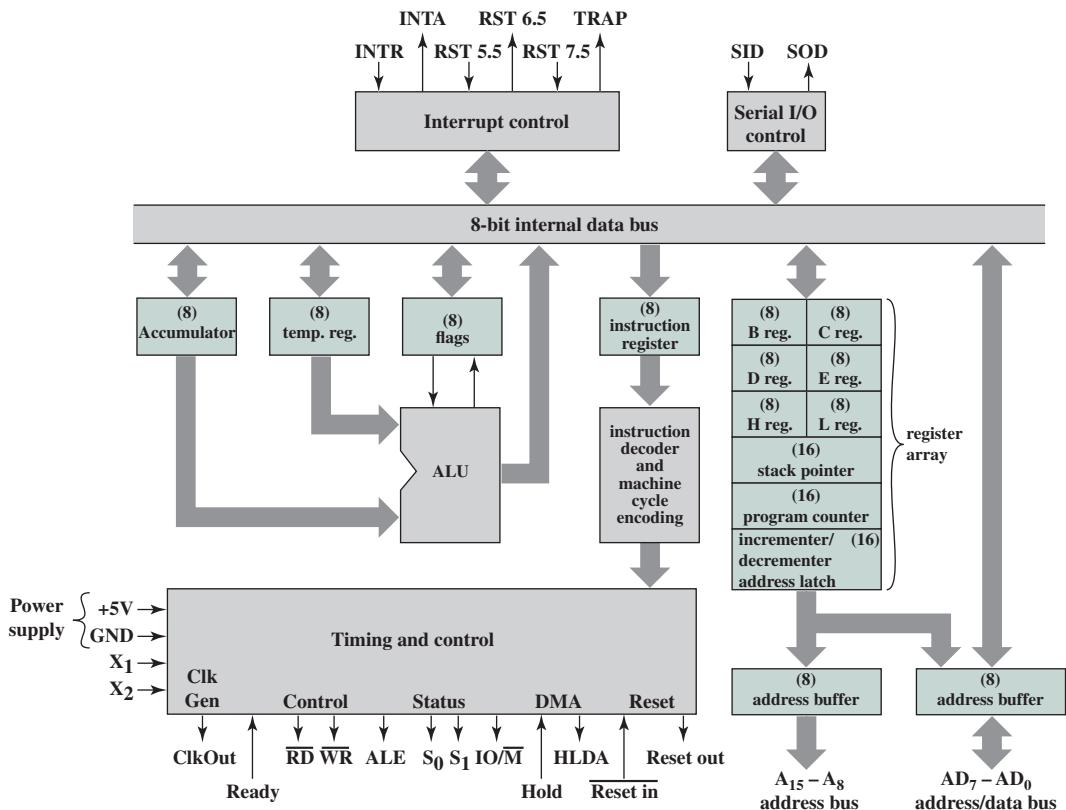
interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space.

### The Intel 8085

To illustrate some of the concepts introduced thus far in this chapter, let us consider the Intel 8085. Its organization is shown in Figure 20.7. Several key components that may not be self-explanatory are:

- **Incrementer/decrementer address latch:** Logic that can add 1 to or subtract 1 from the contents of the stack pointer or program counter. This saves time by avoiding the use of the ALU for this purpose.
- **Interrupt control:** This module handles multiple levels of interrupt signals.
- **Serial I/O control:** This module interfaces to devices that communicate 1 bit at a time.

Table 20.2 describes the external signals into and out of the 8085. These are linked to the external system bus. These signals are the interface between the 8085 processor and the rest of the system (Figure 20.8).



**Figure 20.7** Intel 8085 CPU Block Diagram

The control unit is identified as having two components labeled (1) instruction decoder and machine cycle encoding and (2) timing and control. A discussion of the first component is deferred until the next section. The essence of the control unit is the timing and control module. This module includes a clock and accepts as inputs the current instruction and some external control signals. Its output consists of control signals to the other components of the processor plus control signals to the external system bus.

The timing of processor operations is synchronized by the clock and controlled by the control unit with control signals. Each instruction cycle is divided into from one to five *machine cycles*; each machine cycle is in turn divided into from three to five *states*. Each state lasts one clock cycle. During a state, the processor performs one or a set of simultaneous micro-operations as determined by the control signals.

The number of machine cycles is fixed for a given instruction but varies from one instruction to another. Machine cycles are defined to be equivalent to bus accesses. Thus, the number of machine cycles for an instruction depends on the number of times the processor must communicate with external devices. For example, if an instruction consists of two 8-bit portions, then two machine cycles are required to fetch the instruction. If that instruction involves a 1-byte memory or I/O operation, then a third machine cycle is required for execution.

**Table 20.2** Intel 8085 External Signals

<i>Address and Data Signals</i>	
<b>High Address (A15–A8)</b>	The high-order 8 bits of a 16-bit address.
<b>Address/Data (AD7–AD0)</b>	The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.
<b>Serial Input Data (SID)</b>	A single-bit input to accommodate devices that transmit serially (one bit at a time).
<b>Serial Output Data (SOD)</b>	A single-bit output to accommodate devices that receive serially.
<i>Timing and Control Signals</i>	
<b>CLK (OUT)</b>	The system clock. The CLK signal goes to peripheral chips and synchronizes their timing.
<b>X1, X2</b>	These signals come from an external crystal or other device to drive the internal clock generator.
<b>Address Latch Enabled (ALE)</b>	Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.
<b>Status (S0, S1)</b>	Control signals used to indicate whether a read or write operation is taking place.
<b>IO/M</b>	Used to enable either I/O or memory modules for read and write operations.
<b>Read Control (RD)</b>	Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.
<b>Write Control (WR)</b>	Indicates that data on the data bus is to be written into the selected memory or I/O location.
<i>Memory and I/O Initiated Symbols</i>	
<b>Hold</b>	Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.
<b>Hold Acknowledge (HOLD A)</b>	This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.
<b>READY</b>	Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.
<i>Interrupt-Related Signals</i>	
<b>TRAP</b>	Restart Interrupts (RST 7.5, 6.5, 5.5)
<b>Interrupt Request (INTR)</b>	These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.
<b>Interrupt Acknowledge</b>	Acknowledges an interrupt.

*CPU Initialization***RESET IN**

Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.

**RESET OUT**

Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.

*Voltage and Ground***VCC**

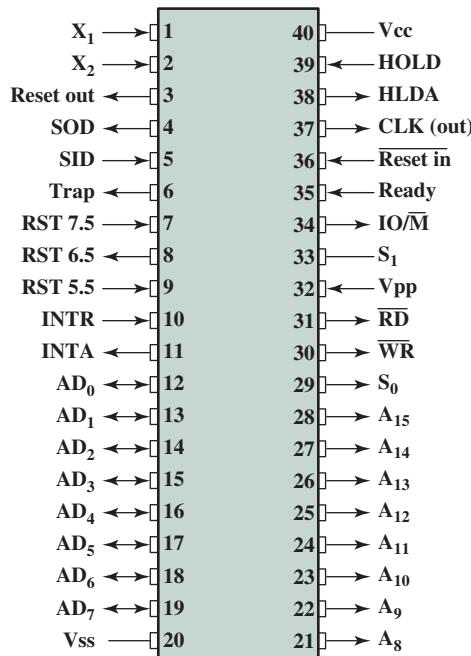
+5-volt power supply

**VSS**

Electrical ground

Figure 20.9 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles ( $M_1, M_2, M_3$ ) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the I/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

The Address Latch Enabled (ALE) pulse signals the start of each machine cycle from the control unit. The ALE pulse alerts external circuits. During timing state  $T_1$  of machine cycle  $M_1$ , the control unit sets the IO/M signal to indicate that this is a memory operation. Also, the control unit causes the contents of the PC



**Figure 20.8** Intel 8085 Pin Configuration

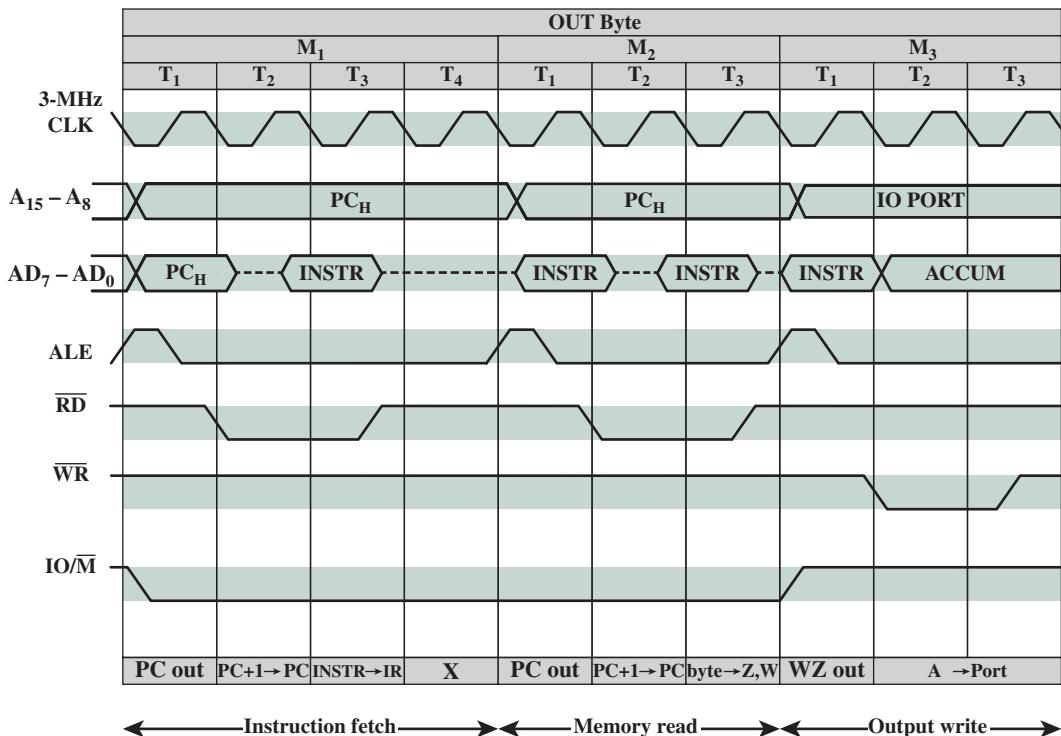


Figure 20.9 Timing Diagram for Intel 8085 OUT Instruction

to be placed on the address bus (A<sub>15</sub> through A<sub>8</sub>) and the address/data bus (AD<sub>7</sub> through AD<sub>0</sub>). With the falling edge of the ALE pulse, the other modules on the bus store the address.

During timing state T<sub>2</sub>, the addressed memory module places the contents of the addressed memory location on the address/data bus. The control unit sets the Read Control (RD) signal to indicate a read, but it waits until T<sub>3</sub> to copy the data from the bus. This gives the memory module time to put the data on the bus and for the signal levels to stabilize. The final state, T<sub>4</sub>, is a *bus idle* state during which the processor decodes the instruction. The remaining machine cycles proceed in a similar fashion.

### 20.3 HARDWIRED IMPLEMENTATION

We have discussed the control unit in terms of its inputs, output, and functions. We now turn to the topic of control unit implementation. A wide variety of techniques have been used. Most of these fall into one of two categories:

- Hardwired implementation
- Microprogrammed implementation

In a **hardwired implementation**, the control unit is essentially a state machine circuit. Its input logic signals are transformed into a set of output logic signals, which

are the control signals. This approach is examined in this section. Microprogrammed implementation is the subject of Chapter 21.

## Control Unit Inputs

Figure 20.4 depicts the control unit as we have so far discussed it. The key inputs are the IR, the clock, flags, and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (e.g., overflow). The other two inputs, however, are not directly useful to the control unit.

First consider the IR. The control unit makes use of the opcode and will perform different actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a *decoder*, which takes an encoded input and produces a single output. In general, a decoder will have  $n$  binary inputs and  $2^n$  binary outputs. Each of the  $2^n$  different input patterns will activate a single unique output. Table 20.3 is an example for  $n = 4$ . The decoder for a control unit will typically have to be more complex than that, to account for variable-length opcodes. An example of the digital logic used to implement a decoder is presented in Chapter 11.

The clock portion of the control unit issues a repetitive sequence of pulses. This is useful for measuring the duration of micro-operations. Essentially, the period of the clock pulses must be long enough to allow the propagation of signals along

**Table 20.3** A Decoder with 4 Inputs and 16 Outputs

data paths and through processor circuitry. However, as we have seen, the control unit emits different control signals at different time units within a single instruction cycle. Thus, we would like a counter as input to the control unit, with a different control signal being used for  $T_1, T_2$ , and so forth. At the end of an instruction cycle, the control unit must feed back to the counter to reinitialize it at  $T_1$ .

With these two refinements, the control unit can be depicted as in Figure 20.10.

### Control Unit Logic

To define the hardwired implementation of a control unit, all that remains is to discuss the internal logic of the control unit that produces output control signals as a function of its input signals.

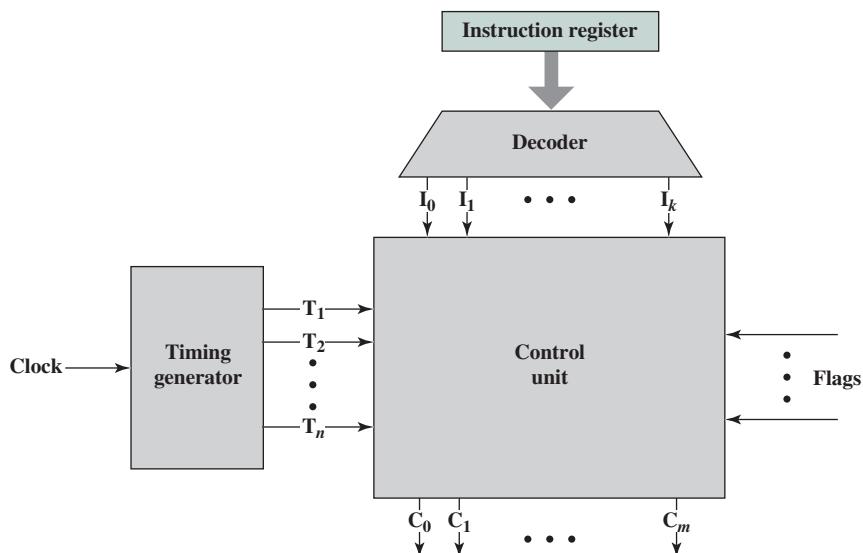
Essentially, what must be done is, for each control signal, to derive a Boolean expression of that signal as a function of the inputs. This is best explained by example. Let us consider again our simple example illustrated in Figure 20.5. We saw in Table 20.1 the micro-operation sequences and control signals needed to control three of the four phases of the instruction cycle.

Let us consider a single control signal,  $C_5$ . This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 20.1. Let us define two new control signals, P and Q, that have the following interpretation:

$PQ = 00$	Fetch Cycle
$PQ = 01$	Indirect Cycle
$PQ = 10$	Execute Cycle
$PQ = 11$	Interrupt Cycle

Then the following Boolean expression defines  $C_5$ :

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2$$



**Figure 20.10** Control Unit with Decoded Inputs

That is, the control signal  $C_5$  will be asserted during the second time unit of both the fetch and indirect cycles.

This expression is not complete.  $C_5$  is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD, and AND. Now we can define  $C_5$  as

$$C_5 = \bar{P} \cdot \bar{Q} \cdot T_2 + \bar{P} \cdot Q \cdot T_2 + P \cdot \bar{Q} \cdot (\text{LDA} + \text{ADD} + \text{AND}) \cdot T_2$$

This same process could be repeated for every control signal generated by the processor. The result would be a set of Boolean equations that define the behavior of the control unit and hence of the processor.

To tie everything together, the control unit must control the state of the instruction cycle. As was mentioned, at the end of each subcycle (fetch, indirect, execute, interrupt), the control unit issues a signal that causes the timing generator to reinitialize and issue  $T_1$ . The control unit must also set the appropriate values of  $P$  and  $Q$  to define the next subcycle to be performed.

The reader should be able to appreciate that in a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a combinatorial circuit that satisfies all of these equations becomes extremely difficult. The result is that a far simpler approach, known as *microprogramming*, is usually used. This is the subject of the next chapter.

## 20.4 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

### Key Terms

control bus control path	control signal control unit	hardwired implementation micro-operations
-----------------------------	--------------------------------	--

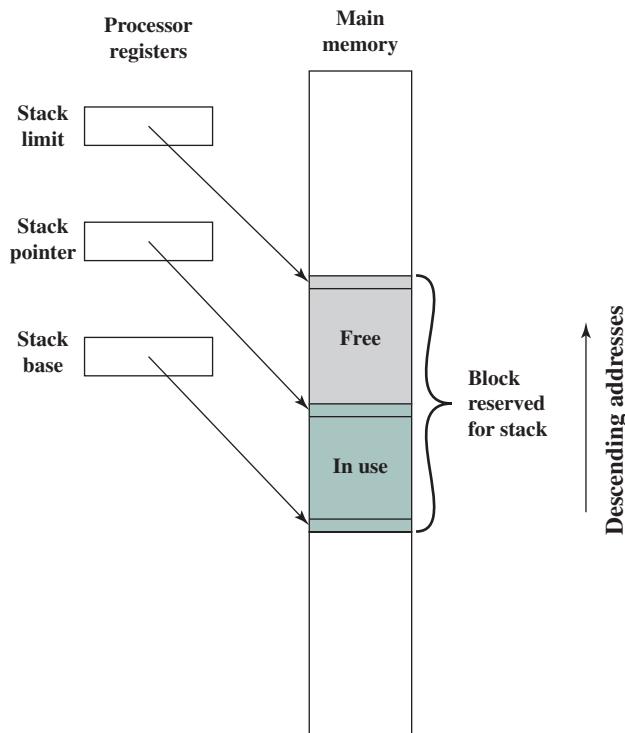
### Review Questions

- 20.1** Explain the distinction between the written sequence and the time sequence of an instruction.
- 20.2** What is the relationship between instructions and micro-operations?
- 20.3** What is the overall function of a processor's control unit?
- 20.4** Outline a three-step process that leads to a characterization of the control unit.
- 20.5** What basic tasks does a control unit perform?
- 20.6** Provide a typical list of the inputs and outputs of a control unit.
- 20.7** List three types of control signals.
- 20.8** Briefly explain what is meant by a hardwired implementation of a control unit.

### Problems

- 20.1** Your ALU can add its two input registers, and it can logically complement the bits of either input register, but it cannot subtract. Numbers are to be stored in twos complement representation. List the micro-operations your control unit must perform to cause a subtraction.

- 20.2** Show the micro-operations and control signals in the same fashion as Table 20.1 for the processor in Figure 20.5 for the following instructions:
- Load Accumulator
  - Store Accumulator
  - Add to Accumulator
  - AND to Accumulator
  - Jump
  - Jump if AC = 0
  - Complement Accumulator
- 20.3** Assume that propagation delay along the bus and through the ALU of Figure 20.6 are 20 and 100 ns, respectively. The time required for a register to copy data from the bus is 10 ns. What is the time that must be allowed for
- data from one register to another?
  - the program counter?
- 20.4** Write the sequence of micro-operations required for the bus structure of Figure 20.6 to add a number to the AC when the number is
- immediate operand;
  - direct-address operand;
  - indirect-address operand.
- 20.5** A stack is implemented as shown in Figure 20.11 (see Appendix I for a discussion of stacks). Show the sequence of micro-operations for
- popping;
  - the stack.



**Figure 20.11** Typical Stack Organization (full/descending)



# CHAPTER **21**

## **MICROPROGRAMMED CONTROL**

### **21.1 Basic Concepts**

- Microinstructions
- Microprogrammed Control Unit
- Wilkes Control
- Advantages and Disadvantages

### **21.2 Microinstruction Sequencing**

- Design Considerations
- Sequencing Techniques
- Address Generation
- LSI-11 Microinstruction Sequencing

### **21.3 Microinstruction Execution**

- A Taxonomy of Microinstructions
- Microinstruction Encoding
- LSI-11 Microinstruction Execution
- IBM 3033 Microinstruction Execution

### **21.4 TI 8800**

- Microinstruction Format
- Microsequencer
- Registered ALU

### **21.5 Key Terms, Review Questions, and Problems**

### LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Present an overview of the basic concepts of microprogrammed control.
- ◆ Understand the difference between hardwired control and microprogrammed control.
- ◆ Discuss the basic categories of sequencing techniques.
- ◆ Present an overview of the taxonomy of microinstructions.

The term *microprogram* was first coined by M. V. Wilkes in the early 1950s [WILK51]. Wilkes proposed an approach to control unit design that was organized and systematic and avoided the complexities of a hardwired implementation. The idea intrigued many researchers but appeared unworkable because it would require a fast, relatively inexpensive control memory.

The state of the microprogramming art was reviewed by *Datamation* in its February 1964 issue. No microprogrammed system was in wide use at that time, and one of the papers [HILL64] summarized the then-popular view that the future of microprogramming “is somewhat cloudy. None of the major manufacturers has evidenced interest in the technique, although presumably all have examined it.”

This situation changed dramatically within a very few months. IBM’s System/360 was announced in April, and all but the largest models were microprogrammed. Although the 360 series predated the availability of semiconductor ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Microprogramming became a popular technique for implementing the control unit of CISC processors. In recent years, microprogramming has become less used but remains a tool available to computer designers. For example, as we have seen on the Pentium 4, machine instructions are converted into a RISC-like format, most of which are executed without the use of microprogramming. However, some of the instructions are executed using microprogramming.

## 21.1 BASIC CONCEPTS

### Microinstructions

The control unit seems a reasonably simple device. Nevertheless, to implement a control unit as an interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operations, for executing micro-operations, for interpreting opcodes, and for making decisions based on ALU flags. It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.

An alternative, which has been used in many CISC processors, is to implement a **microprogrammed control unit**.

Consider Table 21.1. In addition to the use of control signals, each micro-operation is described in symbolic notation. This notation looks suspiciously like a programming language. In fact it is a language, known as a **micropogramming language**. Each line describes a set of micro-operations occurring at one time and is known as a **microinstruction**. A sequence of instructions is known as a **micropogram**, or *firmware*. This latter term reflects the fact that a micropogram is midway between hardware and software. It is easier to design in firmware than hardware, but it is more difficult to write a firmware program than a software program.

How can we use the concept of micropogramming to implement a control unit? Consider that for each micro-operation, all that the control unit is allowed to do is generate a set of control signals. Thus, for any micro-operation, each control line emanating from the control unit is either on or off. This condition can, of course, be represented by a binary digit for each control line. So we could construct a *control word* in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1s and 0s in the control word.

Suppose we string together a sequence of control words to represent the sequence of micro-operations performed by the control unit. Next, we must recognize that the sequence of micro-operations is not fixed. Sometimes we have an indirect cycle; sometimes we do not. So let us put our control words in a memory, with each word having a unique address. Now add an address field to each control word, indicating the location of the next control word to be executed if a certain condition is true (e.g., the indirect bit in a memory-reference instruction is 1). Also, add a few bits to specify the condition.

**Table 21.1** Machine Instruction Set for Wilkes Example

Order	Effect of Order
<i>A n</i>	$C(Acc) + C(n)$ to $Acc_1$
<i>S n</i>	$C(Acc) - C(n)$ to $Acc_1$
<i>H n</i>	$C(n)$ to $Acc_2$
<i>V n</i>	$C(Acc2) \times C(n)$ to $Acc$ , where $C(n) \geq 0$
<i>T n</i>	$C(Acc_1)$ to $n$ , 0 to $Acc$
<i>U n</i>	$C(Acc_1)$ to $n$
<i>R n</i>	$C(Acc) \times 2^{(n+1)}$ to $Acc$
<i>L n</i>	$C(Acc) \times 2^{n+1}$ to $Acc$
<i>G n</i>	IF $C(Acc) < 0$ , transfer control to $n$ ; if $C(Acc) \geq 0$ , ignore (i.e., proceed serially)
<i>I n</i>	Read next character on input mechanism into $n$
<i>O n</i>	Send $C(n)$ to output mechanism

*Notation:*  $Acc$  = accumulator

$Acc_1$  = most significant half of accumulator

$Acc_2$  = least significant half of accumulator

$n$  = storage location  $n$

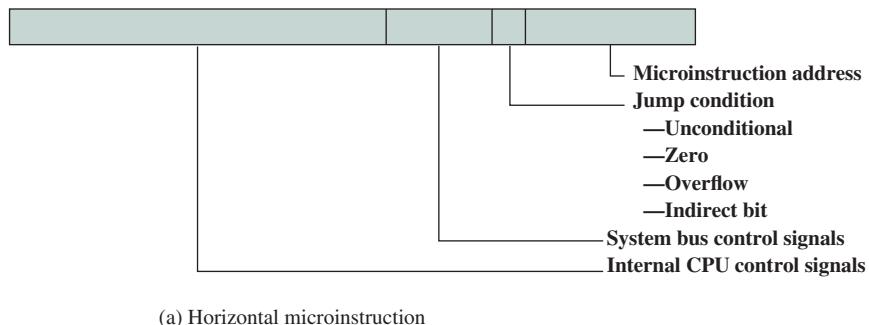
$C(X)$  = contents of  $X$  ( $X$  = register or storage location)

The result is known as a **horizontal microinstruction**, an example of which is shown in Figure 21.1a. The format of the microinstruction or control word is as follows. There is one bit for each internal processor control line and one bit for each system bus control line. There is a condition field indicating the condition under which there should be a branch, and there is a field with the address of the microinstruction to be executed next when a branch is taken. Such a microinstruction is interpreted as follows:

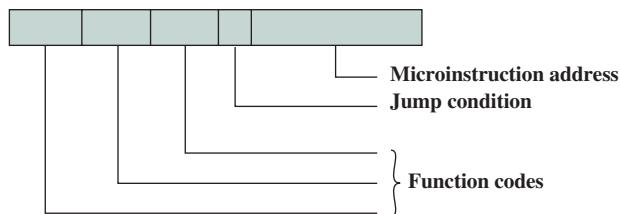
1. To execute this microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
2. If the condition indicated by the condition bits is false, execute the next microinstruction in sequence.
3. If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

Figure 21.2 shows how these control words or microinstructions could be arranged in a **control memory**. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

The control memory of Figure 21.2 is a concise description of the complete operation of the control unit. It defines the sequence of micro-operations to be

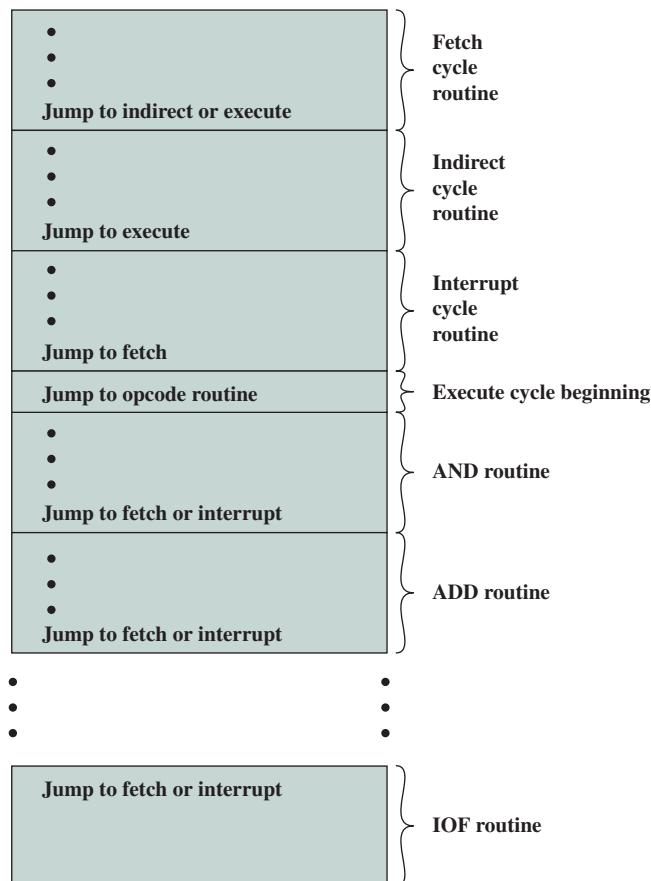


(a) Horizontal microinstruction



(b) Vertical microinstruction

**Figure 21.1** Typical Microinstruction Formats



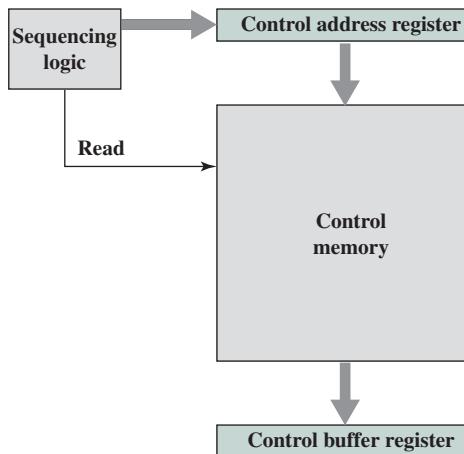
**Figure 21.2** Organization of Control Memory

performed during each cycle (fetch, indirect, execute, interrupt), and it specifies the sequencing of these cycles. If nothing else, this notation would be a useful device for documenting the functioning of a control unit for a particular computer. But it is more than that. It is also a way of implementing the control unit.

### **Microprogrammed Control Unit**

The control memory of Figure 21.2 contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program.

Figure 21.3 shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory*. The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a *control buffer register*. The left-hand portion of that register (see Figure 21.1a) connects to the control lines emanating from the control unit. Thus, *reading* a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

**Figure 21.3** Control Unit Microarchitecture

Let us examine this structure in greater detail, as depicted in Figure 21.4. Comparing this with Figure 21.3, we see that the control unit still has the same inputs (IR, ALU flags, clock) and outputs (control signals). The control unit functions as follows:

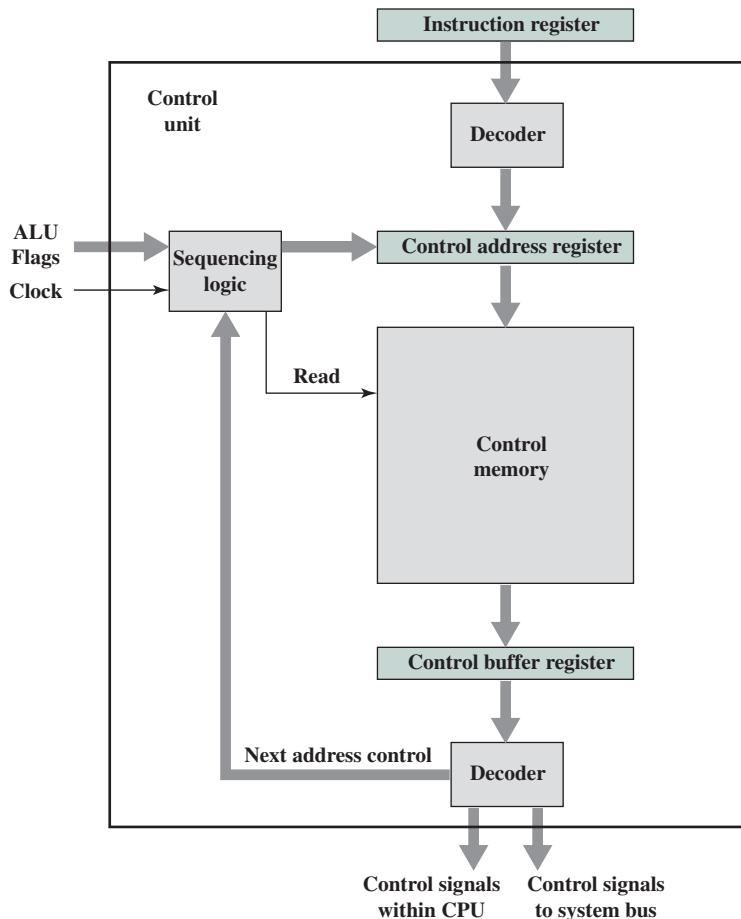
1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse.

The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

- **Get the next instruction:** Add 1 to the control address register.
- **Jump to a new routine based on a jump microinstruction:** Load the address field of the control buffer register into the control address register.
- **Jump to a machine instruction routine:** Load the control address register based on the opcode in the IR.

Figure 21.4 shows two modules labeled *decoder*. The upper decoder translates the opcode of the IR into a control memory address. The lower decoder is not used for horizontal microinstructions but is used for **vertical microinstructions** (Figure 21.1b). As was mentioned, in a horizontal microinstruction every bit in the



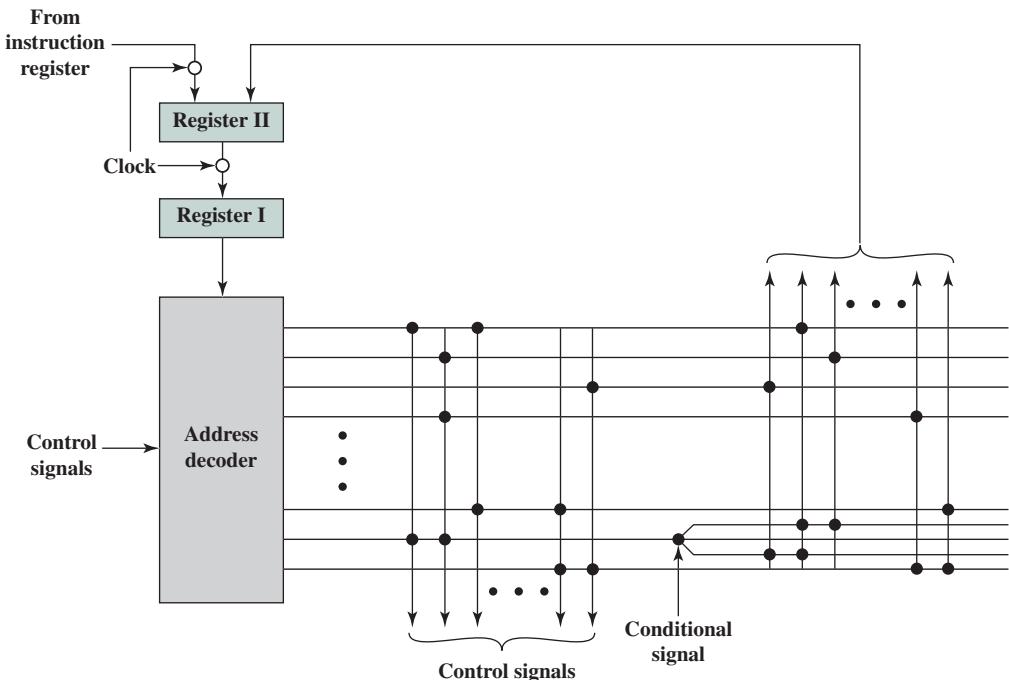
**Figure 21.4** Functioning of Microprogrammed Control Unit

control field attaches to a control line. In a vertical microinstruction, a code is used for each action to be performed [e.g., MAR  $\leftarrow$  (PC)], and the decoder translates this code into individual control signals. The advantage of vertical microinstructions is that they are more compact (fewer bits) than horizontal microinstructions, at the expense of a small additional amount of logic and time delay.

### Wilkes Control

As was mentioned, Wilkes first proposed the use of a microprogrammed control unit in 1951 [WILK51]. This proposal was subsequently elaborated into a more detailed design [WILK53]. It is instructive to examine this seminal proposal.

The configuration proposed by Wilkes is depicted in Figure 21.5. The heart of the system is a matrix partially filled with diodes. During a machine cycle, one row of the matrix is activated with a pulse. This generates signals at those points where a diode is present (indicated by a dot in the diagram). The first part of the row generates the control signals that control the operation of the processor. The second part generates the



**Figure 21.5** Wilkes's Microprogrammed Control Unit

address of the row to be pulsed in the next machine cycle. Thus, each row of the matrix is one microinstruction, and the layout of the matrix is the control memory.

At the beginning of the cycle, the address of the row to be pulsed is contained in Register I. This address is the input to the decoder, which, when activated by a clock pulse, activates one row of the matrix. Depending on the control signals, either the opcode in the instruction register or the second part of the pulsed row is passed into Register II during the cycle. Register II is then gated to Register I by a clock pulse. Alternating clock pulses are used to activate a row of the matrix and to transfer from Register II to Register I. The two-register arrangement is needed because the decoder is simply a combinatorial circuit; with only one register, the output would become the input during a cycle, causing an unstable condition.

This scheme is very similar to the horizontal microprogramming approach described earlier (Figure 21.1a). The main difference is this: In the previous description, the control address register could be incremented by one to get the next address. In the Wilkes scheme, the next address is contained in the microinstruction. To permit branching, a row must contain two address parts, controlled by a conditional signal (e.g., flag), as shown in the figure.

Having proposed this scheme, Wilkes provides an example of its use to implement the control unit of a simple machine. This example, the first known design of a microprogrammed processor, is worth repeating here because it illustrates many of the contemporary principles of microprogramming.

The processor of the hypothetical machine (the example machine by Wilkes) includes the following registers:

A	Multiplicand
B	Accumulator (least significant half)
C	Accumulator (most significant half)
D	Shift register

In addition, there are three registers and two 1-bit flags accessible only to the control unit. The registers are as follows:

E	Serves as both a memory address register (MAR) and temporary storage
F	Program counter
G	Another temporary register; used for counting

Table 21.1 lists the machine instruction set for this example. Table 21.2 is the complete set of microinstructions, expressed in symbolic form, that implements the control unit. Thus, a total of 38 microinstructions is all that is required to define the system completely.

The first full column gives the address (row number) of each microinstruction. Those addresses corresponding to opcodes are labeled. Thus, when the opcode for the add instruction (A) is encountered, the microinstruction at location 5 is executed. Columns 2 and 3 express the actions to be taken by the ALU and control unit, respectively. Each symbolic expression must be translated into a set of control signals (microinstruction bits). Columns 4 and 5 have to do with the setting and use of the two flags (flip-flops). Column 4 specifies the signal that sets the flag. For example, (1)C<sub>s</sub> means that flag number 1 is set by the sign bit of the number in register C. If column 5 contains a flag identifier, then columns 6 and 7 contain the two alternative microinstruction addresses to be used. Otherwise, column 6 specifies the address of the next microinstruction to be fetched.

Instructions 0 through 4 constitute the fetch cycle. Microinstruction 4 presents the opcode to a decoder, which generates the address of a microinstruction corresponding to the machine instruction to be fetched. The reader should be able to deduce the complete functioning of the control unit from a careful study of Table 21.2.

### Advantages and Disadvantages

The principal advantage of the use of microprogramming to implement a control unit is that it simplifies the design of the control unit. Thus, it is both cheaper and less error prone to implement. A *hardwired* control unit must contain complex logic for sequencing through the many micro-operations of the instruction cycle. On the other hand, the decoders and sequencing logic unit of a microprogrammed control unit are very simple pieces of logic.

The principal disadvantage of a microprogrammed unit is that it will be somewhat slower than a hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control units in pure CISC architectures, due to its ease of implementation. RISC processors, with their simpler instruction format, typically use hardwired control units. We now examine the microprogrammed approach in greater detail.

**Table 21.2** Microinstructions for Wilkes Example

Notations:  $A, B, C, \dots$  stand for the various registers in the arithmetical and control register units.  $C$  to  $D$  indicates that the switching circuits connect the output of register  $C$  to the input register  $D$ ;  $(D + A)$  to  $C$  indicates that the output register of  $A$  is connected to the one input of the adding unit (the output of  $D$  is permanently connected to the other input), and the output of the adder to register  $C$ . A numerical symbol  $n$  in quotes (e.g., “ $n$ ”) stands for the source whose output is the number  $n$  in units of the least significant digit.

		Arithmetical Unit	Control Register Unit	Conditional Flip-Flop	Next Microinstruction		
				Set	Use	0	1
	0		$F$ to $G$ and $E$			1	
	1		( $G$ to “1”) to $F$			2	
	2		Store to $G$			3	
	3		$G$ to $E$			4	
	4		$E$ to decoder			—	
$A$	5	$C$ to $D$				16	
$S$	6	$C$ to $D$				17	
$H$	7	Store to $B$				0	
$V$	8	Store to $A$				27	
$T$	9	$C$ to Store				25	
$U$	10	$C$ to Store				0	
$R$	11	$B$ to $D$	$E$ to $G$			19	
$L$	12	$C$ to $D$	$E$ to $G$			22	
$G$	13		$E$ to $G$	(1) $C_5$		18	
$I$	14	Input to Store				0	
$O$	15	Store to Output				0	
	16	( $D +$ Store) to $C$				0	
	17	( $D -$ Store) to $C$				0	
	18				1	0	1
	19	$D$ to $B$ ( $R$ )*	( $G - 1$ ) to $E$			20	
	20	$C$ to $D$		(1) $E_5$		21	
	21	$D$ to $C$ ( $R$ )			1	11	0
	22	$D$ to $C$ ( $L$ )†	( $G - 1$ ) to $E$			23	
	23	$B$ to $D$		(1) $E_5$		24	
	24	$D$ to $B$ ( $L$ )			1	12	0
	25	“0” to $B$				26	
	26	$B$ to $C$				0	
	27	“0” to $C$	“18” to $E$			28	
	28	$B$ to $D$	$E$ to $G$	(1) $B_1$		29	
	29	$D$ to $B$ ( $R$ )	( $G - “1”$ ) to $E$			30	
	30	$C$ to $D$ ( $R$ )		(2) $E_5$	1	31	32

		Arithmetical Unit	Control Register Unit	Conditional Flip-Flop	Next Microinstruction		
				Set	Use	0	1
31	$D$ to $C$				2	28	33
32	$(D + A)$ to $C$				2	28	33
33	$B$ to $D$			$(1)B_1$		34	
34	$D$ to $B$ ( $R$ )					35	
35	$C$ to $D$ ( $R$ )				1	36	37
36	$D$ to $C$					0	
37	$(D - A)$ to $C$					0	

\* Right shift. The switching circuits in the arithmetic unit are arranged so that the least significant digit of the register  $C$  is placed in the most significant place of register  $B$  during right shift micro-operations, and the most significant digit of register  $C$  (sign digit) is repeated (thus making the correction for negative numbers).

† Left shift. The switching circuits are similarly arranged to pass the most significant digit of register  $B$  to the least significant place of register  $C$  during left shift micro-operations.

## 21.2 MICROINSTRUCTION SEQUENCING

The two basic tasks performed by a microprogrammed control unit are as follows:

- **Microinstruction sequencing:** Get the next microinstruction from the control memory.
- **Microinstruction execution:** Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of the control unit. In this section, we will focus on sequencing and say as little as possible about format and timing issues. These issues are examined in more detail in the next section.

### Design Considerations

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address-generation time. The first concern is obvious; minimizing the size of the control memory reduces the cost of that component. The second concern is simply a desire to execute microinstructions as fast as possible.

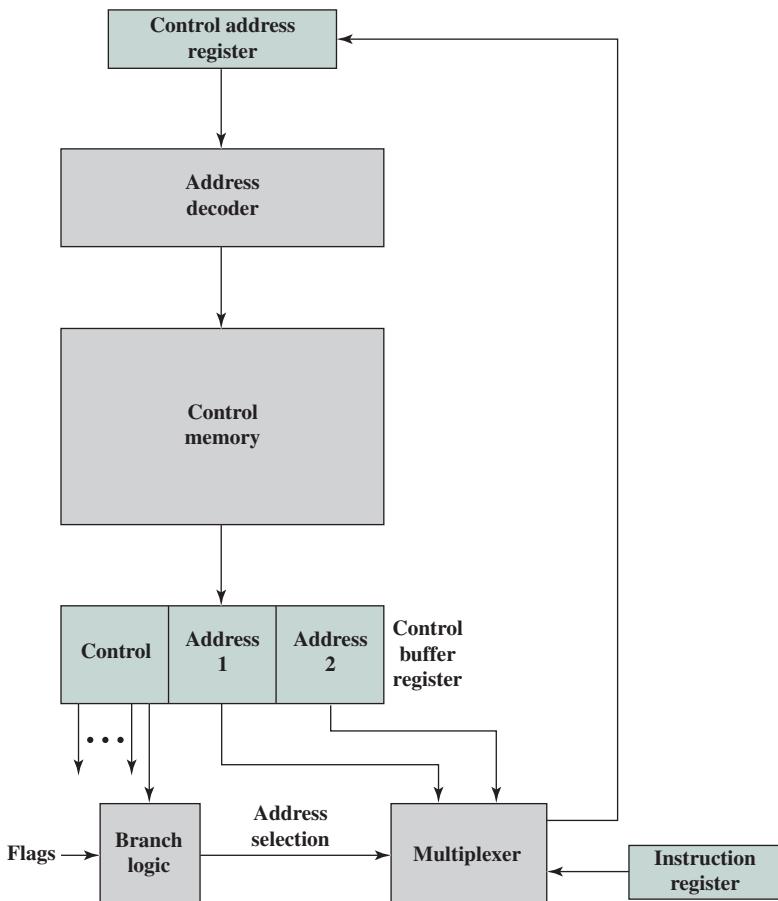
In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

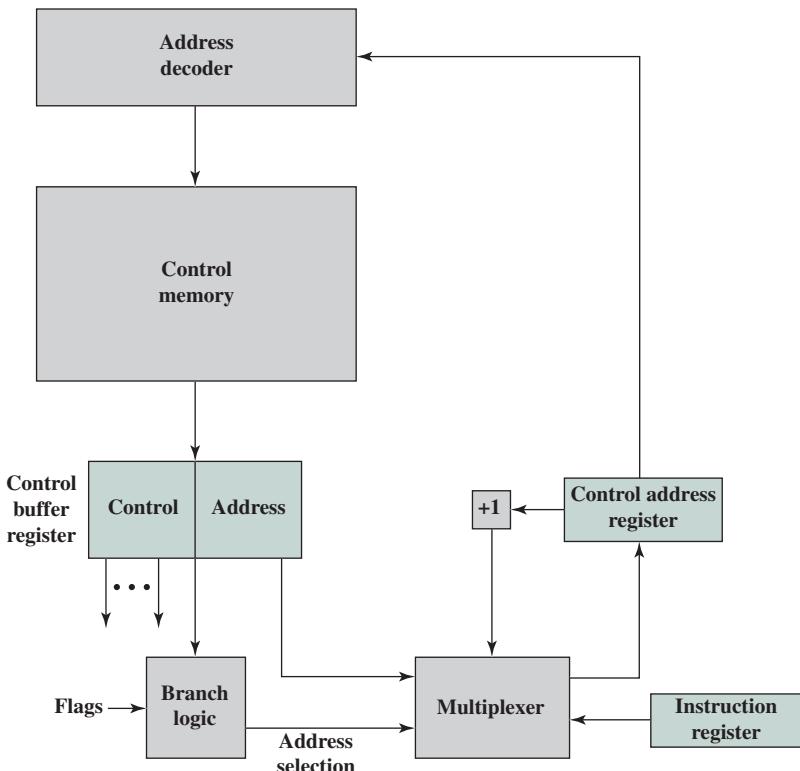
The first category occurs only once per instruction cycle, just after an instruction is fetched. The second category is the most common in most designs. However, the design cannot be optimized just for sequential access. Branches, both conditional and unconditional, are a necessary part of a microprogram. Furthermore, microinstruction sequences tend to be short; one out of every three or four microinstructions could be a branch [SIEW82]. Thus, it is important to design compact, time-efficient techniques for microinstruction branching.

### Sequencing Techniques

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. We can group them into three general categories, as illustrated in Figures 21.6 to 21.8. These categories are based on the format of the address information in the microinstruction:



**Figure 21.6** Branch Control Logic: Two Address Fields



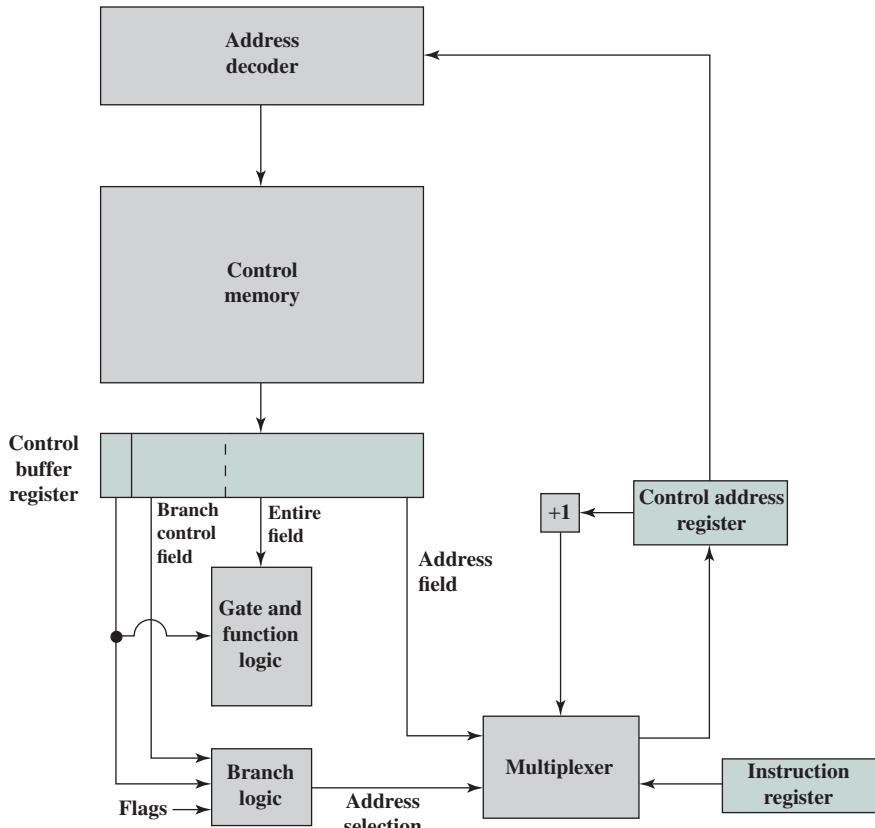
**Figure 21.7** Branch Control Logic: Single Address Field

- Two address fields
- Single address field
- Variable format

The simplest approach is to provide two address fields in each microinstruction. Figure 21.6 suggests how this information is to be used. A multiplexer is provided that serves as a destination for both address fields plus the instruction register. Based on an address-selection input, the multiplexer transmits either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address-selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

Although the two-address approach is simple, it requires more bits in the microinstruction than other approaches. With some additional logic, savings can be achieved. A common approach is to have a single address field (Figure 21.7). With this approach, the options for next address are as follows:

- Address field
- Instruction register code
- Next sequential address



**Figure 21.8** Branch Control Logic: Variable Format

The address-selection signals determine which option is selected. This approach reduces the number of address fields to one. Note, however, that the address field often will not be used. Thus, there is some inefficiency in the microinstruction coding scheme.

Another approach is to provide for two entirely different microinstruction formats (Figure 21.8). One bit designates which format is being used. In one format, the remaining bits are used to activate control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. One disadvantage of this approach is that one entire cycle is consumed with each branch microinstruction. With the other approaches, address generation occurs as part of the same cycle as control signal generation, minimizing control memory accesses.

The approaches just described are general. Specific implementations will often involve a variation or combination of these techniques.

## Address Generation

We have looked at the sequencing problem from the point of view of format considerations and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Table 21.3 lists the various address generation techniques. These can be divided into explicit techniques, in which the address is explicitly available in the microinstruction, and implicit techniques, which require additional logic to generate the address.

We have essentially dealt with the explicit techniques. With a two-field approach, two alternative addresses are available with each microinstruction. Using either a single address field or a variable format, various branch instructions can be implemented. A conditional branch instruction depends on the following types of information:

- ALU flags
- Part of the opcode or address mode fields of the machine instruction
- Parts of a selected register, such as the sign bit
- Status bits within the control unit

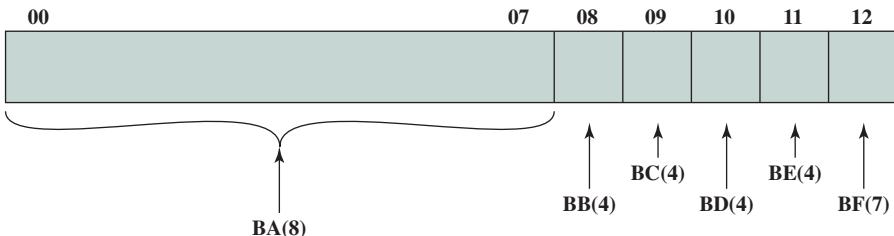
Several implicit techniques are also commonly used. One of these, mapping, is required with virtually all designs. The opcode portion of a machine instruction must be mapped into a microinstruction address. This occurs only once per instruction cycle.

A common implicit technique is one that involves combining or adding two portions of an address to form the complete address. This approach was taken for the IBM S/360 family [TUCK67] and used on many of the S/370 models. We will use the IBM 3033 as an example.

The control address register on the IBM 3033 is 13 bits long and is illustrated in Figure 21.9. Two parts of the address can be distinguished. The highest-order 8 bits (00–07) normally do not change from one microinstruction cycle to the next. During the execution of a microinstruction, these 8 bits are copied directly from an 8-bit field of the microinstruction (the BA field) into the highest-order 8 bits of the control address register. This defines a block of 32 microinstructions in control memory. The remaining 5 bits of the control address register are set to specify the specific address of the microinstruction to be fetched next. Each of these bits is determined by a 4-bit field (except one is a 7-bit field) in the current microinstruction; the field specifies the condition for setting the corresponding bit. For example, a bit in the control address register might be set to 1 or 0 depending on whether a carry occurred on the last ALU operation.

**Table 21.3** Microinstruction Address Generation Techniques

Explicit	Implicit
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control



**Figure 21.9** IBM 3033 Control Address Register

The final approach listed in Table 21.3 is termed *residual control*. This approach involves the use of a microinstruction address that has previously been saved in temporary storage within the control unit. For example, some microinstruction sets come equipped with a subroutine facility. An internal register or stack of registers is used to hold return addresses. An example of this approach is taken on the LSI-11, which we now examine.

### LSI-11 Microinstruction Sequencing

The LSI-11 is a microcomputer version of a PDP-11, with the main components of the system residing on a single board. The LSI-11 is implemented using a microprogrammed control unit [SEBE76].

The LSI-11 makes use of a 22-bit microinstruction and a control memory of 2K 22-bit words. The next microinstruction address is determined in one of five ways:

- **Next sequential address:** In the absence of other instructions, the control unit's control address register is incremented by 1.
- **Opcode mapping:** At the beginning of each instruction cycle, the next microinstruction address is determined by the opcode.
- **Subroutine facility:** Explained presently.
- **Interrupt testing:** Certain microinstructions specify a test for interrupts. If an interrupt has occurred, this determines the next microinstruction address.
- **Branch:** Conditional and unconditional branch microinstructions are used.

A one-level subroutine facility is provided. One bit in every microinstruction is dedicated to this task. When the bit is set, an 11-bit return register is loaded with the updated contents of the control address register. A subsequent microinstruction that specifies a return will cause the control address register to be loaded from the return register.

The return is one form of unconditional branch instruction. Another form of unconditional branch causes the bits of the control address register to be loaded from 11 bits of the microinstruction. The conditional branch instruction makes use of a 4-bit test code within the microinstruction. This code specifies testing of various ALU condition codes to determine the branch decision. If the condition is not true, the next sequential address is selected. If it is true, the 8 lowest-order bits of the control address register are loaded from 8 bits of the microinstruction. This allows branching within a 256-word page of memory.

As can be seen, the LSI-11 includes a powerful address sequencing facility within the control unit. This allows the microprogrammer considerable flexibility and can ease the microprogramming task. On the other hand, this approach requires more control unit logic than do simpler capabilities.

## 21.3 MICROINSTRUCTION EXECUTION

The microinstruction cycle is the basic event on a microprogrammed processor. Each cycle is made up of two parts: fetch and execute. The fetch portion is determined by the generation of a microinstruction address, and this was dealt with in the preceding section. This section deals with the execution of a microinstruction.

Recall that the effect of the execution of a microinstruction is to generate control signals. Some of these signals control points internal to the processor. The remaining signals go to the external control bus or other external interface. As an incidental function, the address of the next microinstruction is determined.

The preceding description suggests the organization of a control unit shown in Figure 21.10. This slightly revised version of Figure 21.4 emphasizes the focus of this section. The major modules in this diagram should by now be clear. The sequencing logic module contains the logic to perform the functions discussed in the preceding section. It generates the address of the next microinstruction, using as inputs the instruction register, ALU flags, the control address register (for incrementing), and the control buffer register. The last may provide an actual address, control bits, or both. The module is driven by a clock that determines the timing of the microinstruction cycle.

The control logic module generates control signals as a function of some of the bits in the microinstruction. It should be clear that the format and content of the microinstruction determines the complexity of the control logic module.

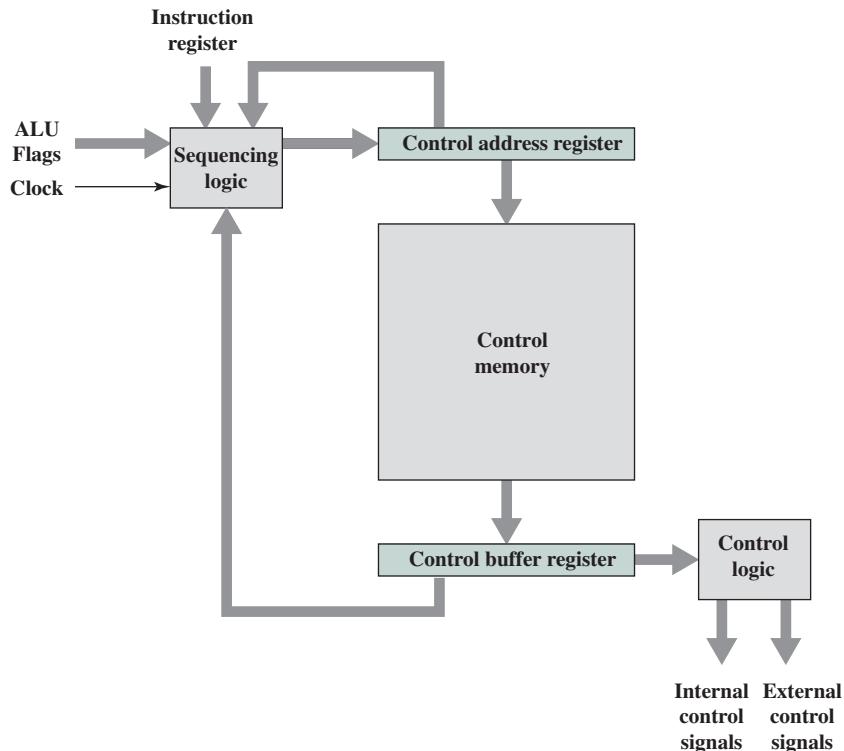
### A Taxonomy of Microinstructions

Microinstructions can be classified in a variety of ways. Distinctions that are commonly made in the literature include the following:

- Vertical/horizontal
- Packed/unpacked
- Hard/soft microprogramming
- Direct/indirect encoding

All of these bear on the format of the microinstruction. None of these terms has been used in a consistent, precise way in the literature. However, an examination of these pairs of qualities serves to illuminate microinstruction design alternatives. In the following paragraphs, we first look at the key design issue underlying all of these pairs of characteristics, and then we look at the concepts suggested by each pair.

In the original proposal by Wilkes [WILK51], each bit of a microinstruction either directly produced a control signal or directly produced one bit of the next address. We have seen, in the preceding section, that more complex address



**Figure 21.10** Control Unit Organization

sequencing schemes, using fewer microinstruction bits, are possible. These schemes require a more complex sequencing logic module. A similar sort of trade-off exists for the portion of the microinstruction concerned with control signals. By encoding control information, and subsequently decoding it to produce control signals, control word bits can be saved.

How can this encoding be done? To answer that, consider that there are a total of  $K$  different internal and external control signals to be driven by the control unit. In Wilkes's scheme,  $K$  bits of the microinstruction would be dedicated to this purpose. This allows all of the  $2K$  possible combinations of control signals to be generated during any instruction cycle. But we can do better than this if we observe that not all of the possible combinations will be used. Examples include the following:

- Two sources cannot be gated to the same destination (e.g.,  $C_2$  and  $C_8$  in Figure 21.5).
- A register cannot be both source and destination (e.g.,  $C_5$  and  $C_{12}$  in Figure 21.5).
- Only one pattern of control signals can be presented to the ALU at a time.
- Only one pattern of control signals can be presented to the external control bus at a time.

So, for a given processor, all possible allowable combinations of control signals could be listed, giving some number  $Q < 2^K$  possibilities. These could be encoded with a minimum  $\log_2 Q$  bits, with  $(\log_2 Q) < K$ . This would be the tightest possible form of encoding that preserves all allowable combinations of control signals. In practice, this form of encoding is not used, for two reasons:

- It is as difficult to program as a pure decoded (Wilkes) scheme. This point is discussed further presently.
- It requires a complex and therefore slow control logic module.

Instead, some compromises are made. These are of two kinds:

- More bits than are strictly necessary are used to encode the possible combinations.
- Some combinations that are physically allowable are not possible to encode.

The latter kind of compromise has the effect of reducing the number of bits. The net result, however, is to use more than  $\log_2 Q$  bits.

In the next subsection, we will discuss specific encoding techniques. The remainder of this subsection deals with the effects of encoding and the various terms used to describe it.

Based on the preceding, we can see that the control signal portion of the microinstruction format falls on a spectrum. At one extreme, there is one bit for each control signal; at the other extreme, a highly encoded format is used. Table 21.4 shows that other characteristics of a microprogrammed control unit also fall along a spectrum and that these spectra are, by and large, determined by the degree-of-encoding spectrum.

The second pair of items in the table is rather obvious. The pure Wilkes scheme will require the most bits. It should also be apparent that this extreme presents the most detailed view of the hardware. Every control signal is individually controllable

**Table 21.4** The Microinstruction Spectrum

Characteristics	
Unencoded	Highly encoded
Many bits	Few bits
Detailed view of hardware	Aggregated view of hardware
Difficult to program	Easy to program
Concurrency fully exploited	Concurrency not fully exploited
Little or no control logic	Complex control logic
Fast execution	Slow execution
Optimize performance	Optimize programming
Terminology	
Unpacked	Packed
Horizontal	Vertical
Hard	Soft

by the microprogrammer. Encoding is done in such a way as to aggregate functions or resources, so that the microprogrammer is viewing the processor at a higher, less detailed level. Furthermore, the encoding is designed to ease the microprogramming burden. Again, it should be clear that the task of understanding and orchestrating the use of all the control signals is a difficult one. As was mentioned, one of the consequences of encoding, typically, is to prevent the use of certain otherwise allowable combinations.

The preceding paragraph discusses microinstruction design from the microprogrammer's point of view. But the degree of encoding also can be viewed from its hardware effects. With a pure unencoded format, little or no decode logic is needed; each bit generates a particular control signal. As more compact and more aggregated encoding schemes are used, more complex decode logic is needed. This, in turn, may affect performance. More time is needed to propagate signals through the gates of the more complex control logic module. Thus, the execution of encoded microinstructions takes longer than the execution of unencoded ones.

Therefore, all of the characteristics listed in Table 21.4 fall along a spectrum of design strategies. In general, a design that falls toward the left end of the spectrum is intended to optimize the performance of the control unit. Designs toward the right end are more concerned with optimizing the process of microprogramming. Indeed, microinstruction sets near the right end of the spectrum look very much like machine instruction sets. A good example of this is the LSI-11 design, described later in this section. Typically, when the objective is simply to implement a control unit, the design will be near the left end of the spectrum. The IBM 3033 design, discussed presently, is in this category. As we shall discuss later, some systems permit a variety of users to construct different microprograms using the same microinstruction facility. In the latter cases, the design is likely to fall near the right end of the spectrum.

We can now deal with some of the terminology introduced earlier. Table 21.4 indicates how three of these pairs of terms relate to the microinstruction spectrum. In essence, all of these pairs describe the same thing but emphasize different design characteristics.

The degree of packing relates to the degree of identification between a given control task and specific microinstruction bits. As the bits become more *packed*, a given number of bits contains more information. Thus, packing connotes encoding. The terms *horizontal* and *vertical* relate to the relative width of microinstructions. [SIEW82] suggests as a rule of thumb that vertical microinstructions have lengths in the range of 16 to 40 bits and that horizontal microinstructions have lengths in the range of 40 to 100 bits. The terms *hard* and *soft* microprogramming are used to suggest the degree of closeness to the underlying control signals and hardware layout. Hard microprograms are generally fixed and committed to read-only memory. Soft microprograms are more changeable and are suggestive of user microprogramming.

The other pair of terms mentioned at the beginning of this subsection refers to direct versus indirect encoding, a subject to which we now turn.

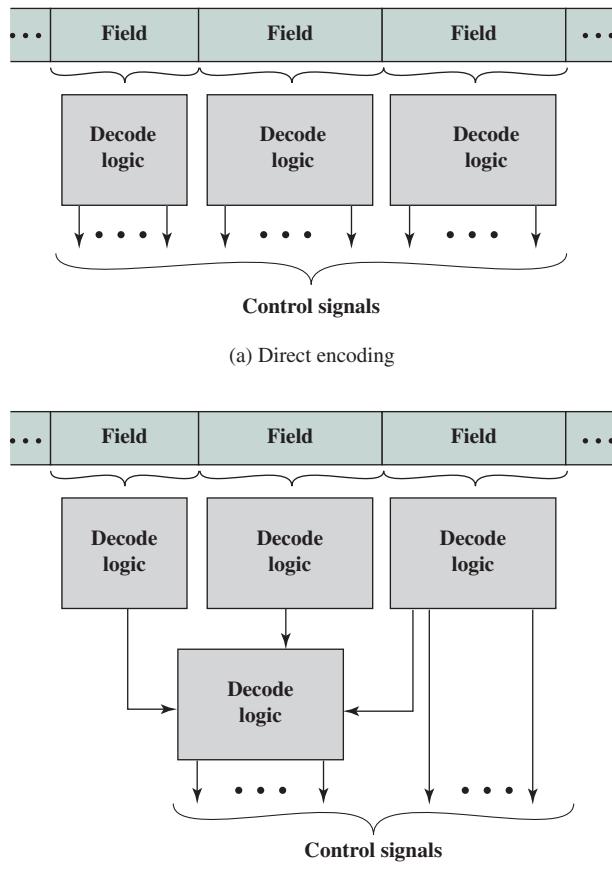
### Microinstruction Encoding

In practice, microprogrammed control units are not designed using a pure unencoded or horizontal microinstruction format. At least some degree of encoding is used to reduce control memory width and to simplify the task of microprogramming.

The basic technique for encoding is illustrated in Figure 21.11a. The microinstruction is organized as a set of fields. Each field contains a code, which, upon decoding, activates one or more control signals.

Let us consider the implications of this layout. When the microinstruction is executed, every field is decoded and generates control signals. Thus, with  $N$  fields,  $N$  simultaneous actions are specified. Each action results in the activation of one or more control signals. Generally, but not always, we will want to design the format so that each control signal is activated by no more than one field. Clearly, however, it must be possible for each control signal to be activated by at least one field.

Now consider the individual field. A field consisting of  $L$  bits can contain one of  $2^L$  codes, each of which can be encoded to a different control signal pattern. Because only one code can appear in a field at a time, the codes are mutually exclusive, and, therefore, the actions they cause are mutually exclusive.



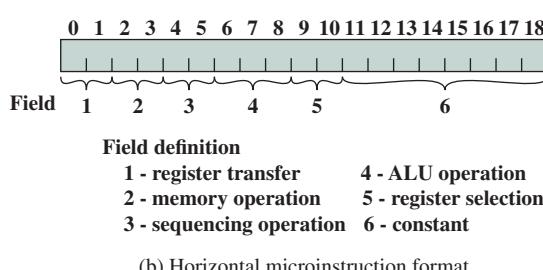
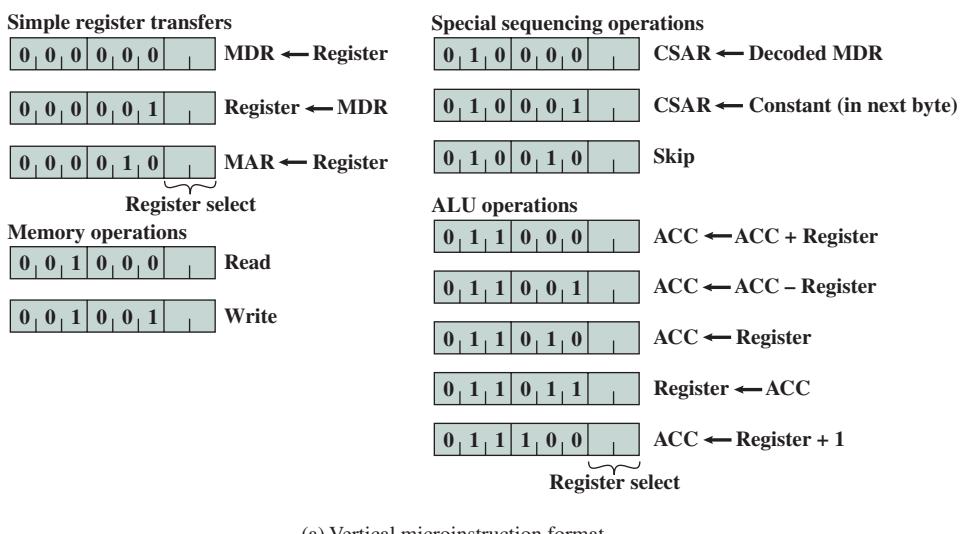
**Figure 21.11** Microinstruction Encoding

The design of an encoded microinstruction format can now be stated in simple terms:

- Organize the format into independent fields. That is, each field depicts a set of actions (pattern of control signals) such that actions from different fields can occur simultaneously.
- Define each field such that the alternative actions that can be specified by the field are mutually exclusive. That is, only one of the actions specified for a given field could occur at a time.

Two approaches can be taken for organizing the encoded microinstruction into fields: functional and resource. The *functional encoding* method identifies functions within the machine and designates fields by function type. For example, if various sources can be used for transferring data to the accumulator, one field can be designated for this purpose, with each code specifying a different source. *Resource encoding* views the machine as consisting of a set of independent resources and devotes one field to each (e.g., I/O, memory, ALU).

Another aspect of encoding is whether it is direct or indirect (Figure 21.11b). With indirect encoding, one field is used to determine the interpretation of another



**Figure 21.12** Alternative Microinstruction Formats for a Simple Machine

field. For example, consider an ALU that is capable of performing eight different arithmetic operations and eight different shift operations. A 1-bit field could be used to indicate whether a shift or arithmetic operation is to be used; a 3-bit field would indicate the operation. This technique generally implies two levels of decoding, increasing propagation delays.

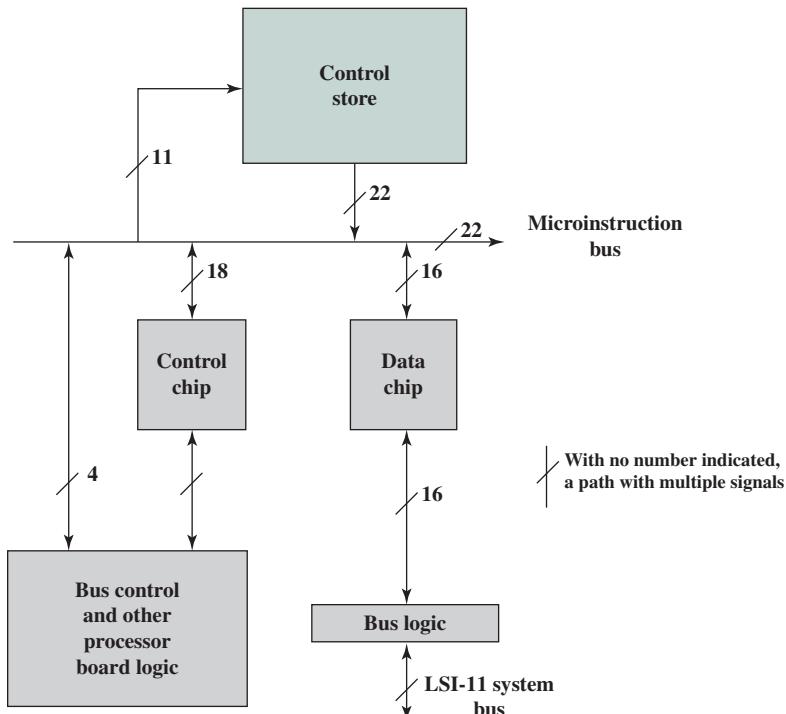
Figure 21.12 is a simple example of these concepts. Assume a processor with a single accumulator and several internal registers, such as a program counter and a temporary register for ALU input. Figure 21.12a shows a highly vertical format. The first 3 bits indicate the type of operation, the next 3 encode the operation, and the final 2 select an internal register. Figure 21.12b is a more horizontal approach, although encoding is still used. In this case, different functions appear in different fields.

### LSI-11 Microinstruction Execution

The LSI-11 [SEBE76] is a good example of a vertical microinstruction approach. We look first at the organization of the control unit, then at the microinstruction format.

**LSI-11 CONTROL UNIT ORGANIZATION** The LSI-11 is the first member of the PDP-11 family that was offered as a single-board processor. The board contains three LSI chips, an internal bus known as the *microinstruction bus* (MIB), and some additional interfacing logic.

Figure 21.13 depicts, in simplified form, the organization of the LSI-11 processor. The three chips are the data, control, and control store chips. The data chip contains an 8-bit ALU, twenty-six 8-bit registers, and storage for several condition



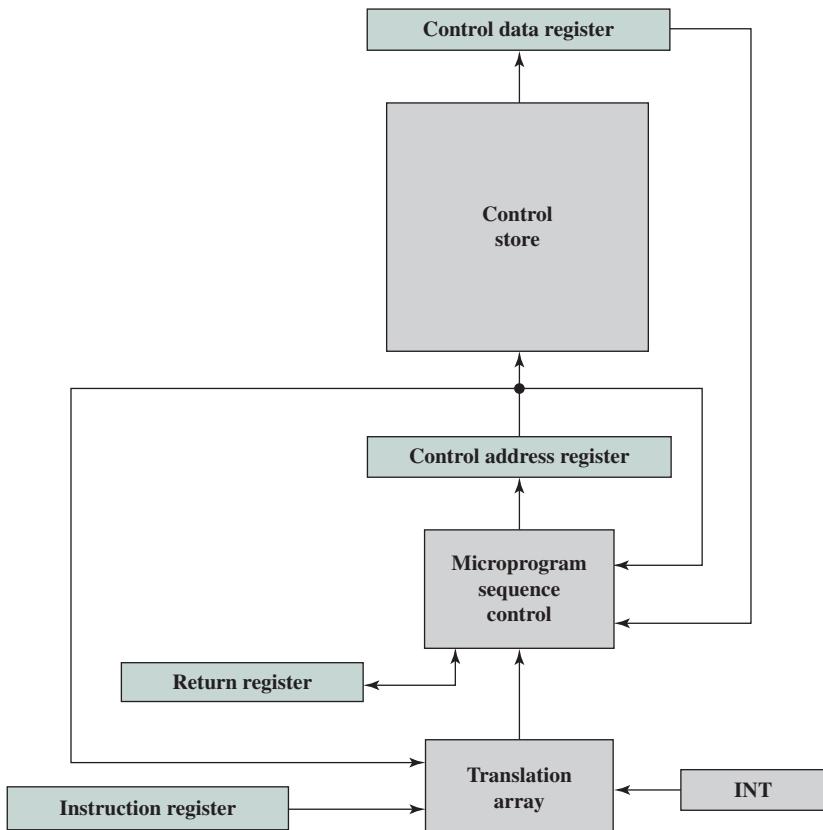
**Figure 21.13** Simplified Block Diagram of the LSI-11 Processor

codes. Sixteen of the registers are used to implement the eight 16-bit general-purpose registers of the PDP-11. Others include a program status word, memory address register (MAR), and memory buffer register. Because the ALU deals with only 8 bits at a time, two passes through the ALU are required to implement a 16-bit PDP-11 arithmetic operation. This is controlled by the microprogram.

The control store chip or chips contain the 22-bit-wide control memory. The control chip contains the logic for sequencing and executing microinstructions. It contains the control address register, the control data register, and a copy of the machine instruction register.

The MIB ties all the components together. During microinstruction fetch, the control chip generates an 11-bit address onto the MIB. Control store is accessed, producing a 22-bit microinstruction, which is placed on the MIB. The low-order 16 bits go to the data chip, while the low-order 18 bits go to the control chip. The high-order 4 bits control special processor board functions.

Figure 21.14 provides a still simplified but more detailed look at the LSI-11 control unit: the figure ignores individual chip boundaries. The address sequencing scheme described in Section 21.2 is implemented in two modules. Overall sequence control is provided by the microprogram sequence control module, which is capable



**Figure 21.14** Organization of the LSI-11 Control Unit

of incrementing the microinstruction address register and performing unconditional branches. The other forms of address calculation are carried out by a separate translation array. This is a combinatorial circuit that generates an address based on the microinstruction, the machine instruction, the microinstruction program counter, and an interrupt register.

The translation array comes into play on the following occasions:

- The opcode is used to determine the start of a microroutine.
- At appropriate times, address mode bits of the microinstruction are tested to perform appropriate addressing.
- Interrupt conditions are periodically tested.
- Conditional branch microinstructions are evaluated.

**LSI-11 MICROINSTRUCTION FORMAT** The LSI-11 uses an extremely vertical microinstruction format, which is only 22 bits wide. The microinstruction set strongly resembles the PDP-11 machine instruction set that it implements. This design was intended to optimize the performance of the control unit within the constraint of a vertical, easily programmed design. Table 21.5 lists some of the LSI-11 microinstructions.

Figure 21.15 shows the 22-bit LSI-11 microinstruction format. The high-order 4 bits control special functions on the processor board. The translate bit enables the translation array to check for pending interrupts. The load return register bit is used at the end of a microroutine to cause the next microinstruction address to be loaded from the return register.

The remaining 16 bits are used for highly encoded micro-operations. The format is much like a machine instruction, with a variable-length opcode and one or more operands.

**Table 21.5** Some LSI-11 Microinstructions

<b>Arithmetic Operations</b>	<b>General Operations</b>
Add word (byte, literal)	MOV word (byte)
Test word (byte, literal)	Jump
Increment word (byte) by 1	Return
Increment word (byte) by 2	Conditional jump
Negate word (byte)	Set (reset) flags
Conditionally increment (decrement) byte	Load G low
Conditionally add word (byte)	Conditionally MOV word (byte)
Add word (byte) with carry	
Conditionally add digits	
Subtract word (byte)	
Compare word (byte, literal)	
Subtract word (byte) with carry	
Decrement word (byte) by 1	
<b>Logical Operations</b>	<b>Input/Output Operations</b>
AND word (byte, literal)	Input word (byte)
Test word (byte)	Input status word (byte)
OR word (byte)	Read
Exclusive-OR word (byte)	Write
Bit clear word (byte)	Read (write) and increment word (byte) by 1
Shift word (byte) right (left) with (without) carry	Read (write) and increment word (byte) by 2
Complement word (byte)	Read (write) acknowledge
	Output word (byte, status)

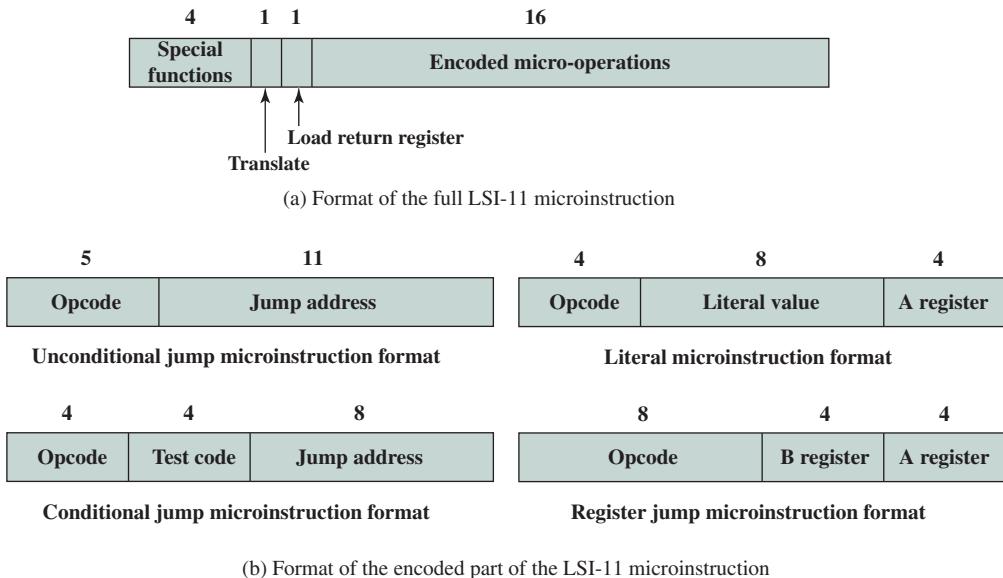


Figure 21.15 LSI-11 Microinstruction Format

### IBM 3033 Microinstruction Execution

The standard IBM 3033 control memory consists of 4K words. The first half of these (0000–07FF) contain 108-bit microinstructions, while the remainder (0800–0FFF) are used to store 126-bit microinstructions. The format is depicted in Figure 21.16.

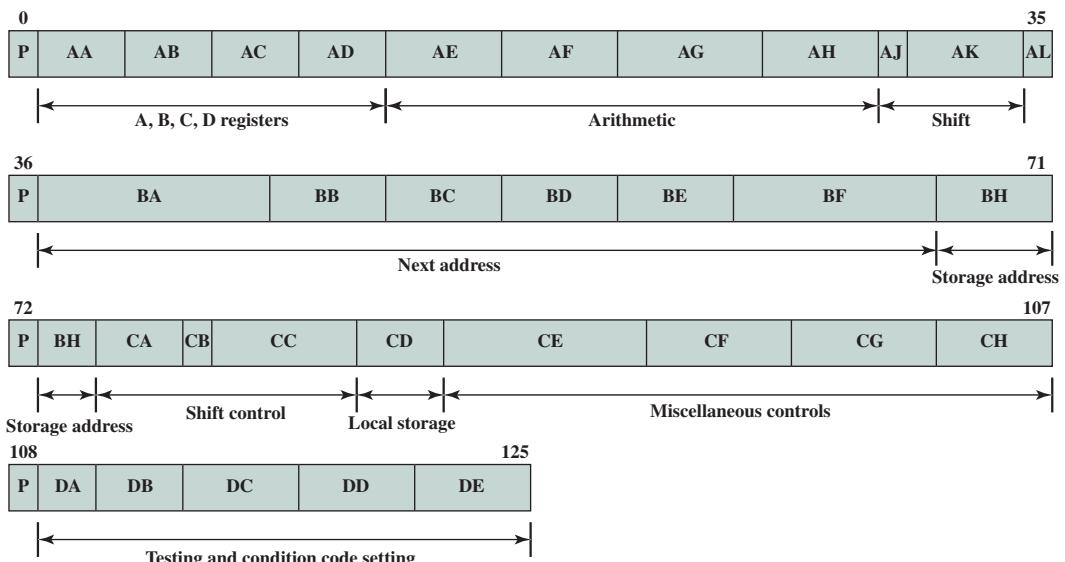


Figure 21.16 IBM 3033 Microinstruction Format

**Table 21.6** IBM 3033 Microinstruction Control Fields

ALU Control Fields	
AA(3)	Load A register from one of data registers
AB(3)	Load B register from one of data registers
AC(3)	Load C register from one of data registers
AD(3)	Load D register from one of data registers
AE(4)	Route specified A bits to ALU
AF(4)	Route specified B bits to ALU
AG(5)	Specifies ALU arithmetic operation on A input
AH(4)	Specifies ALU arithmetic operation on B input
AJ(1)	Specifies D or B input to ALU on B side
AK(4)	Route arithmetic output to shifter
CA(3)	Load F register
CB(1)	Activate shifter
CC(5)	Specifies logical and carry functions
CE(7)	Specifies shift amount
Sequencing and Branching Fields	
AL(1)	End operation and perform branch
BA(8)	Set high-order bits (00–07) of control address register
BB(4)	Specifies condition for setting bit 8 of control address register
BC(4)	Specifies condition for setting bit 9 of control address register
BD(4)	Specifies condition for setting bit 10 of control address register
BE(4)	Specifies condition for setting bit 11 of control address register
BF(7)	Specifies condition for setting bit 12 of control address register

Although this is a rather horizontal format, encoding is still extensively used. The key fields of that format are summarized in Table 21.6.

The ALU operates on inputs from four dedicated, non-user-visible registers, A, B, C, and D. The microinstruction format contains fields for loading these registers from user-visible registers, performing an ALU function, and specifying a user-visible register for storing the result. There are also fields for loading and storing data between registers and memory.

The sequencing mechanism for the IBM 3033 was discussed in Section 21.2.

## 21.4 TI 8800

The Texas Instruments 8800 Software Development Board (SDB) is a microprogrammable 32-bit computer card. The system has a writable control store, implemented in RAM rather than ROM. Such a system does not achieve the speed or

density of a microprogrammed system with a ROM control store. However, it is useful for developing prototypes and for educational purposes.

The 8800 SDB consists of the following components (Figure 21.17):

- Microcode memory
- Microsequencer
- 32-bit ALU
- Floating-point and integer processor
- Local data memory

Two buses link the internal components of the system. The DA bus provides data from the microinstruction data field to the ALU, the floating-point processor, or the microsequencer. In the latter case, the data consists of an address to be used for a branch instruction. The bus can also be used for the ALU or microsequencer to

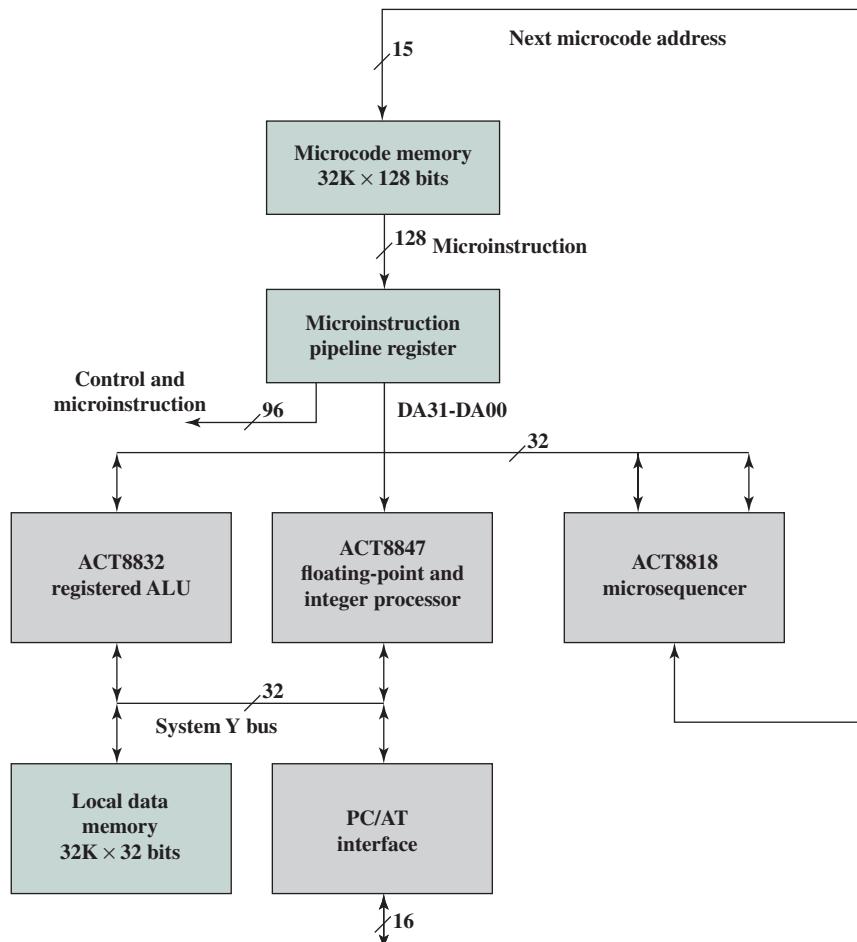


Figure 21.17 TI 8800 Block Diagram

provide data to other components. The system Y bus connects the ALU and floating-point processor to local memory and to external modules via the PC interface.

The board fits into an IBM PC-compatible host computer. The host computer provides a suitable platform for microcode assembly and debug.

## Microinstruction Format

The microinstruction format for the 8800 consists of 128 bits broken down into 30 functional fields, as indicated in Table 21.7. Each field consists of one or more bits, and the fields are grouped into five major categories:

- Control of board
- 8847 floating-point and integer processor chip
- 8832 registered ALU
- 8818 microsequencer
- WCS data field

As indicated in Figure 21.17, the 32 bits of the WCS data field are fed into the DA bus to be provided as data to the ALU, floating-point processor, or microsequencer. The other 96 bits (fields 1–27) of the microinstruction are control signals that are fed directly to the appropriate module. For simplicity, these other connections are not shown in Figure 21.17.

The first six fields deal with operations that pertain to the control of the board, rather than controlling an individual component. Control operations include the following:

- Selecting condition codes for sequencer control. The first bit of field 1 indicates whether the condition flag is to be set to 1 or 0, and the remaining 4 bits indicate which flag is to be set.
- Sending an I/O request to the PC/AT.
- Enabling local data memory read/write operations.
- Determining the unit driving the system Y bus. One of the four devices attached to the bus (Figure 21.17) is selected.

The last 32 bits are the data field, which contain information specific to a particular microinstruction.

The remaining fields of the microinstruction are best discussed in the context of the device that they control. In the remainder of this section, we discuss the microsequencer and the registered ALU. The floating-point unit introduces no new concepts and is skipped.

## Microsequencer

The principal function of the 8818 microsequencer is to generate the next microinstruction address for the microprogram. This 15-bit address is provided to the microcode memory (Figure 21.17).

The next address can be selected from one of five sources:

1. The microprogram counter (MPC) register, used for repeat (reuse same address) and continue (increment address by 1) instructions.

**Table 21.7** TI 8800 Microinstruction Format

Field Number	Number of Bits	Description
<b>Control of Board</b>		
1	5	Select condition code input
2	1	Enable/disable external I/O request signal
3	2	Enable/disable local data memory read/write operations
4	1	Load status/do no load status
5	2	Determine unit driving Y bus
6	2	Determine unit driving DA bus
<b>8847 Floating-Point and Integer Processing Chip</b>		
7	1	C register control: clock, do not clock
8	1	Select most significant or least significant bits for Y bus
9	1	C register data source: ALU, multiplexer
10	4	Select IEEE or FAST mode for ALU and MUL
11	8	Select sources for data operands: RA registers, RB registers, P register, 5 register, C register
12	1	RB register control: clock, do not clock
13	1	RA register control: clock, do not clock
14	2	Data source confirmation
15	2	Enable/disable pipeline registers
16	11	8847 ALU function
<b>8832 Registered ALU</b>		
17	2	Write enable/disable data output to selected register: most significant half, least significant half
18	2	Select register file data source: DA bus, DB bus, ALU Y MUX output, system Y bus
19	3	Shift instruction modifier
20	1	Carry in: force, do not force
21	2	Set ALU configuration mode: 32, 16, or 8 bits
22	2	Select input to 5 multiplexer: register file, DB bus, MQ register
23	1	Select input to R multiplexer: register file, DA bus
24	6	Select register in file C for write
25	6	Select register in file B for read
26	6	Select register in file A for write
27	8	ALU function
<b>8818 Microsequencer</b>		
28	12	Control input signals to the 8818
<b>WCS Data Field</b>		
29	16	Most significant bits of writable control store data field
30	16	Least significant bits of writable control store data field

2. The stack, which supports microprogram subroutine calls as well as iterative loops and returns from interrupts.
3. The DRA and DRB ports, which provide two additional paths from external hardware by which microprogram addresses can be generated. These two ports are connected to the most significant and least significant 16 bits of the DA bus, respectively. This allows the microsequencer to obtain the next instruction address from the WCS data field of the current microinstruction or from a result calculated by the ALU.
4. Register counters RCA and RCB, which can be used for additional address storage.
5. An external input onto the bidirectional Y port to support external interrupts.

Figure 21.18 is a logical block diagram of the 8818. The device consists of the following principal functional groups:

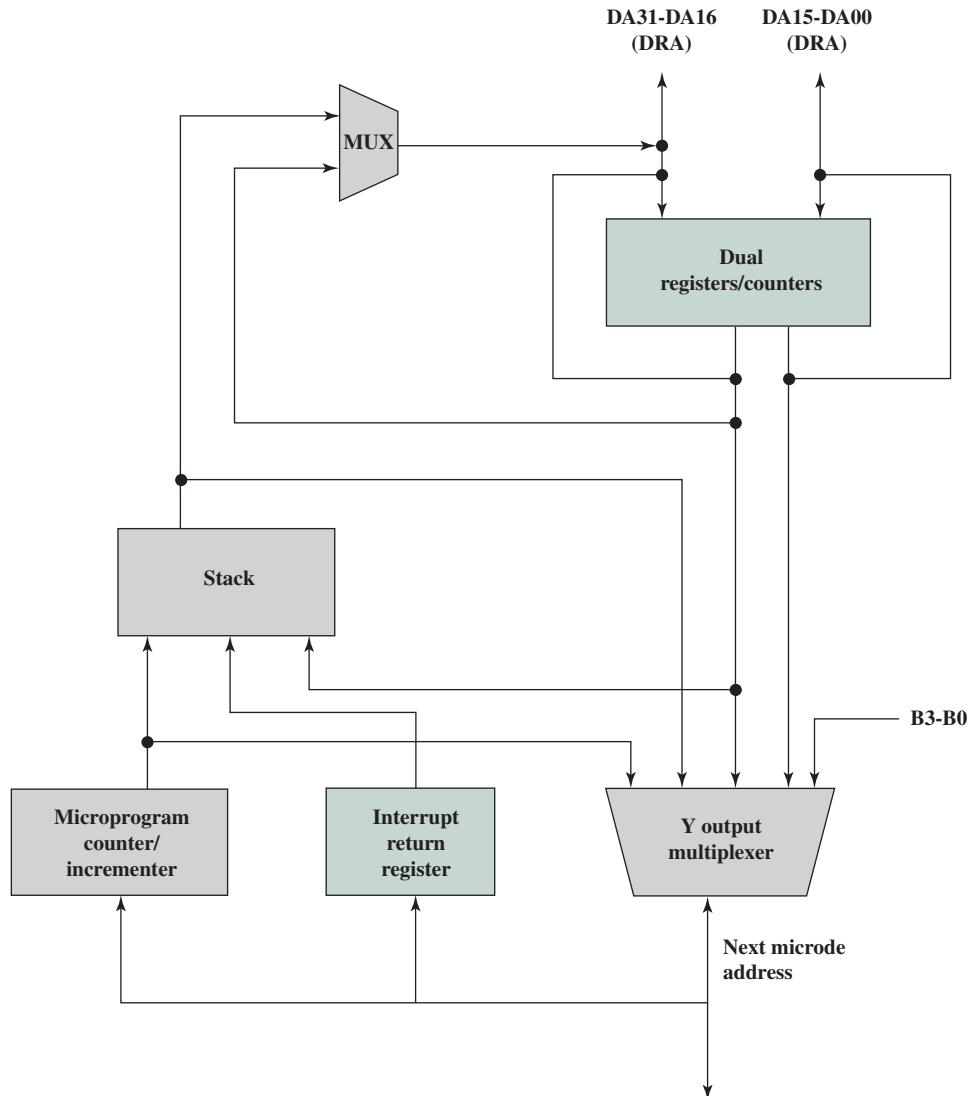
- A 16-bit microprogram counter (MPC) consisting of a register and an incrementer.
- Two register counters, RCA and RCB, for counting loops and iterations, storing branch addresses, or driving external devices.
- A 65-word by 16-bit stack, which allows microprogram subroutine calls and interrupts.
- An interrupt return register and Y output enable for interrupt processing at the microinstruction level.
- A Y output multiplexer by which the next address can be selected from MPC, RCA, RCB, external buses DRA and DRB, or the stack.

**REGISTERS/COUNTERS** The registers RCA and RCB may be loaded from the DA bus, either from the current microinstruction or from the output of the ALU. The values may be used as counters to control the flow of execution and may be automatically decremented when accessed. The values may also be used as microinstruction addresses to be supplied to the Y output multiplexer. Independent control of both registers during a single microinstruction cycle is supported with the exception of simultaneous decrement of both registers.

**STACK** The stack allows multiple levels of nested calls or interrupts, and it can be used to support branching and looping. Keep in mind that these operations refer to the control unit, not the overall processor, and that the addresses involved are those of microinstructions in the control memory.

Six stack operations are possible:

1. Clear, which sets the stack pointer to zero, emptying the stack;
2. Pop, which decrements the stack pointer;
3. Push, which puts the contents of the MPC, interrupt return register, or DRA bus onto the stack and increments the stack pointer;
4. Read, which makes the address indicated by the read pointer available at the Y output multiplexer;



**Figure 21.18** TI 8818 Microsequencer

5. Hold, which causes the address of the stack pointer to remain unchanged;
6. Load stack pointer, which inputs the seven least significant bits of DRA to the stack pointer.

**CONTROL OF MICROSEQUENCER** The microsequencer is controlled primarily by the 12-bit field of the current microinstruction, field 28 (Table 21.7). This field consists of the following subfields:

- **OSEL (1 bit):** Output select. Determines which value will be placed on the output of the multiplexer that feeds into the DRA bus (upper-left-hand corner of

Figure 21.18). The output is selected to come from either the stack or from register RCA. DRA then serves as input to either the Y output multiplexer or to register RCA.

- **SELDR (1 bit):** Select DR bus. If set to 1, this bit selects the external DA bus as input to the DRA/DRB buses. If set to 0, selects the output of the DRA multiplexer to the DRA bus (controlled by OSEL) and the contents of RCB to the DRB bus.
- **ZEROIN (1 bit):** Used to indicate a conditional branch. The behavior of the microsequencer will then depend on the condition code selected in field 1 (Table 21.7).
- **RC2–RC0 (3 bits):** Register controls. These bits determine the change in the contents of registers RCA and RCB. Each register can either remain the same, decrement, or load from the DRA/DRB buses.
- **S2–S0 (3 bits):** Stack controls. These bits determine which stack operation is to be performed.
- **MUX2–MUX0:** Output controls. These bits, together with the condition code if used, control the Y output multiplexer and therefore the next microinstruction address. The multiplexer can select its output from the stack, DRA, DRB, or MPC.

These bits can be set individually by the programmer. However, this is typically not done. Rather, the programmer uses mnemonics that equate to the bit patterns that would normally be required. Table 21.8 lists the 15 mnemonics for field 28. A microcode assembler converts these into the appropriate bit patterns.

**Table 21.8** TI 8818 Microsequencer Microinstruction Bits (Field 28)

Mnemonic	Value	Description
RST8818	00000000110	Reset Instruction
BRA88181	011000111000	Branch to DRA Instruction
BRA88180	010000111110	Branch to DRA Instruction
INC88181	000000111110	Continue Instruction
INC88180	001000001000	Continue Instruction
CAL88181	010000110000	Jump to Subroutine at Address Specified by DRA
CAL88180	010000101110	Jump to Subroutine at Address Specified by DRA
RET8818	000000011010	Return from Subroutine
PUSH8818	000000110111	Push Interrupt Return Address onto Stack
POP8818	100000010000	Return from Interrupt
LOADDRA	000010111110	Load DRA Counter from DA Bus
LOADDRB	000110111110	Load DRB Counter from DA Bus
LOADDRAB	000110111100	Load DRA/DRB
DEC RDRA	010001111100	Decrement DRA Counter and Branch If Not Zero
DEC RDRB	010101111100	Decrement DRB Counter and Branch If Not Zero

As an example, the instruction INC88181 is used to cause the next microinstruction in sequence to be selected, if the currently selected condition code is 1. From Table 21.8, we have

$$\text{INC88181} = 000000111110$$

which decodes directly into

- **OSEL = 0:** Selects RCA as output from DRA output MUX; in this case the selection is irrelevant.
- **SELDI = 0:** As defined previously; again, this is irrelevant for this instruction.
- **ZEROIN = 0:** Combined with the value for MUX, indicates no branch should be taken.
- **R = 000:** Retain current value of RA and RC.
- **S = 111:** Retain current state of stack.
- **MUX = 110:** Choose MPC when condition code = 1, DRA when condition code = 0.

### Registered ALU

The 8832 is a 32-bit ALU with 64 registers that can be configured to operate as four 8-bit ALUs, two 16-bit ALUs, or a single 32-bit ALU.

The 8832 is controlled by the 39 bits that make up fields 17 through 27 of the microinstruction (Table 21.7); these are supplied to the ALU as control signals. In addition, as indicated in Figure 21.17, the 8832 has external connections to the 32-bit DA bus and the 32-bit system Y bus. Inputs from the DA can be provided simultaneously as input data to the 64-word register file and to the ALU logic module. Input from the system Y bus is provided to the ALU logic module. Results of the ALU and shift operations are output to the DA bus or the system Y bus. Results can also be fed back to the internal register file.

Three 6-bit address ports allow a two-operand fetch and an operand write to be performed within the register file simultaneously. An MQ shifter and MQ register can also be configured to function independently to implement double-precision 8-bit, 16-bit, and 32-bit shift operations.

Fields 17 through 26 of each microinstruction control the way in which data flows within the 8832 and between the 8832 and the external environment. The fields are as follows:

- 17. Write Enable.** These two bits specify write 32 bits, 16 most significant bits, 16 least significant bits, or do not write into register file. The destination register is defined by field 24.
- 18. Select Register File Data Source.** If a write is to occur to the register file, these two bits specify the source: DA bus, DB bus, ALU output, or system Y bus.
- 19. Shift Instruction Modifier.** Specifies options concerning supplying end fill bits and reading bits that are shifted during shift instructions.
- 20. Carry In.** This bit indicates whether a bit is carried into the ALU for this operation.

- 21. ALU Configuration Mode.** The 8832 can be configured to operate as a single 32-bit ALU, two 16-bit ALUs, or four 8-bit ALUs.
- 22. S Input.** The ALU logic module inputs are provided by two internal multiplexers referred to as the S and R multiplexers. This field selects the input to be provided by the S multiplexer: register file, DB bus, or MQ register. The source register is defined by field 25.
- 23. R Input.** Selects input to be provided by the R multiplexer: register file or DA bus.
- 24. Destination Register.** Address of register in register file to be used for the destination operand.
- 25. Source Register.** Address of register in register file to be used for the source operand, provided by the S multiplexer.
- 26. Source Register.** Address of register in register file to be used for the source operand, provided by the R multiplexer.

Finally, field 27 is an 8-bit opcode that specifies the arithmetic or logical function to be performed by the ALU. Table 21.9 lists the different operations that can be performed.

As an example of the coding used to specify fields 17 through 27, consider the instruction to add the contents of register 1 to register 2 and place the result in register 3. The symbolic instruction is

CONT11 [17], WELH, SELRFYMX, [24], R3, R2, R1, PASS + ADD  
The assembler will translate this into the appropriate bit pattern. The individual components of the instruction can be described as follows:

- CONT11 is the basic NOP instruction.
- Field [17] is changed to WELH (write enable, low and high), so that a 32-bit register is written into.
- Field [18] is changed to SELRFYMX to select the feedback from the ALU Y MUX output.
- Field [24] is changed to designate register R3 for the destination register.
- Field [25] is changed to designate register R2 for one of the source registers.
- Field [26] is changed to designate register R1 for one of the source registers.
- Field [27] is changed to specify an ALU operation of ADD. The ALU shifter instruction is PASS; therefore, the ALU output is not shifted by the shifter.

Several points can be made about the symbolic notation. It is not necessary to specify the field number for consecutive fields. That is,

CONT11 [17], WELH, [18], SELRFYMX

can be written as

CONT11 [17], WELH, SELRFYMX

because SELRFYMX is in field 18.

ALU instructions from Group 1 of Table 21.9 must always be used in conjunction with Group 2. ALU instructions from Groups 3 to 5 must not be used with Group 2.

**Table 21.9** TI 8832 Registered ALU Instruction Field (Field 27)

<b>Group 1</b>		<b>Function</b>
ADD	H#01	R + S + Cn
SUBR	H#02	(NOT R) + S + Cn
SUBS	H#03	R = (NOT S) + Cn
INSC	H#04	S + Cn
INCNS	H#05	(NOT S) + Cn
INCR	H#06	R + Cn
INCNR	H#07	(NOT R) + Cn
XOR	H#09	R XOR S
AND	H#0A	R AND S
OR	H#0B	R OR S
NAND	H#0C	R NAND S
NOR	H#0D	R NOR S
ANDNR	H#0E	(NOT R) AND S
<b>Group 2</b>		<b>Function</b>
SRA	H#00	Arithmetic right single precision shift
SRAD	H#10	Arithmetic right double precision shift
SRL	H#20	Logical right single precision shift
SRLD	H#30	Logical right double precision shift
SLA	H#40	Arithmetic left single precision shift
SLAD	H#50	Arithmetic left double precision shift
SLC	H#60	Circular left single precision shift
SLCD	H#70	Circular left double precision shift
SRC	H#80	Circular right single precision shift
SRCD	H#90	Circular right double precision shift
MQSRA	H#A0	Arithmetic right shift MQ register
MQSRL	H#B0	Logical right shift MQ register
MQSLL	H#C0	Logical left shift MQ register
MQSLC	H#D0	Circular left shift MQ register
LOADMQ	H#E0	Load MQ register
PASS	H#F0	Pass ALU to Y (no shift operation)
<b>Group 3</b>		<b>Function</b>
SET1	H#08	Set bit 1
Set0	H#18	Set bit 0
TB1	H#28	Test bit 1
TB0	H#38	Test bit 0
ABS	H#48	Absolute value
SMTc	H#58	Sign magnitude/twos-complement

<b>Group 3</b>		<b>Function</b>
ADDI	H#68	Add immediate
SUBI	H#78	Subtract immediate
BADD	H#88	Byte add R to S
BSUBS	H#98	Byte subtract S from R
BSUBR	H#A8	Byte subtract R from S
BINCS	H#B8	Byte increment S
BINCNS	H#C8	Byte increment negative S
BXOR	H#D8	Byte XOR R and S
BAND	H#E8	Byte AND R and S
BOR	H#F8	Byte OR R and S
<b>Group 4</b>		<b>Function</b>
CRC	H#00	Cyclic redundancy character accum.
SEL	H#10	Select S or R
SNORM	H#20	Single length normalize
DNORM	H#30	Double length normalize
DIVRF	H#40	Divide remainder fix
SDIVQF	H#50	Signed divide quotient fix
SMULI	H#60	Signed multiply iterate
SMULT	H#70	Signed multiply terminate
SDIVIN	H#80	Signed divide initialize
SDIVIS	H#90	Signed divide start
SDIVI	H#A0	Signed divide iterate
UDIVIS	H#B0	Unsigned divide start
UDIVI	H#C0	Unsigned divide iterate
UMULI	H#D0	Unsigned multiply iterate
SDIVIT	H#E0	Signed divide terminate
UDIVIT	H#F0	Unsigned divide terminate
<b>Group 5</b>		<b>Function</b>
LOADFF	H#0F	Load divide/BCD flip-flops
CLR	H#1F	Clear
DUMPFF	H#5F	Output divide/BCD flip-flops
BCDBIN	H#7F	BCD to binary
EX3BC	H#8F	Excess (3 byte correction)
EX3C	H#9F	Excess (3 word correction)
SDIVO	H#AF	Signed divide overflow test
BINEX3	H#DF	Binary to excess – 3
NOP32	H#FF	No operation

## 21.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

### Key Terms

control memory control word firmware hard microprogramming horizontal microinstruction	microinstruction encoding microinstruction execution microinstruction sequencing microinstructions microprogram	microprogrammed control unit microprogramming language soft microprogramming unpacked microinstruction vertical microinstruction
--	---	--

### Review Questions

- 21.1** What is the difference between a hardwired implementation and a microprogrammed implementation of a control unit?
- 21.2** How is a horizontal microinstruction interpreted?
- 21.3** What is the purpose of a control memory?
- 21.4** What is a typical sequence in the execution of a horizontal microinstruction?
- 21.5** What is the difference between horizontal and vertical microinstructions?
- 21.6** What are the basic tasks performed by a microprogrammed control unit?
- 21.7** What is the difference between packed and unpacked microinstructions?
- 21.8** What is the difference between hard and soft microprogramming?
- 21.9** What is the difference between functional and resource encoding?
- 21.10** List some common applications of microprogramming.

### Problems

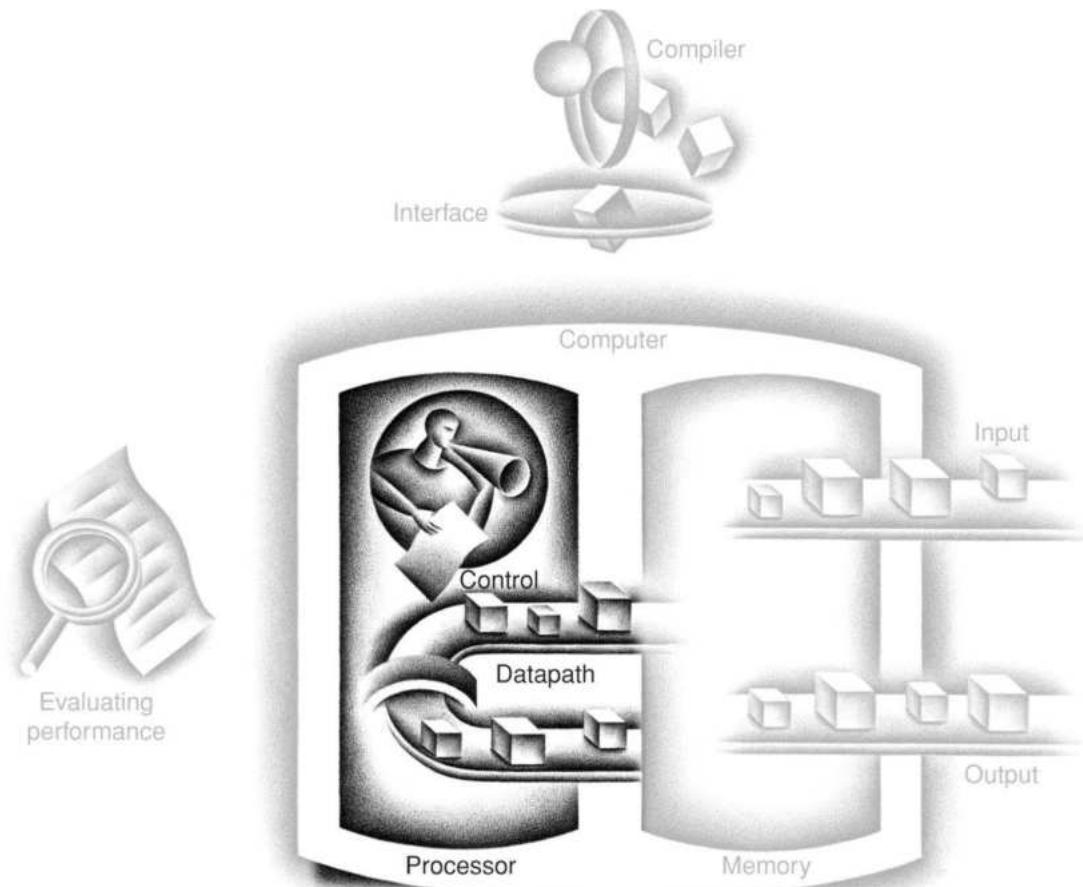
- 21.1** Describe the implementation of the multiply instruction in the hypothetical machine designed by Wilkes. Use narrative and a flowchart.
- 21.2** Assume a microinstruction set that includes a microinstruction with the following symbolic form:

IF ( $AC_0 = 1$ ) THEN  $CAR \leftarrow (C_{0-6})$  ELSE  $CAR \leftarrow (CAR) + 1$

where  $AC_0$  is the sign bit of the accumulator and  $C_{0-6}$  are the first seven bits of the microinstruction. Using this microinstruction, write a microprogram that implements a Branch Register Minus (BRM) machine instruction, which branches if the AC is negative. Assume that bits  $C_1$  through  $C_n$  of the microinstruction specify a parallel set of micro-operations. Express the program symbolically.

- 21.3** A simple processor has four major phases to its instruction cycle: fetch, indirect, execute, and interrupt. Two 1-bit flags designate the current phase in a hardwired implementation.
  - a. Why are these flags needed?
  - b. Why are they not needed in a microprogrammed control unit?
- 21.4** Consider the control unit of Figure 21.7. Assume that the control memory is 24 bits wide. The control portion of the microinstruction format is divided into two fields. A micro-operation field of 13 bits specifies the micro-operations to be performed. An address selection field specifies a condition, based on the flags, that will cause a microinstruction branch. There are eight flags.

- a. How many bits are in the address selection field?
  - b. How many bits are in the address field?
  - c. What is the size of the control memory?
- 21.5** How can unconditional branching be done under the circumstances of the previous problem? How can branching be avoided; that is, describe a microinstruction that does not specify any branch, conditional or unconditional.
- 21.6** We wish to provide 8 control words for each machine instruction routine. Machine instruction opcodes have 5 bits, and control memory has 1024 words. Suggest a mapping from the instruction register to the control address register.
- 21.7** An encoded microinstruction format is to be used. Show how a 9-bit micro-operation field can be divided into subfields to specify 46 different actions.
- 21.8** A processor has 16 registers, an ALU with 16 logic and 16 arithmetic functions, and a shifter with 8 operations, all connected by an internal processor bus. Design a microinstruction format to specify the various micro-operations for the processor.



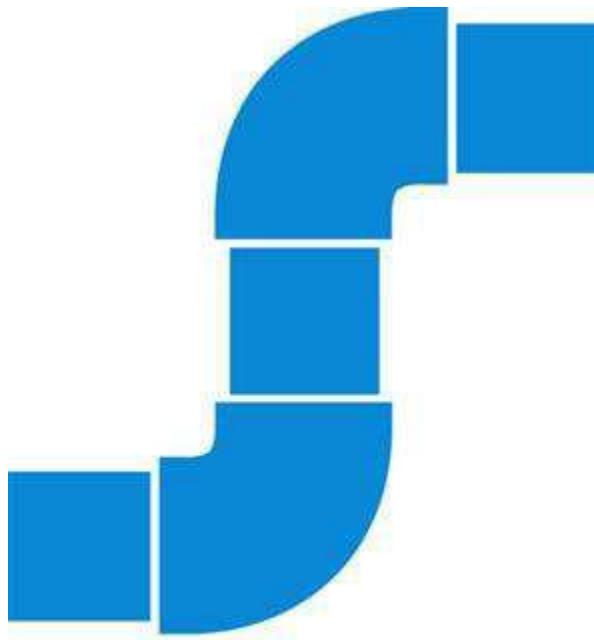
### The Five Classic Components of a Computer

## 4.1 Introduction

[Chapter 1](#) explains that the performance of a computer is determined by three key factors: instruction count, clock cycle time, and *clock cycles per instruction* (CPI). [Chapter 2](#) explains that the compiler and the instruction set architecture determine the instruction count required for a given program. However, the implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction. In this chapter, we construct the datapath and control unit for two different implementations of the RISC-V instruction set.

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section. It is followed by a section that builds up a datapath and constructs a simple version of

a processor sufficient to implement an instruction set like RISC-V. The bulk of the chapter covers a more realistic **pipelined** RISC-V implementation, followed by a section that develops the concepts necessary to implement more complex instruction sets, like the x86.



## PIPELINING

For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section and [Section 4.5](#) present the basic concepts of pipelining. Current trends are covered in [Section 4.10](#), and [Section 4.11](#) describes the recent Intel Core i7 and ARM Cortex-A53 architectures. [Section 4.12](#) shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from [Section 3.9](#). These sections provide enough background to understand the pipeline concepts at a high level.

For the reader interested in understanding the processor and its performance in more depth, [Sections 4.3, 4.4](#), and [4.6](#) will be useful. Those interested in learning how to build a processor should also cover [Sections 4.2, 4.7–4.9](#). For readers with an interest in modern

hardware design,  [Section 4.13](#) describes how hardware design languages and CAD tools are used to implement hardware, and

then how to use a hardware design language to describe a pipelined implementation. It also gives several more illustrations of how pipelining hardware executes.

## A Basic RISC-V Implementation

We will be examining an implementation that includes a subset of the core RISC-V instruction set:

- The memory-reference instructions *load doubleword* (`ld`) and *store doubleword* (`sd`)
- The arithmetic-logical instructions `add`, `sub`, `and`, `or`
- The conditional branch instruction *branch if equal* (`beq`)

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. However, it illustrates the key principles used in creating a datapath and designing the control. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer. Many of the key design principles introduced in [Chapter 1](#) can be illustrated by looking at the implementation, such as *Simplicity favors regularity*. In addition, most concepts used to implement the RISC-V subset in this chapter are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors.

## An Overview of the Implementation

In [Chapter 2](#), we looked at the core RISC-V instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the

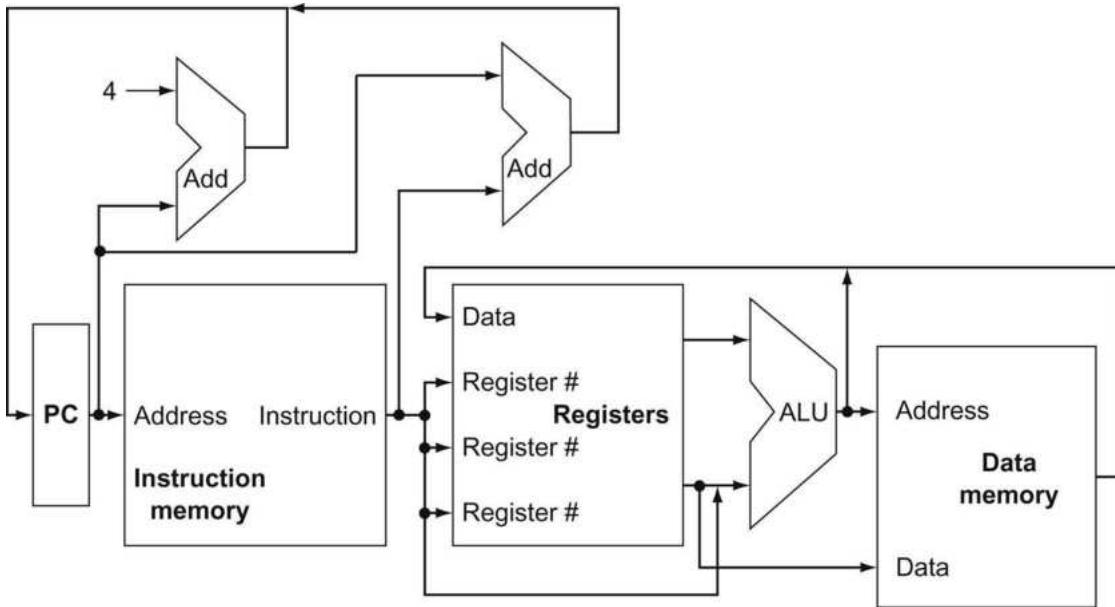
code and fetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registers to read. For the `ld` instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the RISC-V instruction set simplify the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and conditional branches for the equality test. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.

[Figure 4.1](#) shows the high-level view of a RISC-V implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.



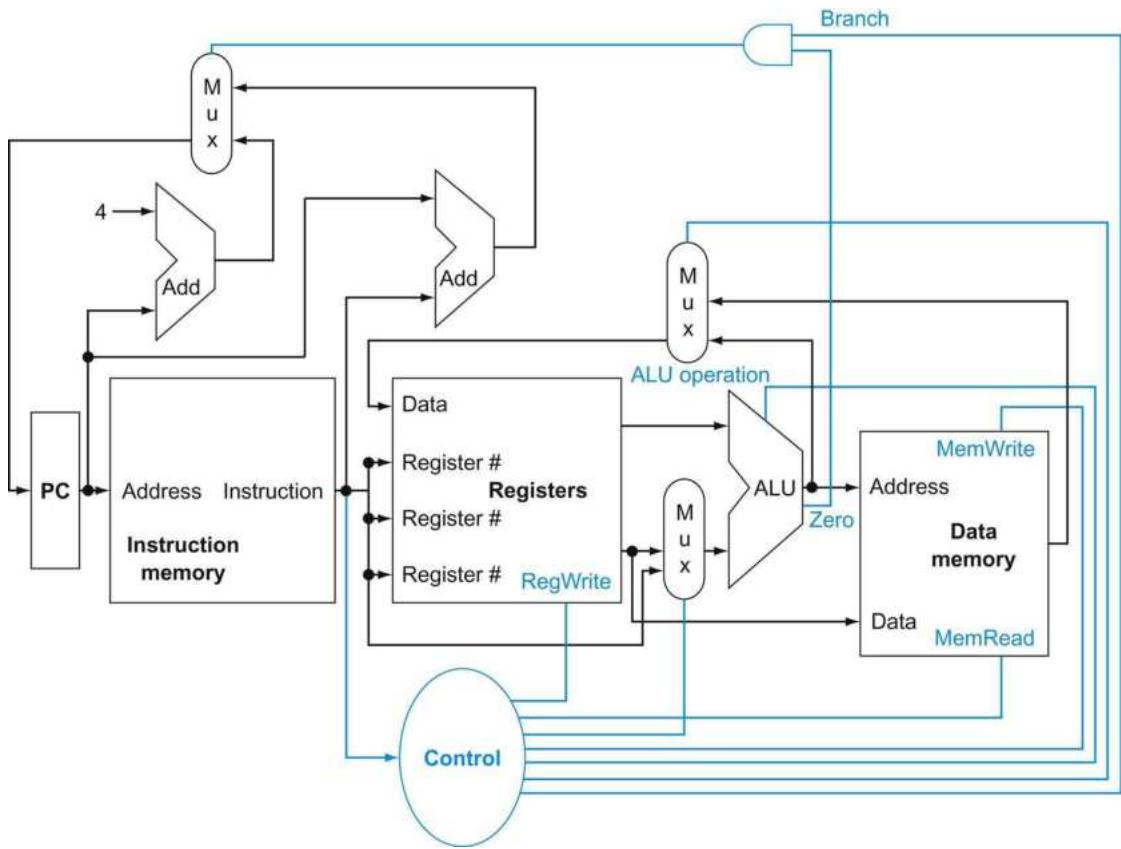
**FIGURE 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them.**

All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

First, in several places, [Figure 4.1](#) shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction. In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. [Appendix A](#) describes the multiplexor, which selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

The second omission in [Figure 4.1](#) is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The register file must be written only on a load or an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. ([Appendix A](#) describes the detailed design of the ALU.) Like the multiplexors, control lines that are set based on various fields in the instruction direct these operations.

[Figure 4.2](#) shows the datapath of [Figure 4.1](#) with the three required multiplexors added, as well as control lines for the major functional units. A *control unit*, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The top multiplexor, which determines whether PC +4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a `bneq` instruction. The regularity and simplicity of the RISC-V instruction set mean that a simple decoding process can be used to determine how to set the control lines.



**FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.**

The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottom-most multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, enhance a control unit to control what actions are taken for different instruction classes. [Sections 4.3](#) and [4.4](#) describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of [Figures 4.1](#) and [4.2](#). In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since the clock cycle must be severely stretched to accommodate the longest instruction. After designing the control for this simple computer, we will look at pipelined implementation with all its complexities, including exceptions.

### Check Yourself

How many of the five classic components of a computer—shown on page 235—do [Figures 4.1](#) and [4.2](#) include?

## 4.2 Logic Design Conventions

To discuss the design of a computer, we must decide how the hardware logic implementing the computer will operate and how the computer is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this chapter. If you have little or no background in digital logic, you will find it helpful to read [Appendix A](#) before continuing.

The datapath elements in the RISC-V implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in [Figure 4.1](#) and discussed in [Appendix A](#) is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

### combinational element

An operational element, such as an AND gate or an ALU

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the computer had never lost power. Thus, these state elements completely characterize the computer. In [Figure 4.1](#), the instruction and data memories, as well as the registers, are all examples of state elements.

## state element

A memory element, such as a register or a memory.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see [Appendix A](#)), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our RISC-V implementation uses two other types of state elements: memories and registers, both of which appear in [Figure 4.1](#). The clock is used to determine when the state element should be written; a state element can be read at any time.

Logic components that contain state are also called *sequential*, because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously. [Appendix A](#) discusses the operation of both the combinational and sequential elements and their construction in more detail.

# Clocking Methodology

## clocking methodology

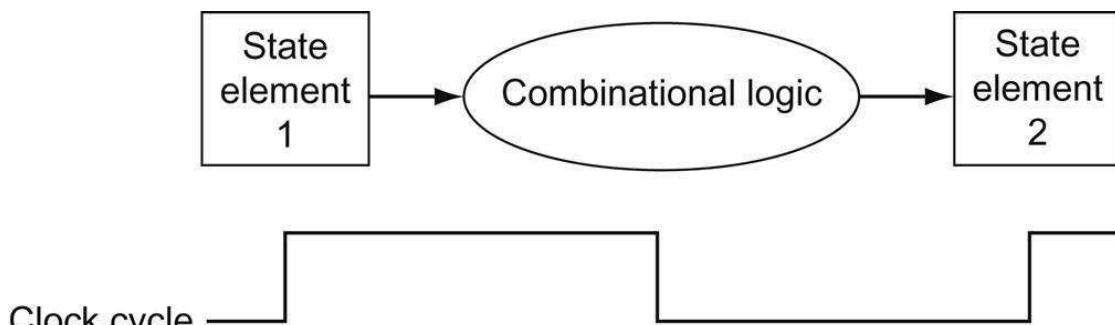
The approach used to determine when data are valid and stable relative to the clock.

A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written at the same time that it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Computer designs cannot tolerate such unpredictability. A clocking methodology is designed to make hardware predictable.

## edge-triggered clocking

A clocking scheme in which all state changes occur on a clock edge.

For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa (see [Figure 4.3](#)). Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.



**FIGURE 4.3** Combinational logic, state elements, and the clock are closely related.

In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge.

Figure 4.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

## control signal

A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a *data signal*, which contains information that is operated on by a functional unit.

For simplicity, we do not show a write **control signal** when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

We will use the word **asserted** to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and *deassert* or **deasserted** to represent logically low. We use the terms assert and deassert because when we implement hardware, at times 1 represents logically high and at times it can represent logically low.

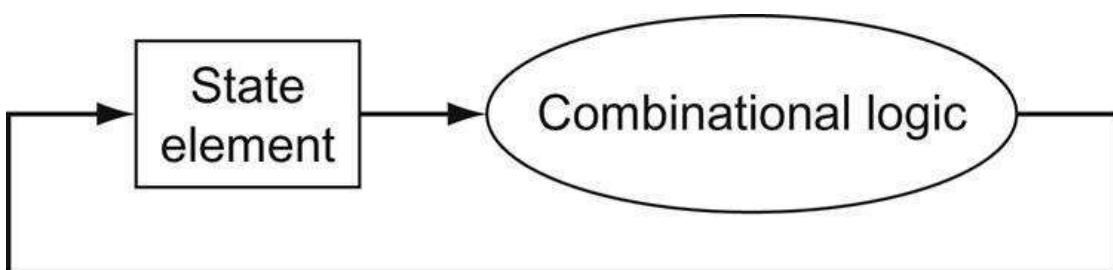
## asserted

The signal is logically high or true.

## deasserted

The signal is logically low or false.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle. [Figure 4.4](#) gives a generic example. It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge. In this book, we use the rising clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in [Figure 4.4](#) works correctly. In [Appendix A](#), we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.



**FIGURE 4.4** An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.

Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

For the 64-bit RISC-V architecture, nearly all of these state and logic elements will have inputs and outputs that are 64 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 64 bits in width. The figures will indicate *buses*, which

are signals wider than 1 bit, with thicker lines. At times, we will want to combine several buses to form a wider bus; for example, we may want to obtain a 64-bit bus by combining two 32-bit buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, **color** indicates a control signal contrary to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

### Check Yourself

True or false: Because the register file is both read and written on the same clock cycle, any RISC-V datapath using edge-triggered writes must have more than one copy of the register file.

### Elaboration

There is also a 32-bit version of the RISC-V architecture, and, naturally enough, most paths in its implementation would be 32 bits wide.

## 4.3 Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of RISC-V instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.



## A B S T R A C T I O N

### datapath element

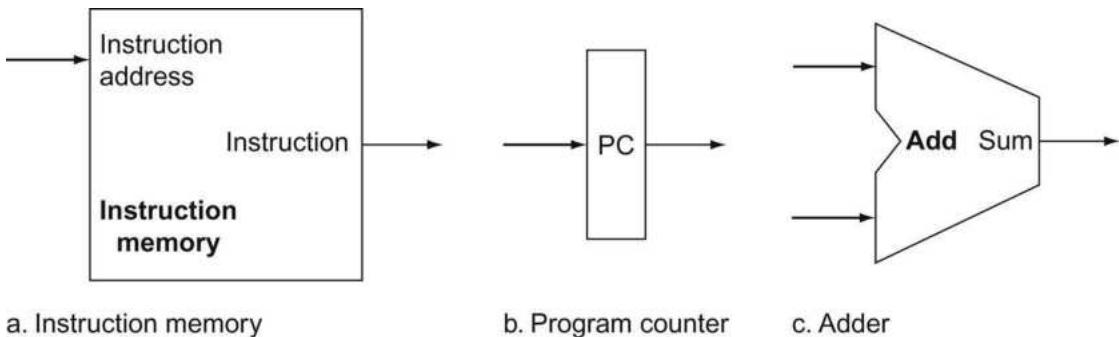
A unit used to operate on or hold data within a processor. In the RISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

### program counter (PC)

The register containing the address of the instruction in the program being executed.

Figure 4.5a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 4.5b also shows the **program counter (PC)**, which as we saw in Chapter 2 is a register that holds the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU described in detail in Appendix A simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with

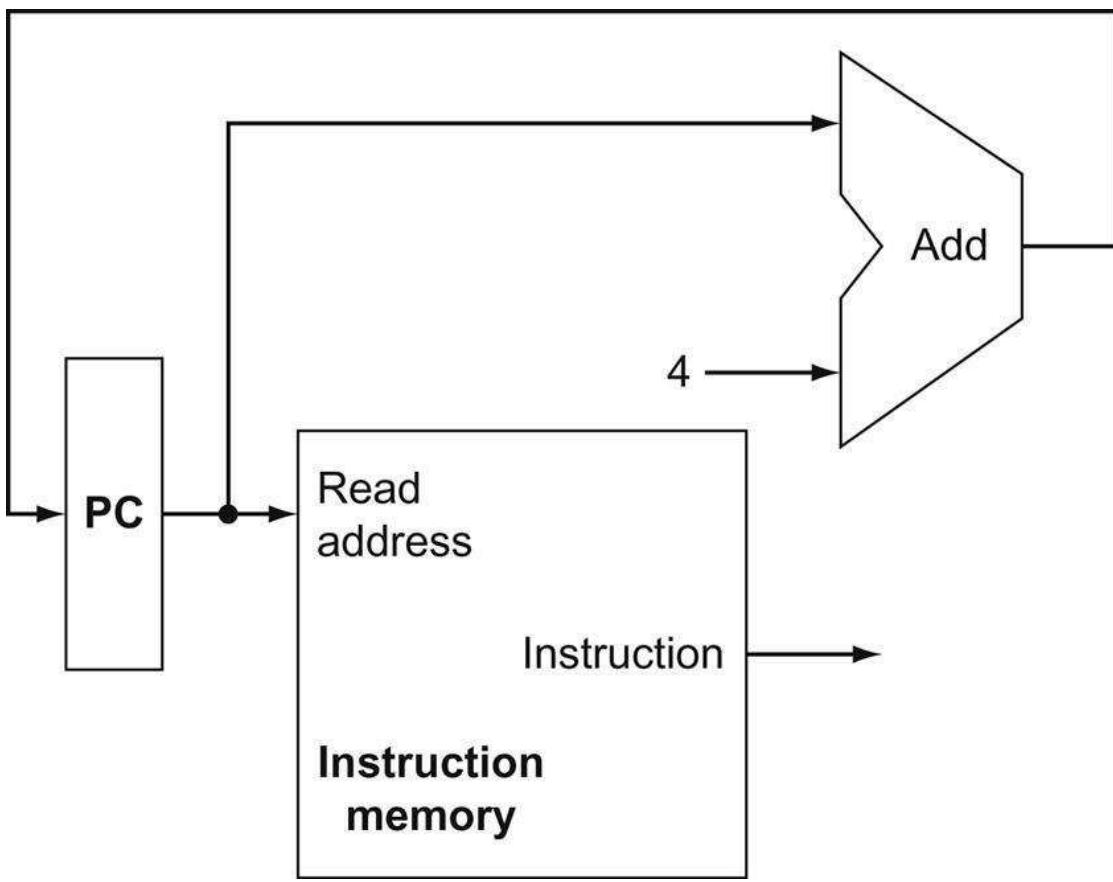
the label *Add*, as in [Figure 4.5c](#), to indicate that it has been permanently made an adder and cannot perform the other ALU functions.



**FIGURE 4.5** Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 64-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 64-bit inputs and place the sum on its output.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. [Figure 4.6](#) shows how to combine the three elements from [Figure 4.5](#) to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.



**FIGURE 4.6** A portion of the datapath used for fetching instructions and incrementing the program counter.

The fetched instruction is used by other parts of the datapath.

Now let's consider the R-format instructions (see [Figure 2.19](#) on page 120). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `add`, `sub`, `and`, and `or`, which were introduced in [Chapter 2](#). Recall that a typical instance of such an instruction is `add x1, x2, x3`, which reads `x2` and `x3` and writes the sum into `x1`.

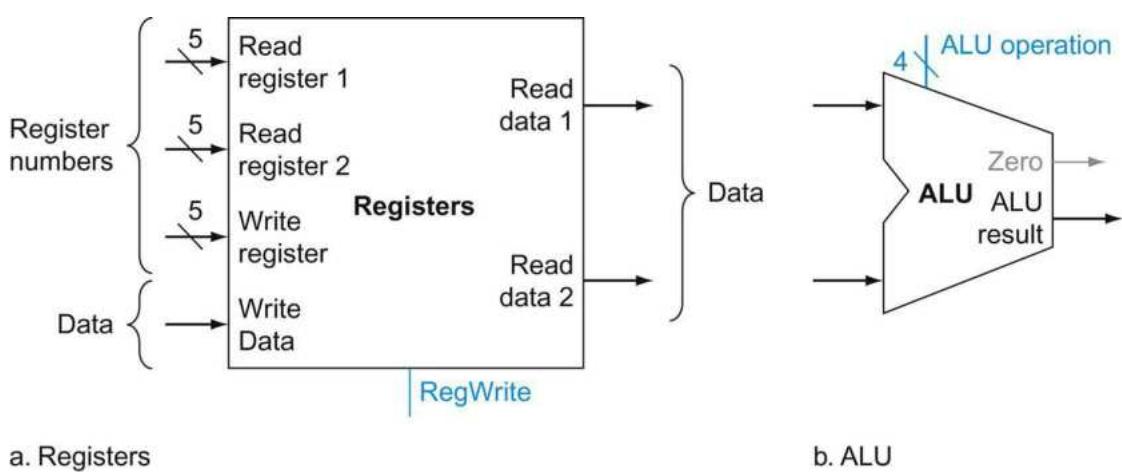
## register file

A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

The processor's 32 general-purpose registers are stored in a

structure called a **register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. [Figure 4.7a](#) shows the result; we need a total of three inputs (two for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ( $32 = 2^5$ ), whereas the data input and two data output buses are each 64 bits wide.



**FIGURE 4.7** The two elements needed to implement R-format ALU operations are the register file and the ALU.

The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in [Section A.8](#) of

[Appendix A](#). The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 64 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in [Appendix A](#). We will use the Zero detection output of the ALU shortly to implement conditional branches.

[Figure 4.7b](#) shows the ALU, which takes two 64-bit inputs and produces a 64-bit result, as well as a 1-bit signal if the result is 0. The 4-bit control signal of the ALU is described in detail in [Appendix A](#); we will review the ALU control shortly when we need to know how to set it.

Next, consider the RISC-V load register and store register instructions, which have the general form `ld x1, offset(x2)` or `sd x1, offset(x2)`. These instructions compute a memory address by adding the base register, which is  $x_2$ , to the 12-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in  $x_1$ . If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is  $x_1$ . Thus, we will need both the register file and the ALU from [Figure 4.7](#).

## sign-extend

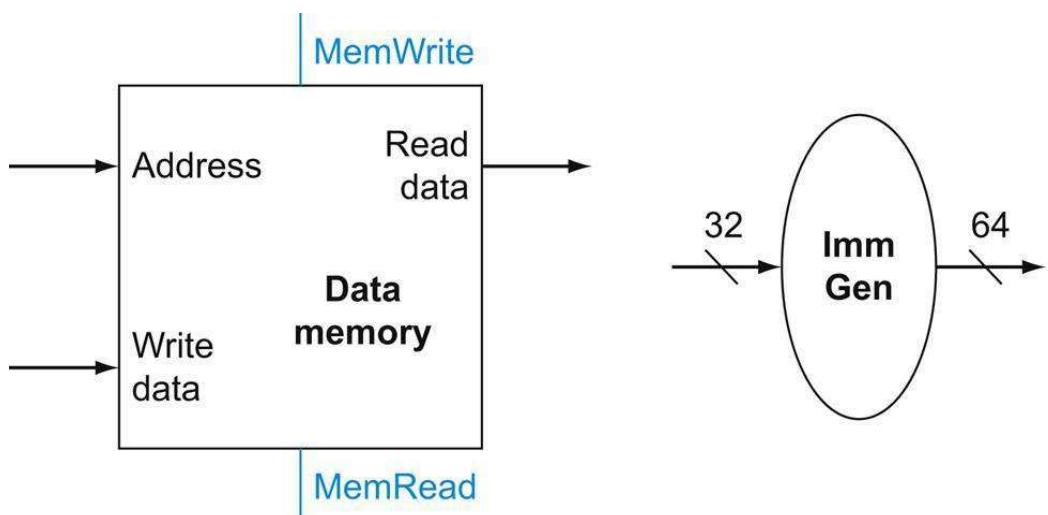
To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger,

destination data item.

## branch target address

The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch.

In addition, we will need a unit to **sign-extend** the 12-bit offset field in the instruction to a 64-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory. [Figure 4.8](#) shows these two elements.



a. Data memory unit

b. Immediate generation unit

**FIGURE 4.8** The two units needed to implement loads and stores, in addition to the register file and ALU of [Figure 4.7](#), are the data memory unit and the immediate generation unit.

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in [Chapter 5](#). The immediate generation unit

(ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 64-bit result appearing on the output (see [Chapter 2](#)). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See [Section A.8 of Appendix A](#) for further discussion of how real memory chips work.

The `beq` instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq x1, x2, offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see [Chapter 2](#)) to which we must pay attention:

- The instruction set architecture specifies that the base for the branch address calculation is the address of the branch instruction.
- The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2.

To deal with the latter complication, we will need to shift the offset field by 1.

As well as computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., two operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operand is not zero, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch** is **not taken**.

## branch taken

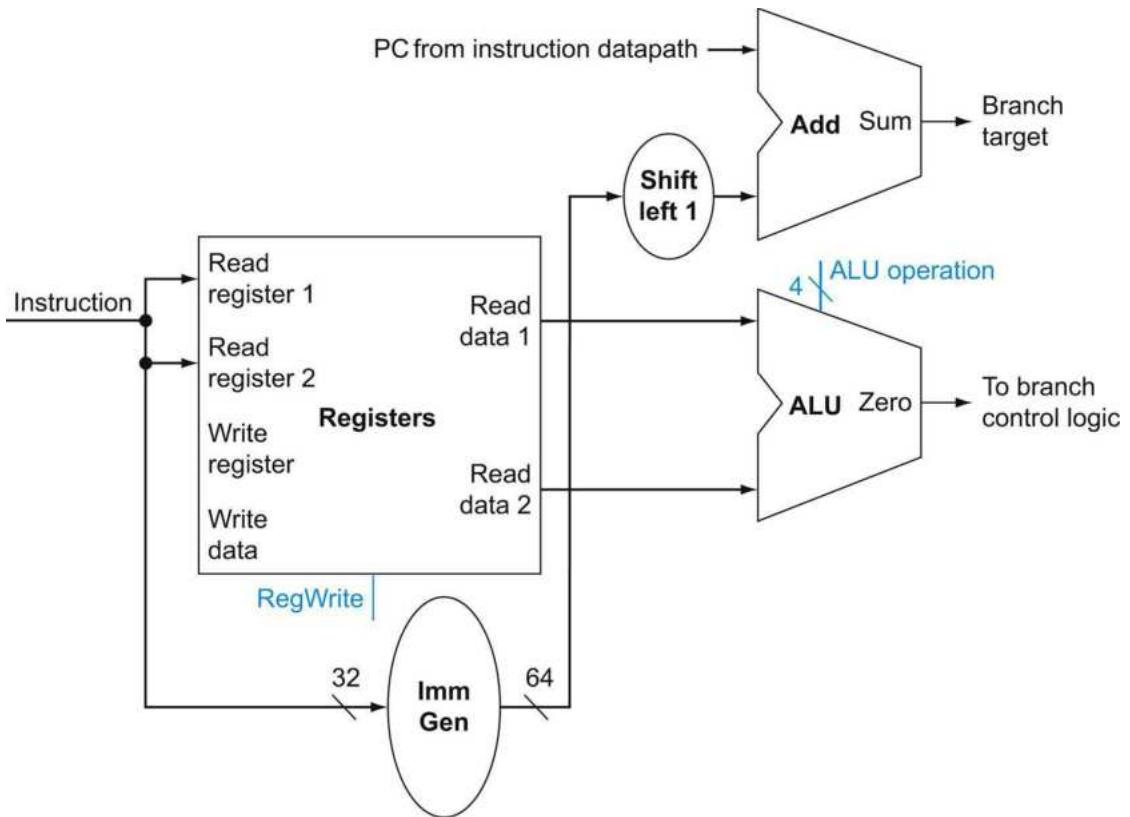
A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional

branches are taken branches.

## branch not taken or (untaken branch)

A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

Thus, the branch datapath must do two operations: compute the branch target address and test the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) [Figure 4.9](#) shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes an immediate generation unit, from [Figure 4.8](#) and an adder. To perform the compare, we need to use the register file shown in [Figure 4.7a](#) to supply two register operands (although we will not need to write into the register file). In addition, the equality comparison can be done using the ALU we designed in [Appendix A](#). Since that ALU provides an output signal that indicates whether the result was 0, we can send both register operands to the ALU with the control set to subtract two values. If the Zero signal out of the ALU unit is asserted, we know that the register values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equality test of conditional branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.



**FIGURE 4.9** The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and the sign-extended 12 bits of the instruction (the branch displacement), shifted left 1 bit.

The unit labeled *Shift left 1* is simply a routing of the signals between input and output that adds  $0_{\text{two}}$  to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the “shift” is constant. Since we know that the offset was sign-extended from 12 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

The branch instruction operates by adding the PC with the 12 bits of the instruction shifted left by 1 bit. Simply concatenating 0 to the branch offset accomplishes this shift, as described in [Chapter 2](#).

## Creating a Single Datapath

Now that we have examined the datapath components needed for

the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. This simplest datapath will attempt to execute all instructions in one clock cycle. This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

## Building a Datapath

### Example

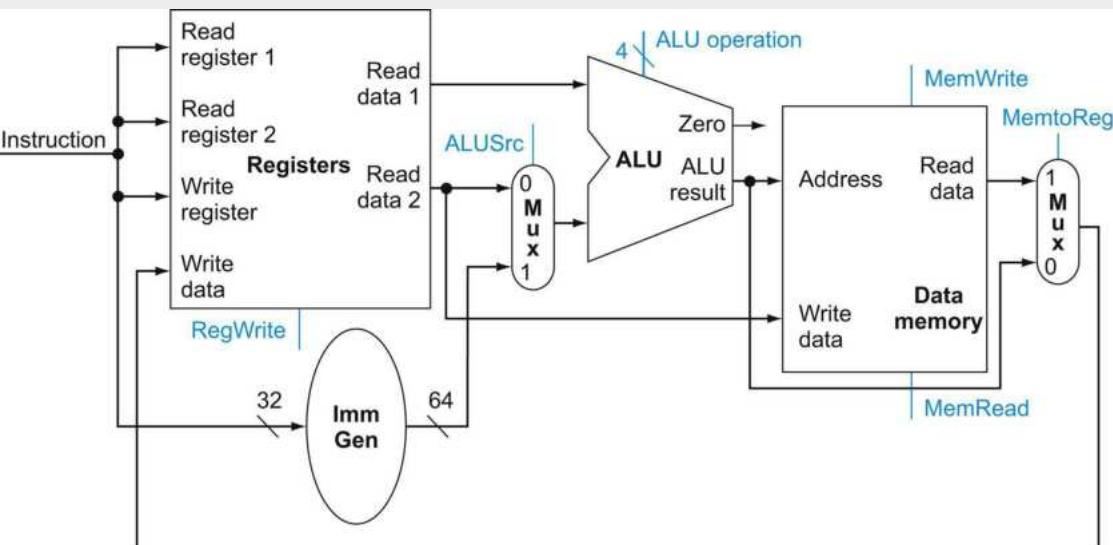
The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 12-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

### Answer

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. [Figure 4.10](#) shows the operational portion of the combined datapath.



**FIGURE 4.10** The datapath for the memory instructions and the R-type instructions.

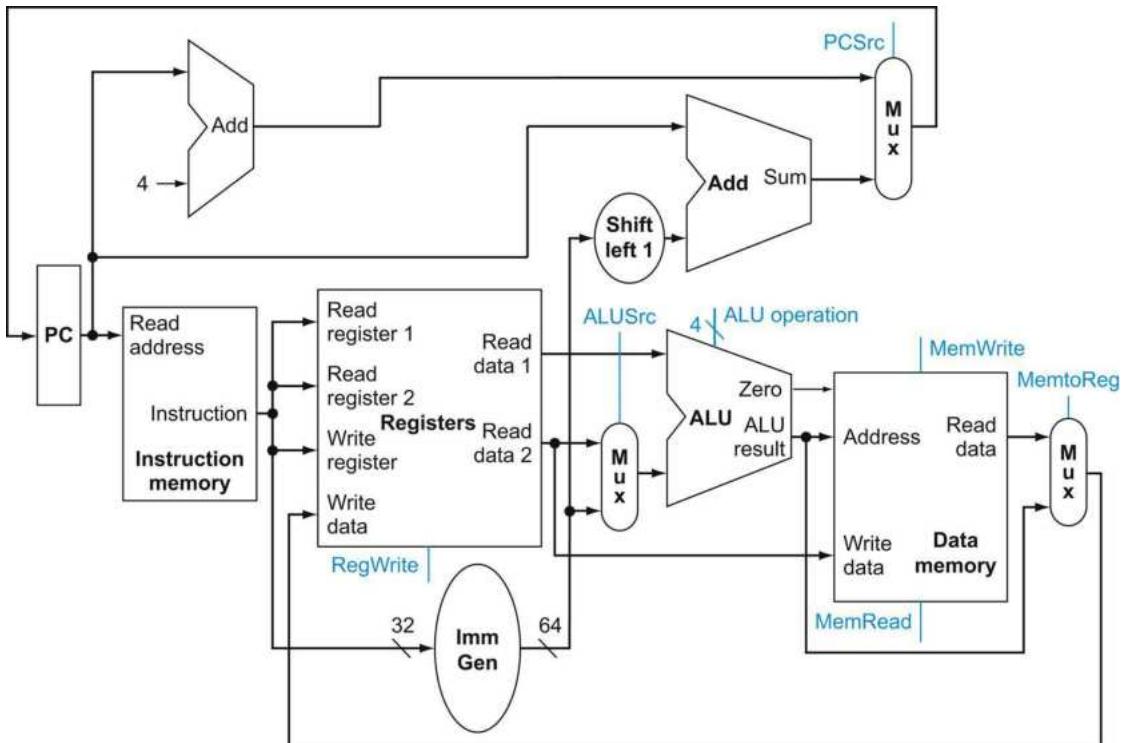
This example shows how a single datapath can be assembled from the pieces in [Figures 4.7](#) and [4.8](#) by adding multiplexors. Two multiplexors are needed, as described in the example.

Now we can combine all the pieces to make a simple datapath for the core RISC-V architecture by adding the datapath for instruction fetch ([Figure 4.6](#)), the datapath from R-type and memory instructions ([Figure 4.10](#)), and the datapath for branches ([Figure 4.9](#)). [Figure 4.11](#) shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU to compare two register operands for equality, so we must keep the adder from [Figure 4.9](#) for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC +4) or the branch target address to be written into the PC.

## Check Yourself

- I. Which of the following is correct for a load instruction? Refer to [Figure 4.10](#).
- MemtoReg should be set to cause the data from memory to be sent to the register file.
  - MemtoReg should be set to cause the correct register destination to be sent to the register file.
  - We do not care about the setting of MemtoReg for loads.

- II. The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories, because
- a. the formats of data and instructions are different in RISC-V, and hence different memories are needed;
  - b. having separate memories is less expensive;
  - c. the processor operates in one cycle and cannot use a (single-ported) memory for two different accesses within that cycle.



**FIGURE 4.11** The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes.

The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

## Elaboration

The immediate generation logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for load instructions, bits 31:25 and 11:7 for store instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch. Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field. RISC-V opcode bit 6 happens to be 0 for

data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 happens to be 0 for load instructions and 1 for store instructions. Thus, bits 5 and 6 can control a 3:1 multiplexor inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

## 4.4 A Simple Implementation Scheme

In this section, we look at what might be thought of as a simple implementation of our RISC-V subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load doubleword* (`ld`), *store doubleword* (`sd`), *branch if equal* (`beq`), and the arithmetic-logical instructions `add`, `sub`, `and`, and `or`.

### The ALU Control

The RISC-V ALU in [Appendix A](#) defines the four following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the four actions (AND, OR, add, or subtract), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction (see [Chapter 2](#)). For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract and test if zero (01) for `beq`, or be determined by the operation encoded in the funct7 and funct3 fields (10). The output of the ALU control unit is a 4-bit signal that

directly controls the ALU by generating one of the 4-bit combinations shown previously.

In [Figure 4.12](#), we show how to set the ALU control inputs based on the 2-bit ALUOp control, funct7, and funct3 fields. Later in this chapter, we will see how the ALUOp bits are generated from the main control unit.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

**FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction.**

The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we “don’t care” about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See [Appendix A](#).

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially reduce the latency of the control unit. Such optimizations are

important, since the latency of the control unit is often a critical factor in determining the clock cycle time.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the funct fields to the four ALU operation control bits. Because only a small number of the possible funct field values are of interest and funct fields are used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and generates the appropriate ALU control signals.

As a step in designing this logic, it is useful to create a *truth table* for the interesting combinations of funct fields and the ALUOp signals, as we've done in [Figure 4.13](#); this **truth table** shows how the 4-bit ALU control is set depending on these input fields. Since the full truth table is very large, and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries for outputs that must be asserted and not showing those that are all deasserted or don't care. (This practice has a disadvantage, which we discuss in [Section C.2](#)

of  [Appendix C.](#))

## truth table

From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

ALUOp		Funct7 field								Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]		
0	0	X	X	X	X	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	X	X	X	X	0110	
1	X	0	0	0	0	0	0	0	0	0	0	0010	
1	X	0	1	0	0	0	0	0	0	0	0	0110	
1	X	0	0	0	0	0	0	0	1	1	1	0000	
1	X	0	0	0	0	0	0	0	1	1	0	0001	

**FIGURE 4.13** The truth table for the 4 ALU control bits (called Operation).

The inputs are the ALUOp and funct fields. Only the entries for which the ALU control is asserted are

shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

Because in many instances we do not care about the values of some of the inputs, and because we wish to keep the tables compact, we also include **don't-care terms**. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first row of [Figure 4.13](#), we always set the ALU control to 0010, independent of the funct fields. In this case, then, the funct inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see [Appendix A](#) for more information.

Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process



and the result in [Section C.2](#) of [Appendix C](#).

### don't-care term

An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

## Designing the Main Control Unit

Now that we have described how to design an ALU that uses the opcode and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in [Figure 4.11](#). To understand how to connect the fields of an instruction to the datapath, it is useful to

review the formats of the four instruction classes: arithmetic, load, store, and conditional branch instructions. [Figure 4.14](#) shows these formats.

Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0	Fields
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode	
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode	
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	

**FIGURE 4.14 The four instruction classes  
(arithmetic, load, store, and conditional branch)  
use four different instruction formats.**

- (a) Instruction format for R-type arithmetic instructions (opcode =  $51_{\text{ten}}$ ), which have three register operands: rs1, rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, and or.
- (b) Instruction format for I-type load instructions (opcode =  $3_{\text{ten}}$ ). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value.
- (c) Instruction format for S-type store instructions (opcode =  $35_{\text{ten}}$ ). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory.
- (d) Instruction format for SB-type conditional branch instructions (opcode =  $99_{\text{ten}}$ ). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address.

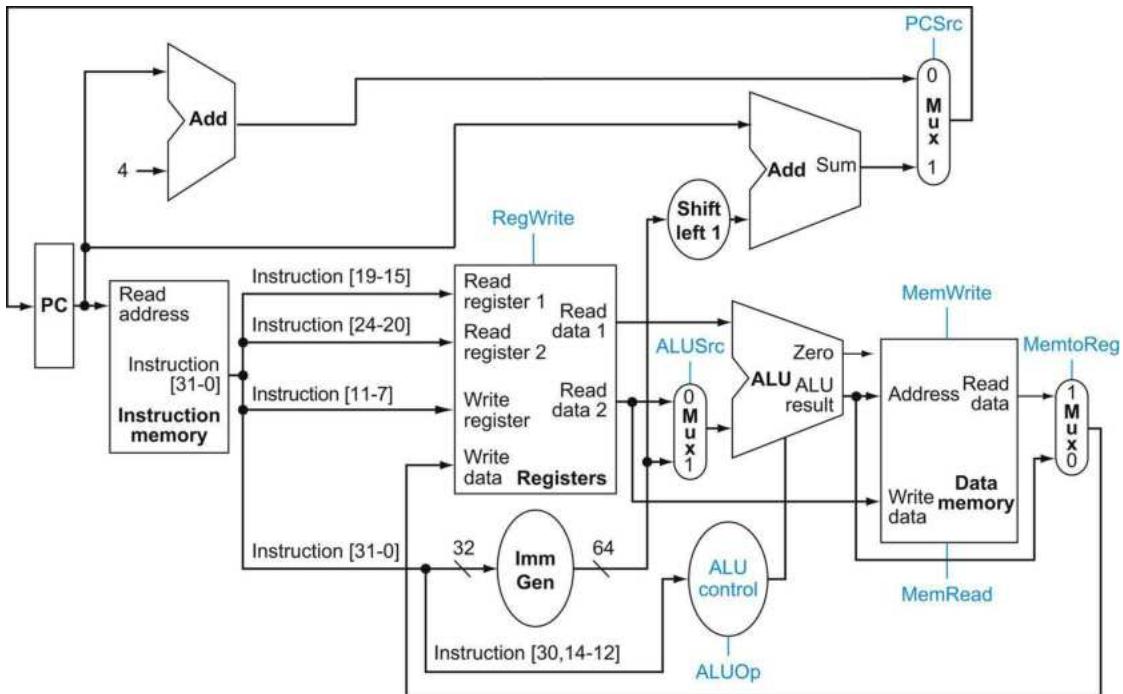
There are several major observations about this instruction format that we will rely on:

## opcode

The field that denotes the operation and format of an instruction.

- The **opcode** field, which as we saw in [Chapter 2](#), is always in bits 6:0. Depending on the opcode, the funct3 field (bits 14:12) and funct7 field (bits 31:25) serve as an extended opcode field.
- The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions. This field also specifies the base register for load and store instructions.
- The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions. This field also specifies the register operand that gets copied to memory for store instructions.
- Another operand can also be a 12-bit offset for branch or load-store instructions.
- The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instructions.  
The first design principle from [Chapter 2](#)—*simplicity favors regularity*—pays off here in specifying control.

Using this information, we can add the instruction labels to the simple datapath. [Figure 4.15](#) shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.



**FIGURE 4.15** The datapath of [Figure 4.11](#) with all necessary multiplexors and all control lines identified.

The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

[Figure 4.15](#) shows six single-bit control lines plus the 2-bit **ALUOp** control signal. We have already defined how the **ALUOp** control signal works, and it is useful to define what the six other control signals do informally before we determine how to set these control signals during instruction execution. [Figure 4.16](#) describes the function of these six control lines.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

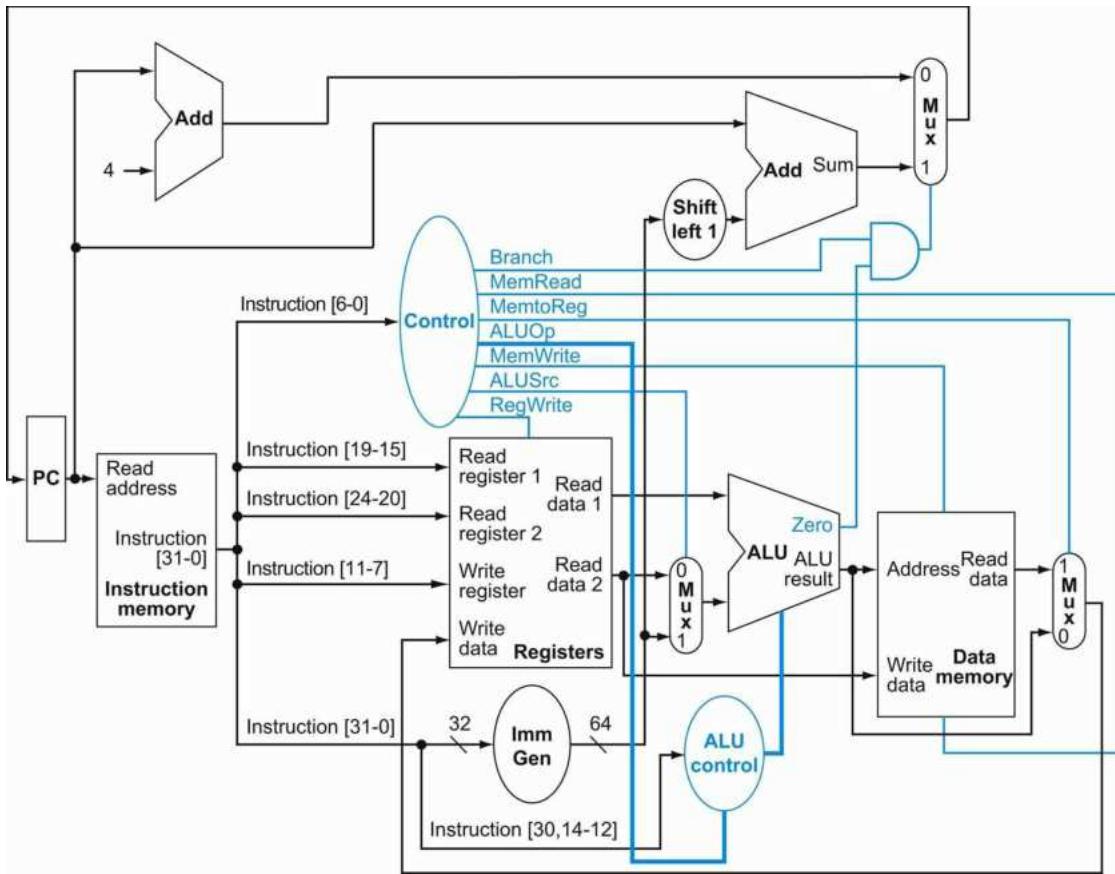
**FIGURE 4.16 The effect of each of the six control signals.**

When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input.

Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See [Appendix A](#) for further discussion of this problem.)

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode and funct fields of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch if equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for the equality test, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These eight control signals (six from [Figure 4.16](#) and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 6:0. [Figure 4.17](#) shows the datapath with the control unit and the control signals.



**FIGURE 4.17** The simple datapath with the control unit.

The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values. [Figure 4.18](#) defines

how the control signals should be set for each opcode; this information follows directly from [Figures 4.12, 4.16, and 4.17](#).

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

**FIGURE 4.18** The setting of the control lines is completely determined by the opcode fields of the instruction.

The first row of the table corresponds to the R-format instructions (`add`, `sub`, `and`, and `or`). For all these instructions, the source register fields are `rs1` and `rs2`, and the destination register field is `rd`; this defines how the signals `ALUSrc` is set. Furthermore, an R-type instruction writes a register (`RegWrite =1`), but neither reads nor writes data memory. When the `Branch` control signal is 0, the PC is unconditionally replaced with `PC +4`; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.

The `ALUOp` field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the `funct` fields. The second and third rows of this table give the control signal settings for `ld` and `sd`. These

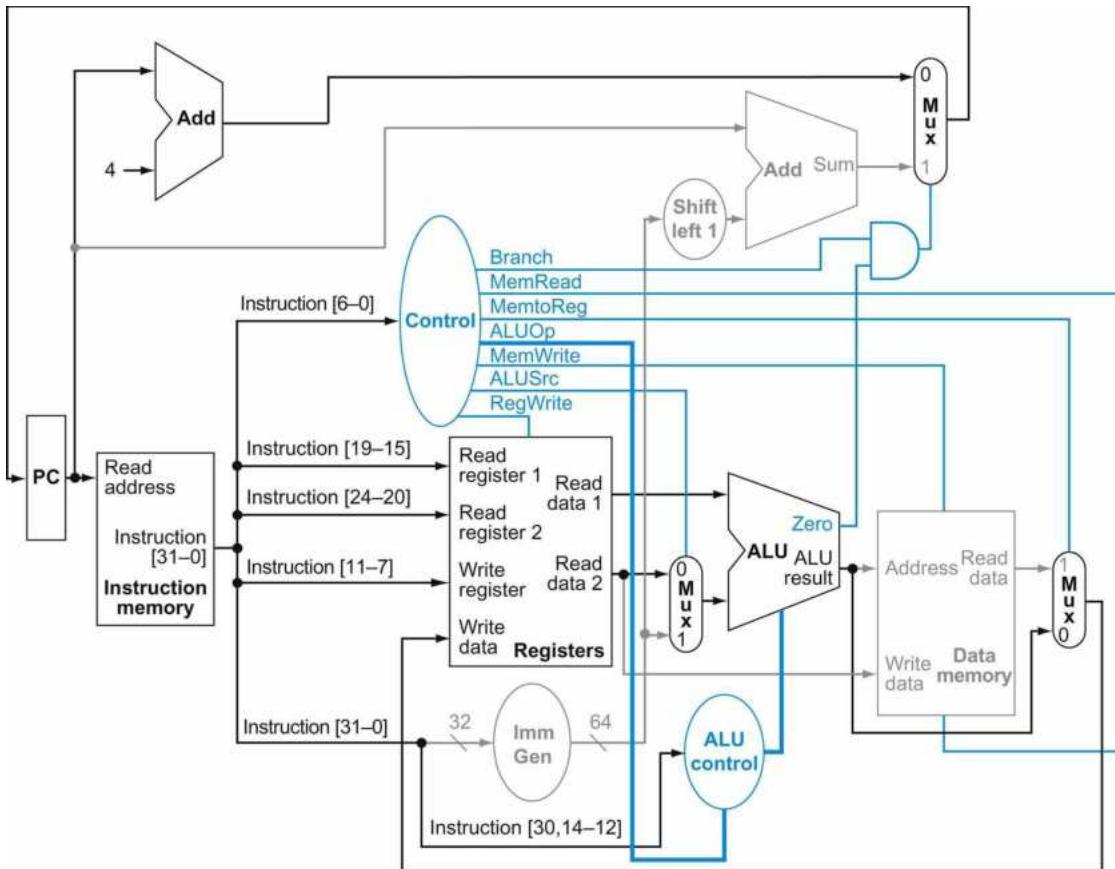
`ALUSrc` and `ALUOp` fields are set to perform the address calculation. The `MemRead` and `MemWrite` are set to perform the memory access. Finally, `RegWrite` is set for a load to cause the result to be stored in the `rd` register. The `ALUOp` field for branch is set for subtract (`ALU control =01`), which is used to test for equality. Notice that the `MemtoReg` field is irrelevant when the `RegWrite` signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry `MemtoReg` in the last two rows of the table is replaced with X for don't care. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

## Operation of the Datapath

With the information contained in [Figures 4.16](#) and [4.18](#), we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

[Figure 4.19](#) shows the operation of the datapath for an R-type instruction, such as `add x1, x2, x3`. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.
2. Two registers,  $x_2$  and  $x_3$ , are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register ( $x_1$ ) in the register file.



**FIGURE 4.19 The datapath in operation for an R-type instruction, such as `add x1, x2, x3`.**

The control lines, datapath units, and connections that are active are highlighted.

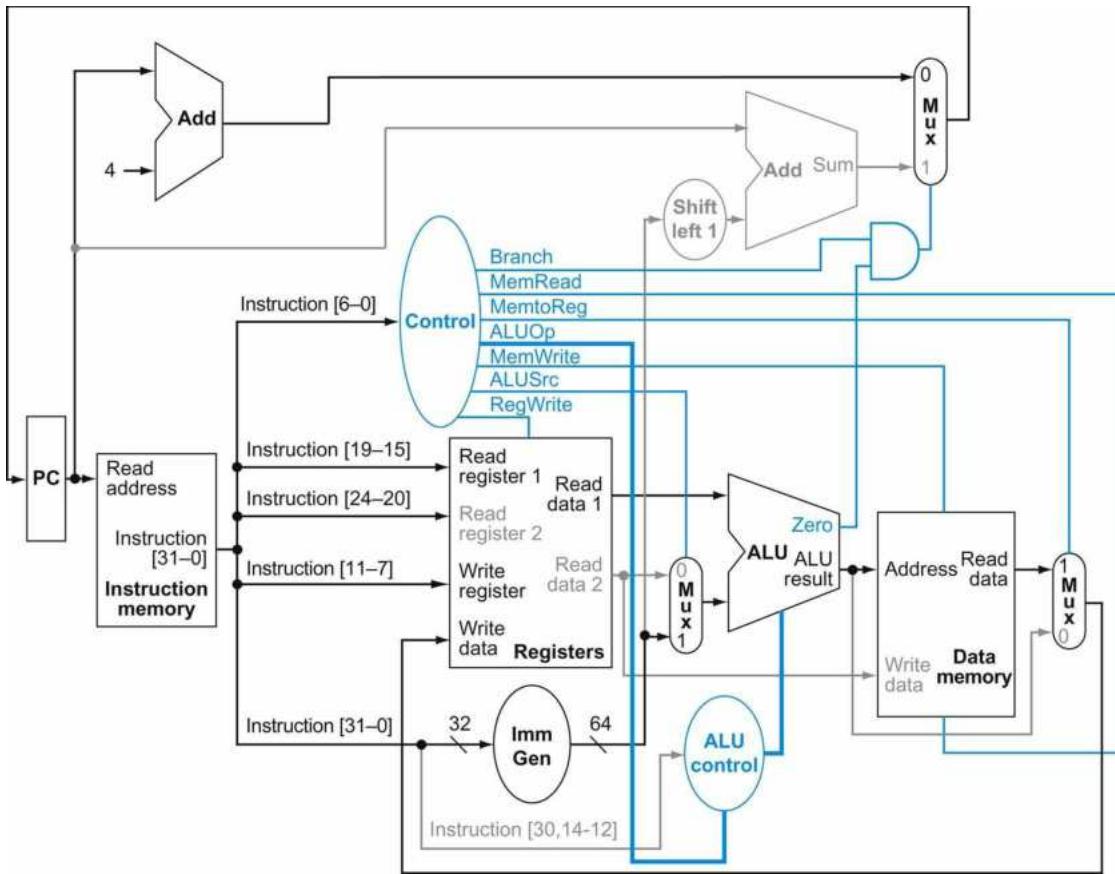
Similarly, we can illustrate the execution of a load register, such as

`ld x1, offset(x2)`

in a style similar to Figure 4.19. Figure 4.20 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ( $x_2$ ) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction ( $offset$ ).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file

(x1).



**FIGURE 4.20 The datapath in operation for a load instruction.**

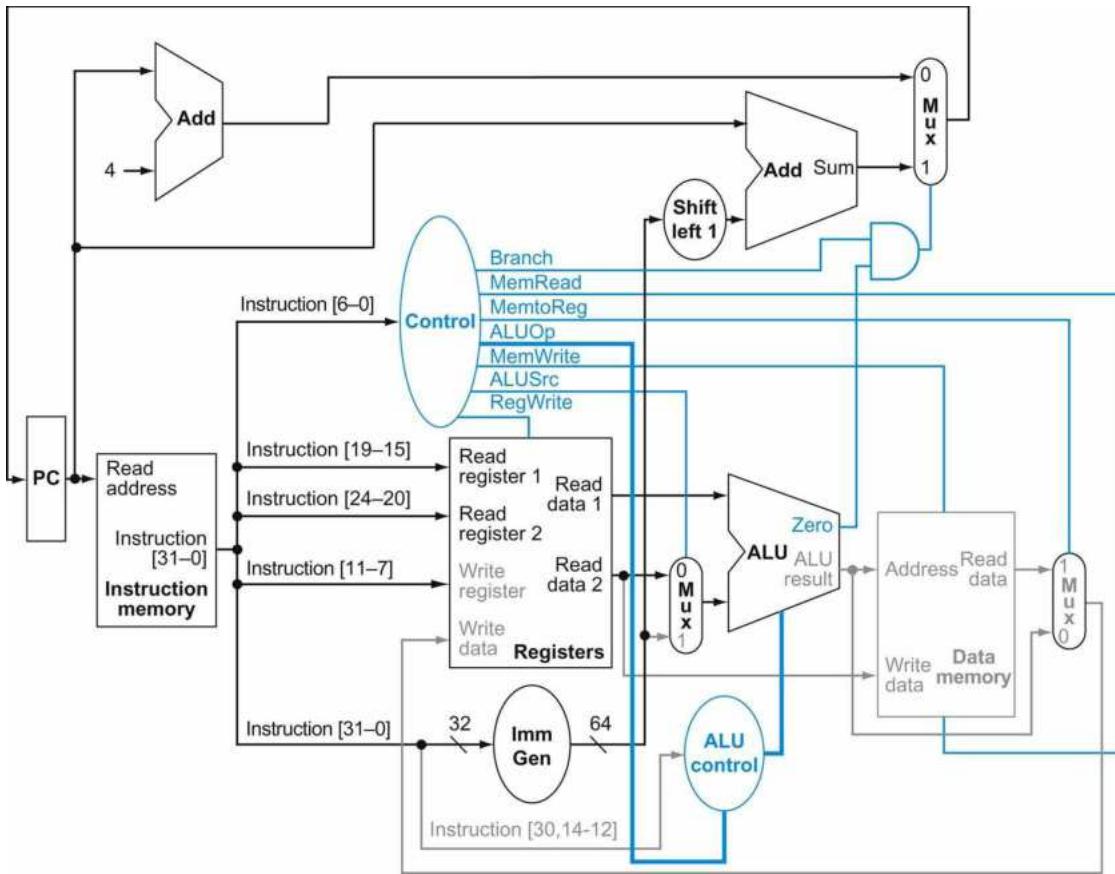
The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

Finally, we can show the operation of the branch-if-equal instruction, such as `beq x1, x2, offset`, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with  $\text{PC} + 4$  or the branch target address. [Figure 4.21](#) shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers,  $x_1$  and  $x_2$ , are read from the register file.
3. The ALU subtracts one data value from the other data value, both

read from the register file. The value of PC is added to the sign-extended, 12 bits of the instruction ( $\text{offset}$ ) left shifted by one; the result is the branch target address.

4. The Zero status information from the ALU is used to decide which adder result to store in the PC.



**FIGURE 4.21** The datapath in operation for a branch-if-equal instruction.

The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

## Finalizing Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of [Figure 4.18](#). The outputs are the control lines, and the inputs are the opcode bits. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

[Figure 4.22](#) defines the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show



this final step in [Section C.2](#) in [Appendix C](#).

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

**FIGURE 4.22** The control function for the simple single-cycle implementation is completely specified by this truth table.

The top half of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression  $Op4 \cdot Op5$ , since this is sufficient to distinguish the R-format instructions from ld, sd, and beq. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation.

## Why a Single-Cycle Implementation is not Used Today

Although the single-cycle design will work correctly, it is too

inefficient to be used in modern designs. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design. Of course, the longest possible path in the processor determines the clock cycle. This path is most likely a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1 (see [Chapter 1](#)), the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early computers with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates the great idea from [Chapter 1](#) of making the **common case fast**.



## COMMON CASE FAST

In next section, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput. Pipelining improves efficiency by executing multiple

instructions simultaneously.

### Check Yourself

Look at the control signals in [Figure 4.22](#). Can you combine any together? Can any control signal output in the figure be replaced by the inverse of another? (Hint: take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

## 4.5 An Overview of Pipelining

### **pipelining**

An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

**Pipelining** is an implementation technique in which multiple instructions are overlapped in execution. Today, **pipelining** is nearly universal.

*Never waste time.*

*American proverb*

Smith, J. E. and A. R. Plezkun [1988]. "Implementing precise interrupts in pipelined processors", *IEEE Trans. on Computers* 37:5 (May), 562–73

*Covers the difficulties in interrupting pipelined computers.*

Thornton, J. E. [1970]. *Design of a Computer. The Control Data 6600*, Glenview, IL: Scott, Foresman.

*A classic book describing a classic computer, considered the first supercomputer.*

## 4.17 Exercises

4.1 Consider the following instruction:

Instruction: and rd, rs1, rs2

Interpretation:  $\text{Reg}[rd] = \text{Reg}[rs1] \text{ AND } \text{Reg}[rs2]$

4.1.1 [5] <§4.3> What are the values of control signals generated by the control in [Figure 4.10](#) for this instruction?

4.1.2 [5] <§4.3> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.3> Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

4.2 [10] <§4.4> Explain each of the “don’t cares” in [Figure 4.18](#).

4.3 Consider the following instruction mix:

R-type	I-type (non-ld)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

4.4 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits.

A very common defect is for one signal wire to get “broken” and always register a logical 0. This is often called a “stuck-at-0” fault.

4.4.1 [5] <§4.4> Which instructions fail to operate correctly if the `MemToReg` wire is stuck at 0?

4.4.2 [5] <§4.4> Which instructions fail to operate correctly if the `ALUSrc` wire is stuck at 0?

4.5 In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: `0x00c6ba23`.

4.5.1 [10] <§4.4> What are the values of the ALU control unit’s inputs for this instruction?

4.5.2 [5] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.5.3 [10] <§4.4> For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at `Reg [xn]`.

4.5.4 [10] <§4.4> What are the input values for the ALU and the two add units?

4.5.5 [10] <§4.4> What are the values of all inputs for the registers unit?

4.6 [Section 4.4](#) does not discuss I-type instructions like `addi` or `andi`.

4.6.1 [5] <§4.4> What additional logic blocks, if any, are needed to add I-type instructions to the CPU shown in [Figure 4.21](#)? Add any necessary logic blocks to [Figure 4.21](#) and explain their purpose.

4.6.2 [10] <§4.4> List the values of the signals generated by the control unit for `addi`. Explain the reasoning for any “don’t care” control signals.

4.7 Problems in this exercise assume that the logic blocks used to implement a processor’s datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

“Register read” is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only.

“Register setup” is the amount of time a register’s data input must be stable