1

Computer Abstractions and Technology

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

PETITIONER'S
Exhibit No.:
1022
26-Feb-25
[Kiara Miller]

This Page Intentionally Left Blank

1.1 **Introduction** 3
1.2 **Below Your Program** 11
1.3 **Under the Covers** 15
1.4 **Real Stuff: Manufacturing Pentium 4 Chips** 28
1.5 **Fallacies and Pitfalls** 33
1.6 **Concluding Remarks** 35
1.7 **Historical Perspective and Further Reading** 36
1.8 **Exercises** 36

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. This unusual industry embraces innovation at a breathtaking rate. Since 1985 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 coast to coast in about a second for roughly a few cents. 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.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000003

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and also 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, physics, . . .

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."

- *Automatic teller machines:* A computer placed in the wall of banks to distribute and collect cash would have been a ridiculous concept in the 1950s, when the cheapest computer cost at least $500,000 and was the size of a car.
- *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 and improve fuel efficiency via engine controls and increase safety through the prevention of dangerous skids and through the inflation of air bags to protect occupants in a crash.
- *Laptop computers:* Who would have dreamed that advances in computer systems would lead to laptop computers, allowing students to bring computers to coffeehouses and on airplanes?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences is 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 10 to 20 years ago.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the World Wide Web has transformed our society. Among its uses are distributing news, sending flowers, buying from online catalogues, taking electronic tours to help pick vacation spots, finding others who share your esoteric interests, and even more mundane topics like finding the lecture notes of the authors of your textbooks.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, and explain why computers are omnipresent. Tomorrow's science fiction computer applications are the cashless society, automated intelligent highways, and genuinely ubiquitous computing: no one carries computers because they are available everywhere.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000004

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (discussed in Sections 1.3 and 1.4) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three different classes of applications.

Desktop computers are possibly the best-known form of computing and are characterized by the personal computer, which most readers of this book have probably used extensively. Desktop computers emphasize delivering good performance to a single user at low cost and usually are used to execute third-party software, also called shrink-wrap software. Desktop computing is one of the largest markets for computers, and the evolution of many computing technologies is driven by this class of computing, which is only about 30 years old!

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

Servers are the modern form of what was once mainframes, minicomputers, and supercomputers, and are usually accessed only via a network. Servers are oriented to carrying large 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 often 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 expandability of both computing and input/output capacity. As we will see in the Chapter 4, the performance of a server can be measured in several different ways, depending on the application of interest. In general, servers also place a greater emphasis on dependability, since a crash is usually more costly than it would be on a single-user desktop computer.

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 machine without a screen or keyboard and with a cost of 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 hundreds to thousands of processors, and usually gigabytes to **terabytes** of memory and terabytes to petabytes of storage, and cost millions 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 are a relatively small fraction of the servers and a relatively small 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 millions of dollars.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

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 washing machine and car, the computers in a cell phone

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000005

or personal digital assistant, the computers in a video game or digital television, 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, which is 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 applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a cell phone: the processor need only be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power are the most important objectives. Despite their low cost, embedded computers often have the least 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 car 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, which are used in servers, are often employed. Although this book focuses on general-purpose computers, most of the concepts apply directly, or with slight modifications, to embedded computers. In several places, we will touch on some of the unique aspects of embedded computers.

Figure 1.1 shows that during the last several years, the growth in the number of embedded computers has been much faster (40% compounded annual growth rate) than the growth rate among desktop computers and servers (9% annually). Note that the embedded computers include cell phones, video games, digital TVs and set-top boxes, personal digital assistants, and a variety of such consumer devices. Note that this data does not include low-end embedded control devices that use 8-bit and 16-bit processors.

Elaboration: Elaborations are short sections used throughout the text to provide more detail on a particular subject, which 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. The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip. The availability of synthesis tools that can generate a chip from a Verilog specification, together with the capacity of modern silicon chips, has made such special-purpose processors highly attractive. Since the core can be synthesized for different semiconductor manufacturing lines, using a core provides flexibility in choosing a manufacturer as well. In the last few years, the use of cores has

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000006

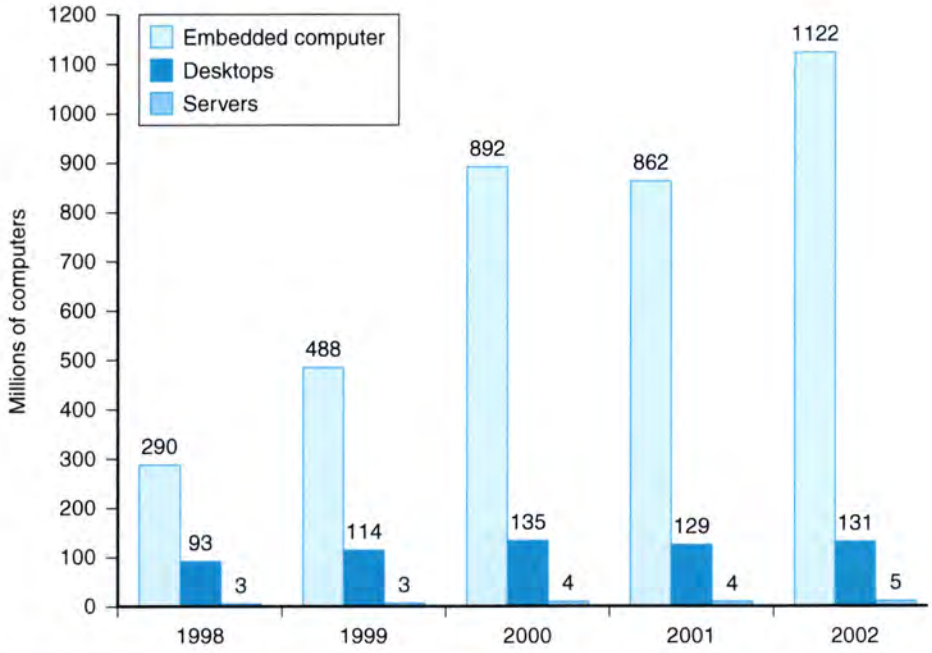


FIGURE 1.1 The number of distinct processors sold between 1998 and 2002. These counts are obtained somewhat differently, so some caution is required in interpreting the results. For example, the totals for desktops and servers count complete computer systems, because some fraction of these include multiple processors, the number of processors sold is somewhat higher, but probably by only 10–20% in total (since the servers, which may average more than one processor per system, are only about 3% of the desktop sales, which are predominantly single-processor systems). The totals for embedded computers actually count processors, many of which are not even visible, and in some cases there may be multiple processors per device.

been growing very fast. For example, in 1998 only 31% of the embedded processors were cores. By 2002, 56% of the embedded processors were cores. Furthermore, while the overall growth rate in the embedded market has been 40% per year, this growth has been primarily driven by cores, where the compounded annual growth rate has been 63%!

Figure 1.2 shows the major architectures sold in these markets with counts for each architecture, across all three types of products (embedded, desktop, and server). Only 32-bit and 64-bit processors are included, although 32-bit processors are the vast majority for most of the architectures.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000007

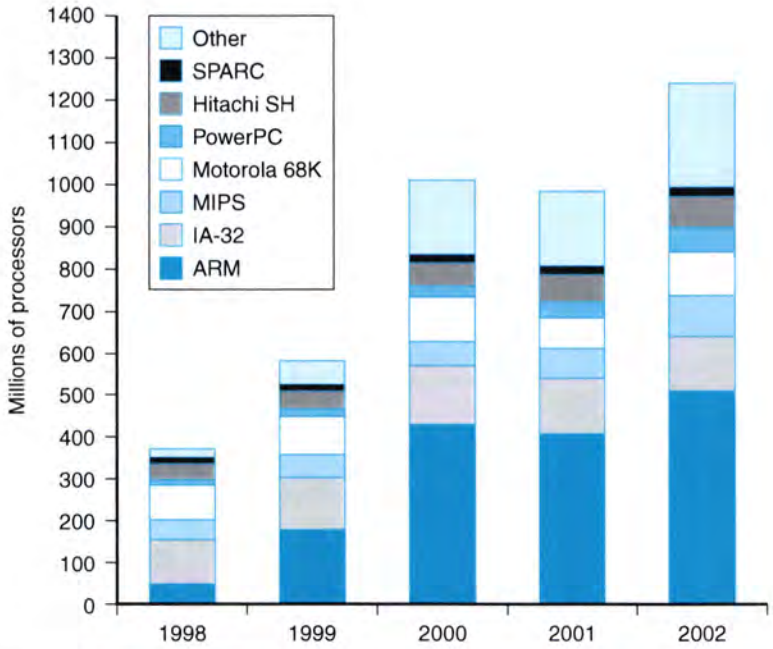


FIGURE 1.2 Sales of microprocessors between 1998 and 2002 by instruction set architecture combining all uses. The "other" category refers to processors that are either application-specific or customized architectures. In the case of ARM, roughly 80% of the sales are for cell phones, where an ARM core is used in conjunction with application-specific logic on a chip.

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 successful 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 hierarchical nature of memories and the parallel nature of processors. Programmers who seek to build competitive versions of compilers, operating systems, databases, and even applications will therefore need to increase their knowledge of computer organization.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000008

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, *A Computer Architecture: A Quantitative Approach.*

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, and introduces integrated circuits, the technology that fuels the computer revolution. In this chapter, and later ones, you will likely see a lot of 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, the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000009

will be impressed as you correctly use words such as BIOS, DIMM, CPU, cache, DRAM, ATA, PCI, and many others.

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, with the first one appearing below. These elements summarize important insights into program performance.

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 I/O operations. The following 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 machine instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 5, 6, and 7
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapter 8

Check Yourself

"Check Yourself" sections are designed to help readers assess whether they have comprehended 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. Section 1.1 showed that the number of embedded processors sold every year greatly outnumbers the number of desktop 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 desktop computers in your home?

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000010

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 Below Your Program

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

A typical application, such as a word processor or a large database system, may consist of hundreds of thousands to 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 simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions.

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 applications software, as shown in Figure 1.3.

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

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

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

- handling basic input and output operations
- allocating storage and memory
- providing for sharing the computer among multiple applications using it simultaneously

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

Compilers perform another vital function: the translation of a program written in a high-level language, such as C or Java, into instructions that the hardware

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000011

Although the decimal example above happened to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; the hardware is shown in Figure 3.5. We have drawn the hardware so that data flows from top to bottom to more closely resemble the paper-and-pencil method.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step as it may be added to the intermediate products. Over 32 steps a 32-bit multiplicand would move 32 bits to the left. Hence we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and 0 in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

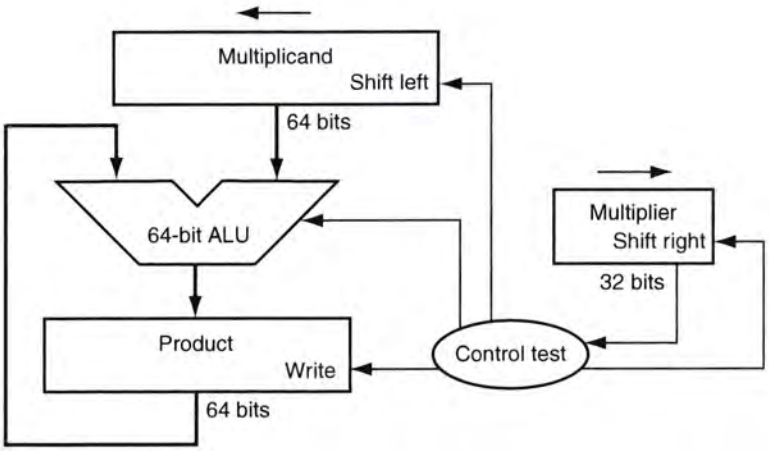


FIGURE 3.5 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000013

Figure 3.6 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took a

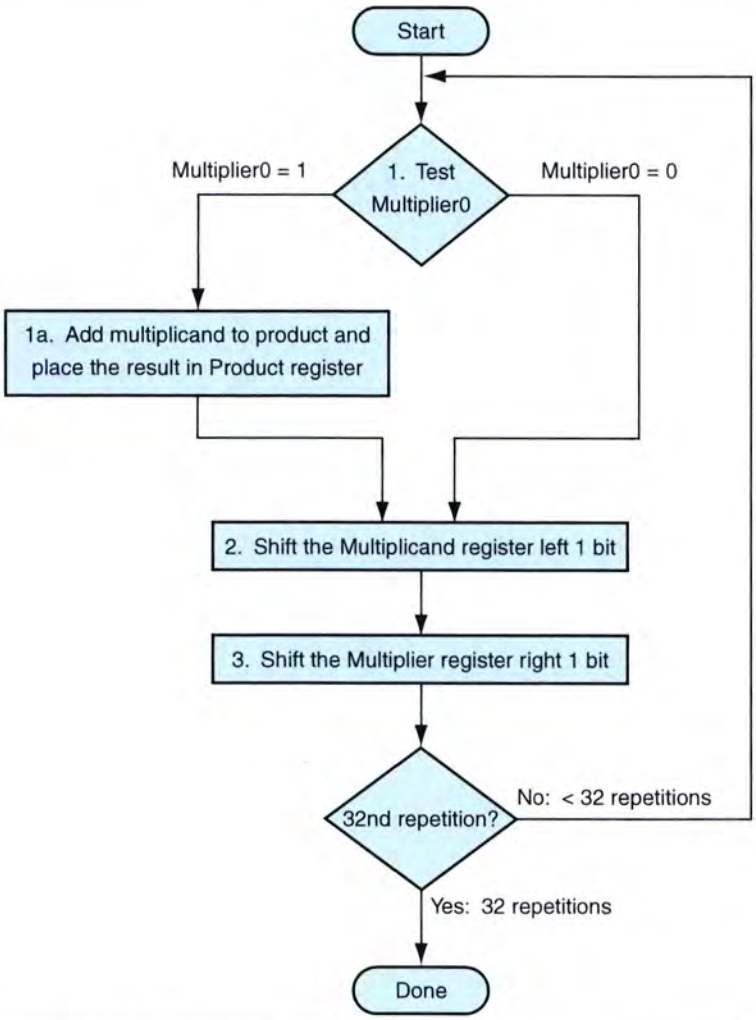


FIGURE 3.6 The first multiplication algorithm, using the hardware shown in Figure 3.5. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000014

clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Chapter 4, page 267) reminds us that even a moderate frequency for a slow operation can limit performance.

This algorithm and hardware are easily refined to take 1 clock cycle per step. The speedup comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a one. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.7 shows the revised hardware.

Hardware Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

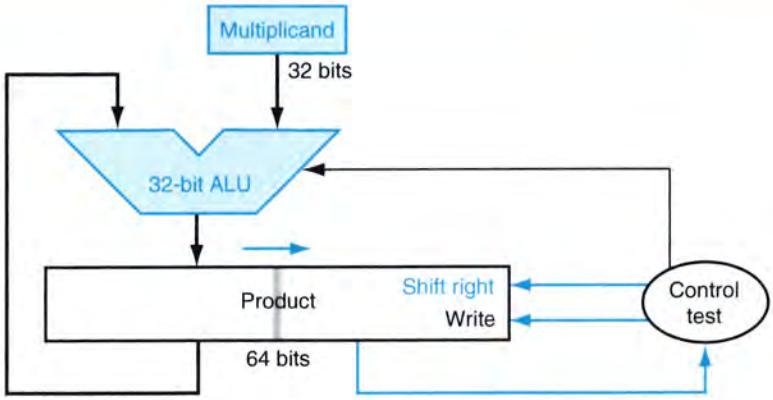


FIGURE 3.7 Refined version of the multiplication hardware. Compare with the first version in Figure 3.5. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000015

EXAMPLE

A Multiply Algorithm

Using 4-bit numbers to save space, multiply 2_{ten} □ 3_{ten}, or 0010_{two} □ 0011_{two}.

ANSWER

Figure 3.8 shows the value of each register for each of the steps labeled according to Figure 3.6, with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Signed Multiplication

So far we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers provided that we remember that the numbers we are dealing with have infinite digits, and that we are only representing them with 32 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 □ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 □ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 □ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 □ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.8 Multiply example using algorithm in Figure 3.6. The bit examined to determine the next step is circled in color.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000016

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build a much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder. Figure 3.9 shows how they would be connected.

Why is this much hardware faster? The sequential multiplier pays the overhead of a clock for each bit of the product. This multiplier array of adders does not. A second reason is this large collection of adders lends itself to many optimizations to gain further improvements. One example is using *carry save adders* to add such a large column of numbers; see Exercises 3.24 and 3.49. A third reason is that it is easy to pipeline such a design to be able to support many multiplies simultaneously (see Chapter 6).

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Summary

Multiplication is accomplished by simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of two.

Hardware Software Interface

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. There is no overflow if Hi is 0 for `multu` or the replicated sign of Lo for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer Hi to a general-purpose register to test for overflow.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000017

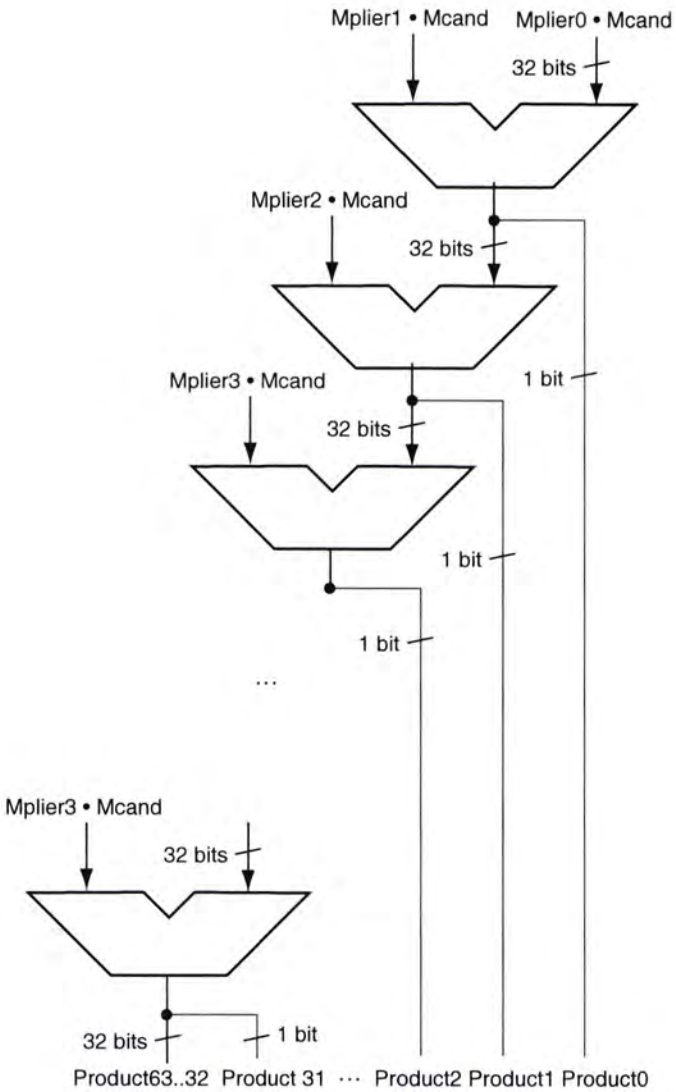


FIGURE 3.9 Fast multiplication hardware. Rather than use a single 32-bit adder 32 times, this hardware "unrolls the loop" to use 32 adders. Each adder produces a 32-bit sum and a carry out. The least significant bit is a bit of the product, and the carry out and the upper 31 bits of the sum are passed along to the next adder.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000018

3.5 Division

Divide et impera.

Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)1001010_{ten}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & \quad 10 & \\
 & \quad 101 & \\
 & \quad 1010 & \\
 & \quad \underline{-1000} & \\
 & \qquad 10_{ten} & \text{Remainder}
\end{array}
$$

The two operands (**dividend** and **divisor**) and the result (**quotient**) of divide are accompanied by a second result called the **remainder**. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} \square \text{Divisor} + \text{Remainder}$$

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.

The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

Let's assume that both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values, and we will ignore the sign for now.

dividend A number being divided.

divisor A number that the dividend is divided by.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000019

A Division Algorithm and Hardware

Figure 3.10 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 3.11 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

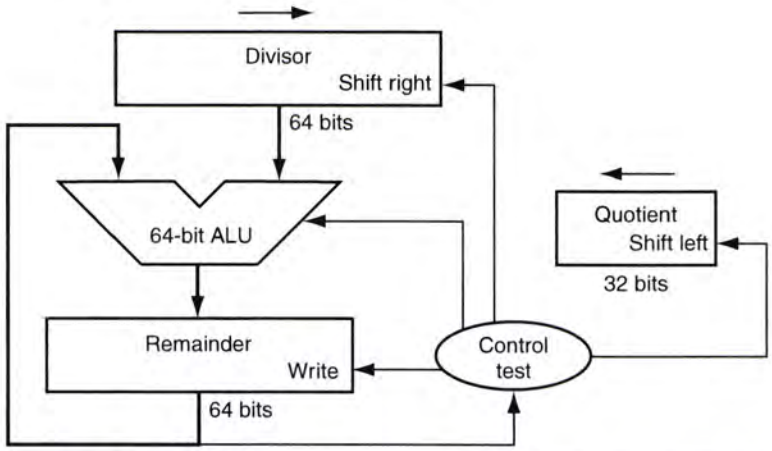


FIGURE 3.10 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000020

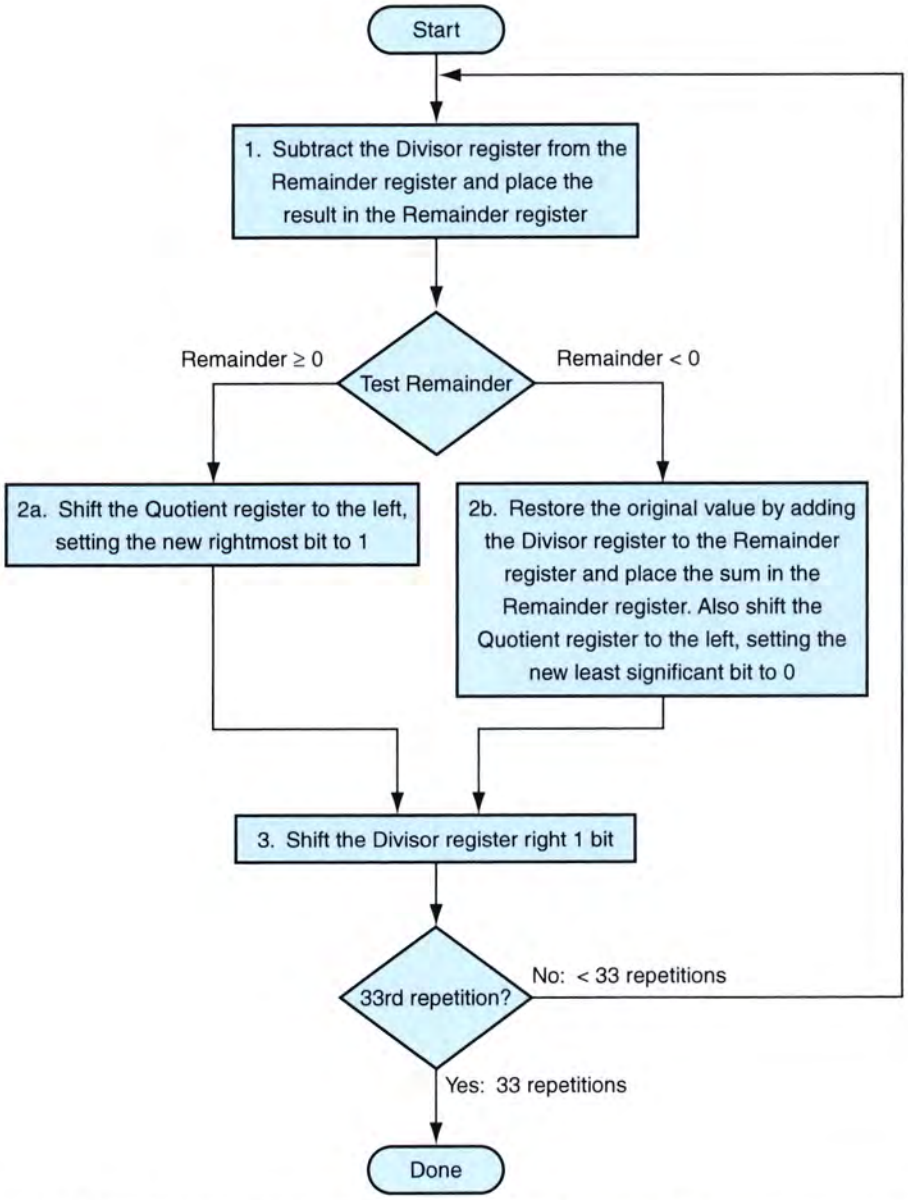


FIGURE 3.11 A division algorithm, using the hardware in Figure 3.10. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000021

A Divide Algorithm

EXAMPLE

Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten}, or $0000\ 0111_{two}$ by 0010_{two}.

ANSWER

Figure 3.12 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

This algorithm and hardware can be refined to be faster and cheaper. The speedup comes from shifting the operands and quotient at the same time as the subtract. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.13 shows the revised hardware.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 □ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 □ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 □ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 □ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 □ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.12 Division example using the algorithm in Figure 3.11. The bit examined to determine the next step is circled in color.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000022

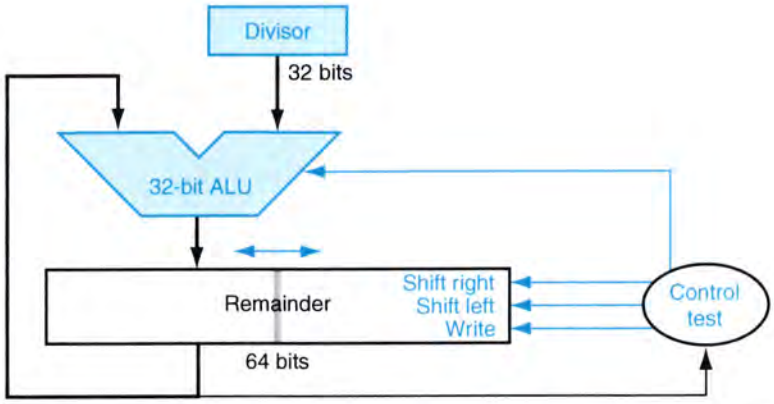


FIGURE 3.13 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.10, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register.

Signed Division

So far we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

Dividend = Quotient □ Divisor + Remainder

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

+7 ÷ +2: Quotient = +3, Remainder = +1

Checking the results:

7 = 3 □ 2 + (+1) = 6 + 1

If we change the sign of the dividend, the quotient must change as well:

–7 ÷ +2: Quotient = –3

Rewriting our basic formula to calculate the remainder:

Remainder = □Dividend – Quotient □ Divisor□ = – 7 – □–3 □ +2□ = –7–□–6□ = –1

So,

–7 ÷ +2: Quotient = –3, Remainder = –1

Checking the results again:

–7 = –3 □ 2 + (–1) = –6 – 1

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000023

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

+7 ÷ −2: Quotient = −3, Remainder = +1
−7 ÷ −2: Quotient = +3, Remainder = −1

Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division

We used 32 adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per bit. The *SRT division* technique tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong guesses. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The fallacy on page 222 in Section 3.8 shows what can happen if the table is incorrect.

Divide in MIPS

You may have already observed that the same sequential hardware can be used for both multiply and divide in Figures 3.7 and 3.13. The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts. Hence, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide. As we might expect from the algorithm above, Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.

To handle both signed integers and unsigned integers, MIPS has two instructions: *divide* (`div`) and *divide unsigned* (`divu`). The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000024

Summary

The common hardware support for multiply and divide allows MIPS to provide a single pair of 32-bit registers that are used both for multiply and divide. Figure 3.14 summarizes the additions to the MIPS architecture for the last two sections.

Hardware Software Interface

MIPS divide instructions ignore overflow, so software must determine if the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. MIPS software must check the divisor to discover division by 0 as well as overflow.

Elaboration: An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step since $(r + d) \square 2 - d = r \square 2 + d \square 2 - d = r \square 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock per step, is explored further in Exercise 3.29; the algorithm here is called *restoring* division.

3.6 Floating Point

Speed gets you nowhere if you're headed the wrong way.

American proverb

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ ($\square$)

$2.71828\ldots_{ten}$ (e)

0.000000001_{ten} or $1.0_{ten} \square 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \square 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{ten} \square 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \square 10^{-8}$ and $10.0_{ten} \square 10^{-10}$ are not.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

normalized A number in floating-point notation that has no leading 0s.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000025

MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw	$s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	lhu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh	$s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor	$s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Condi-tional branch	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Uncondi-tional jump	jump	j	2500	go to 10000	Jump to target address
	jump register	jr	$ra	go to $ra	For switch, procedure return
	jump and link	jal	2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.14 MIPS architecture revealed thus far. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the hi and lo registers to support multiply and divide. Color indicates the portions revealed since Figure 3.4 on page 175. MIPS machine language is listed in the MIPS summary reference card at the front of this book.

(page 190)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000026

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \square 2^{-1}$$

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called **floating point** because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

$$1.xxxxxxxxx_{\text{two}} \square 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

A standard scientific notation for reals in normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent** because a fixed word size means you must take a bit from one to add a bit to the other. This trade-off is between precision and range: Increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 2 reminds us, good design demands good compromise.

fraction The value, generally between 0 and 1, placed in the fraction field.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000027

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
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

In general, floating-point numbers are generally of the form

$$(-1)^S \square F \square 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that MIPS does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \square 10^{-38}$ and numbers almost as large as $2.0_{ten} \square 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, people call this event **underflow**. This situation occurs when the negative exponent is too large to fit in the exponent field.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent. In C this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

double precision A floating-point value represented in two 32-bit words.

single precision A floating-point value represented in a single 32-bit word.

The representation of a double precision floating-point number takes two MIPS words, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction.

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
s	exponent											fraction																			
1 bit	11 bits											20 bits																			
fraction (continued)																															
32 bits																															

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000028

MIPS double precision allows numbers almost as small as $2.0_{ten} \square 10^{-308}$ and almost as large as $2.0_{ten} \square 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the larger significand.

These formats go beyond MIPS. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the significand, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or 53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00 \ldots 00_{two}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$\square -1 \square^{S} \square \square 1 + \text{Fraction} \square \square 2^{E}$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s1, s2, s3, . . . , then the value is

$$\square -1 \square^{S} \square \square 1 + \square s1 \square 2^{-1} \square + \square s2 \square 2^{-2} \square + \square s3 \square 2^{-3} \square + \square s4 \square 2^{-4} \square + \ldots \square \square 2^{E}$$

Figure 3.15 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\square$ or $-\square$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when it is convenient.

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a quick test of less than, greater than, or equal to 0. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000029

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

FIGURE 3.15 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the elaboration on page 217.

Placing the exponent before the significand also simplifies sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{two} \square 2^{-1}$ would be represented as

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	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \square 2^{+1}$ would look like the smaller binary number

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00\ldots00_{two}$ and the most positive as $11\ldots11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so –1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \square (1 + \text{Fraction}) \square 2^{(\text{Exponent} - \text{Bias})}$$

The exponent bias for double precision is 1023.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000030

Thus IEEE 754 notation can be processed by integer compares to accelerate sorting of floating-point numbers. Let's show the representation.

Floating-Point Representation

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

ANSWER

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2{}_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2{}_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \square 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \square 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \square (1 + \text{Fraction}) \square 2^{(\text{Exponent} - 127)}$$

When we subtract the bias 127 from the exponent of $-1.1_{two} \square 2^{-1}$, the result is

$$(-1)^1 \square (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \square 2^{(126 - 127)}$$

The single precision binary representation of -0.75_{ten} is then

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

1 bit 8 bits 23 bits

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000031

The double precision representation is

$$(-1)^1 \square (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\text{two}) \square 2^{(1022-1023)}$$

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

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Now let's try going the other direction.

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

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
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

ANSWER

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \square 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \square (1 + \text{Fraction}) \square 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \square (1 + 0.25) \square 2^{(129-127)} \\ &= -1 \square 1.25 \square 2^2 \\ &= -1.25 \square 4 \\ &= -5.0\end{aligned}$$

In the next sections we will give the algorithms for floating-point addition and multiplication. At their core, they use the corresponding integer operations on the significands, but extra bookkeeping is necessary to handle the exponents and normalize the result. We first give an intuitive derivation of the algorithms in decimal, and then give a more detailed, binary version in the figures.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000032

Elaboration: In an attempt to increase range without removing bits from the significand, some computers before the IEEE 754 standard used a base other than 2. For example, the IBM 360 and 370 mainframe computers use base 16. Since changing the IBM exponent by one means shifting the significand by 4 bits, "normalized" base 16 numbers can have up to 3 leading bits of 0s! Hence, hexadecimal digits mean that up to 3 bits must be dropped from the significand, which leads to surprising problems in the accuracy of floating-point arithmetic, as noted in Section .

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \square 10^1 + 1.610_{ten} \square 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \square 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \square 10^{-1} = 0.1610_{ten} \square 10^0 = 0.01610_{ten} \square 10^1$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \square 10^1$. Thus the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really:

$$0.016_{ten} \square 10^1$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{ten} \\ +\ \ 0.016_{ten} \\ \hline 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \square 10^1$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \square 10^1 = 1.0015_{ten} \square 10^2$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000033

shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{ten} \square 10^2$$

is rounded to four digits in the significand to

$$1.002_{ten} \square 10^2$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Figure 3.16 shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all zero bits in the exponent is reserved and used for the floating-point representation of zero. Also, the pattern of all one bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the Elaboration on page 217). Thus, for single precision, the maximum exponent is 127, and the minimum exponent is –126. The limits for double precision are 1023 and –1022.

EXAMPLE

Decimal Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.16.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000034

ANSWER

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{aligned} 0.5_{ten} &= 1/2_{ten} &&= 1/2^1{}_{ten} \\ &= 0.1_{two} &&= 0.1_{two} \times 2^0 &&= 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} &= -7/16_{ten} &&= -7/2^4{}_{ten} \\ &= -0.0111_{two} &&= -0.0111_{two} \times 2^0 &&= -1.110_{two} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{two} \times 2^{-1} &= 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} \\ &= 1.000_{two} \times 2^{-4} \end{aligned}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{two} \times 2^{-4} &= 0.0001000_{two} &&= 0.0001_{two} \\ &= 1/2^4{}_{ten} &&= 1/16_{ten} &&= 0.0625_{ten} \end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure 3.17 sketches the basic organization of hardware for floating-point addition.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000035

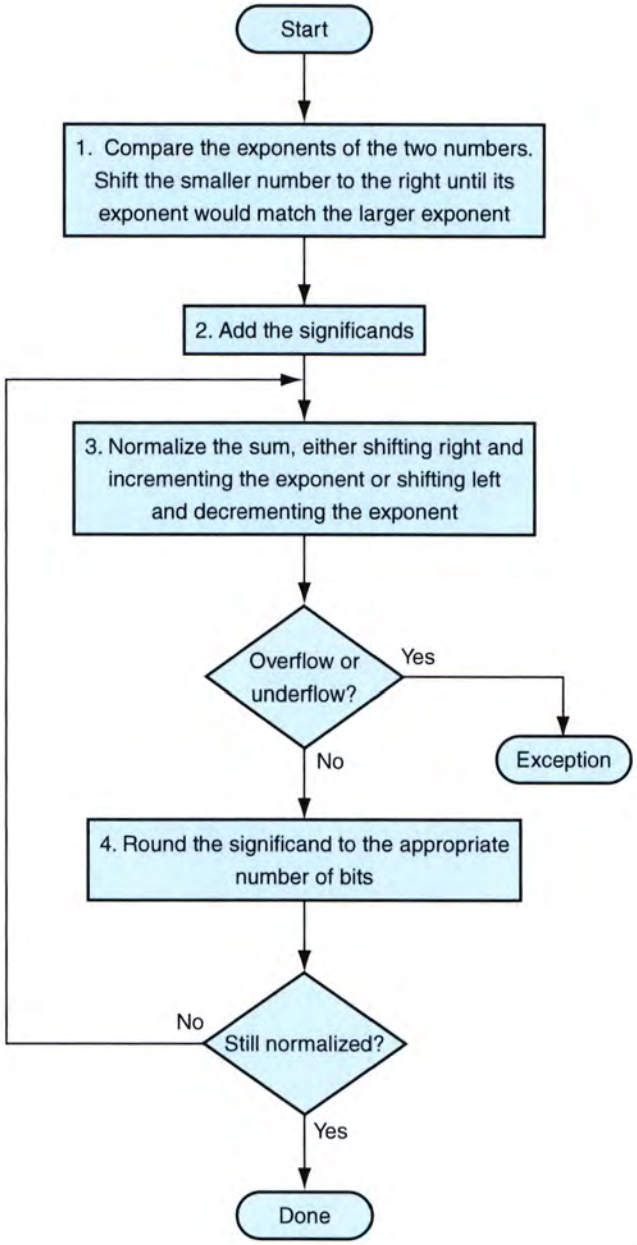


FIGURE 3.16 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000036

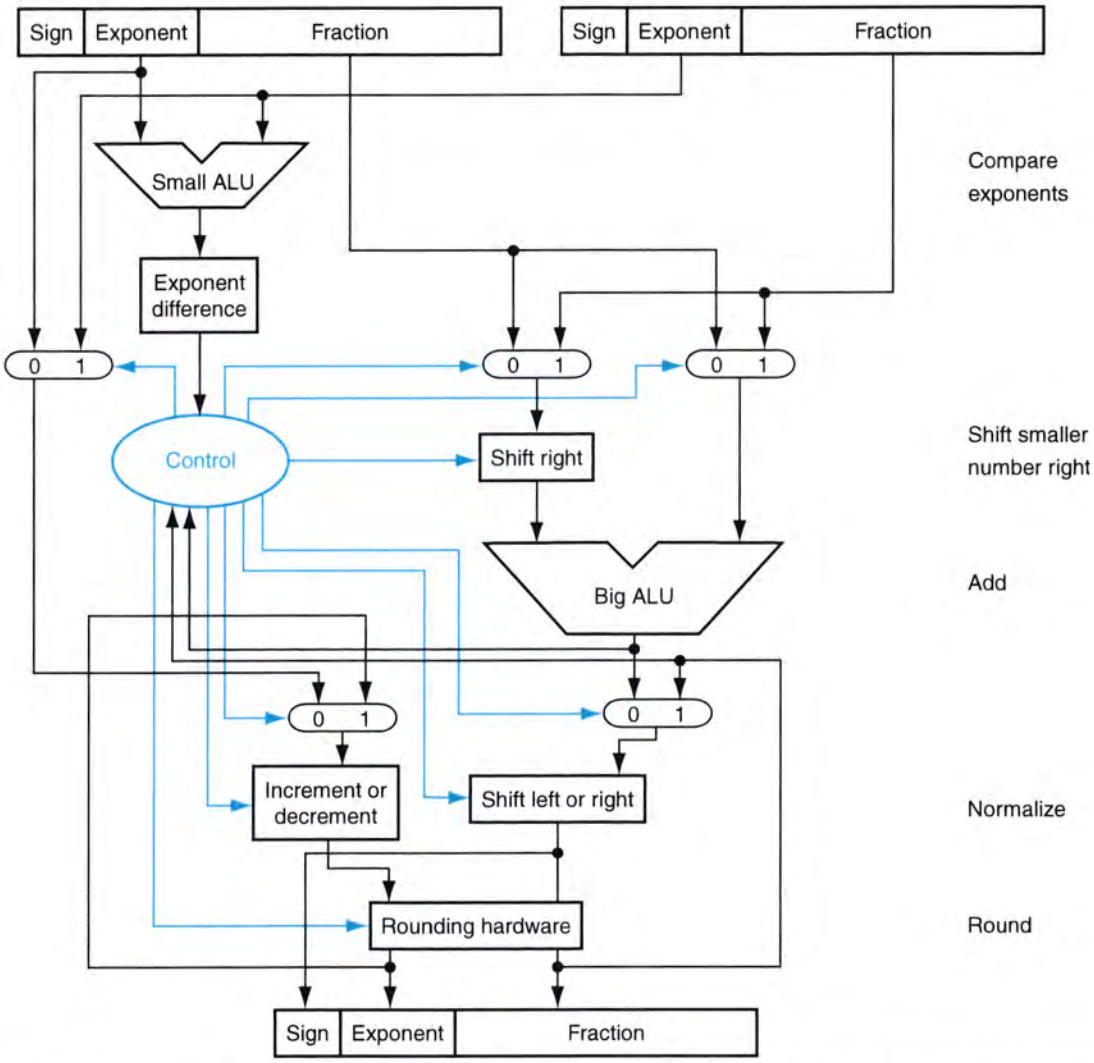


FIGURE 3.17 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.16 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000037

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{ten} \square 10^{10} \square 9.200_{ten} \square 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: $10 + 127 = 137$, and $-5 + 127 = 122$, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \square 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{ten} \\ \times\ 9.200_{ten} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{ten} \end{array}$$

There are three digits to the right of the decimal for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{ten}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is $10.212 \square 10^5$.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000038

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \square 10^5 = 1.0212_{ten} \square 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{ten} \square 10^6$$

is rounded to four digits in the significand to

$$1.021_{ten} \square 10^6$$

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise it's negative. Hence the product is

$$+1.021_{ten} \square 10^6$$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication the sign of the product is determined by the signs of the operands.

Once again, as Figure 3.18 shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

EXAMPLE

Decimal Floating-Point Multiplication

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.18.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000039

ANSWER

In binary, the task is multiplying $1.000_{two} \square 2^{-1}$ by $-1.110_{two} \square 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \square 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \square 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \square 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{two} \square 2^{-3}$$

Converting to decimal to check our results:

$$-1.110_{two} \square 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^5{}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000040

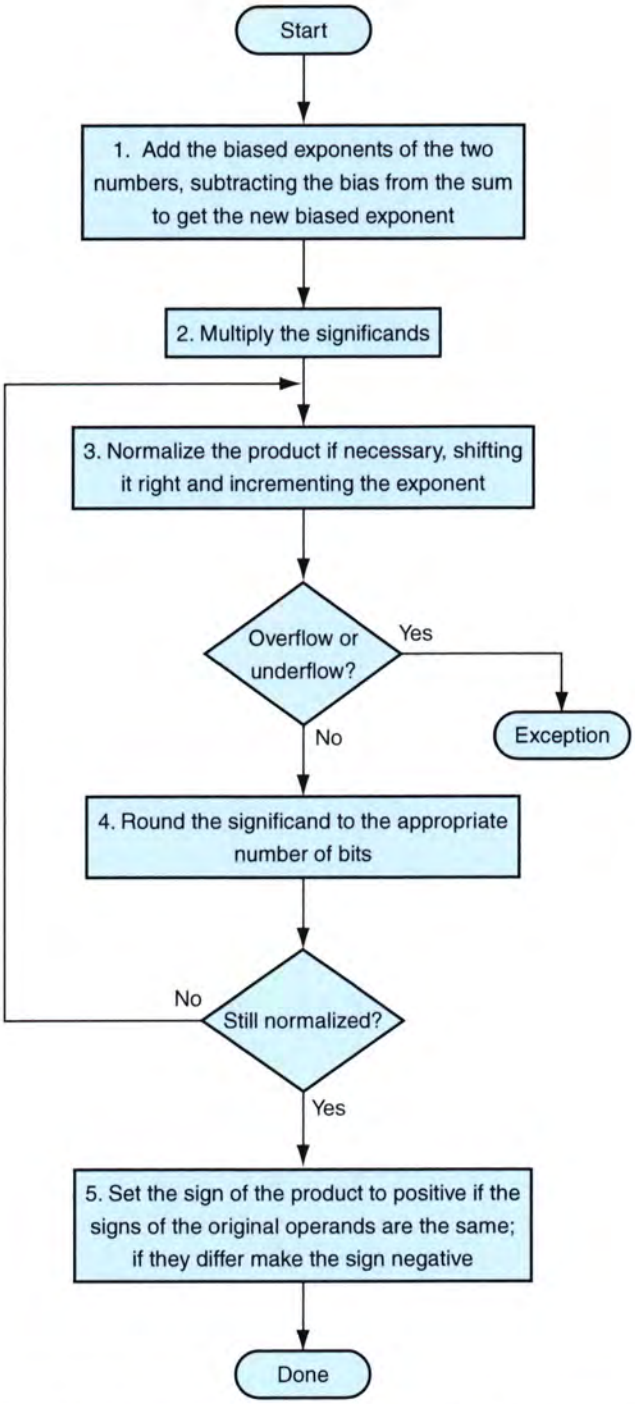


FIGURE 3.18 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

(page 205)

Floating-Point Instructions in MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

- Floating-point *addition, single* (`add.s`) and *addition, double* (`add.d`)
- Floating-point *subtraction, single* (`sub.s`) and *subtraction, double* (`sub.d`)
- Floating-point *multiplication, single* (`mul.s`) and *multiplication, double* (`mul.d`)
- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where `x` may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

The MIPS designers decided to add separate floating-point registers—called `$f0`, `$f1`, `$f2`, ...—used either for single precision or double precision. Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1    $f4,x($sp)  # Load 32-bit F.P. number into F4
lwc1    $f6,y($sp)  # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1    $f2,z($sp)  # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

Figure 3.19 summarizes the floating-point portion of the MIPS architecture revealed in this chapter, with the additions to support floating point shown in color. Similar to Figure 2.25 on page 103 in Chapter 2, we show the encoding of these instructions in Figure 3.20.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000042

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 □ $f6	FP multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 □ $f6	FP multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

FIGURE 3.19 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, on page A-49, for more detail.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000043

op(31:26):								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FlPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

FIGURE 3.20 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure lw is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op = 010001_{two}) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction) is 000001_{two} and the op field (bits 31–26) is 010001_{two}. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (f = s so rs = 10000) or double precision (f = d so rs = 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 = 1 =>bc1.t) or false (bit 16 = 0 =>bc1.f). Instructions in color are described in Chapters 2 or 3, with Appendix A covering all instructions.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000044

Hardware Software Interface

One issue that computer designers face in supporting floating-point arithmetic is whether to use the same registers used by the integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a separate set of data transfer instructions to move data between floating-point registers and memory.

The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point; for example, some computers convert all sized operands in registers into a single internal format

Compiling a Floating-Point C Program into MIPS Assembly Code

EXAMPLE

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
        {
                return ((5.0/9.0) * (fahr - 32.0));
        }
```

Assume that the floating-point argument `fahr` is passed in `$f12` and the result should go in `$f0`. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

ANSWER

We assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer `$gp`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
      lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
      lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
```

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000045

They are then divided to get the fraction 5.0/9.0:

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next we load the constant 32.0 and then subtract it from `fahr` (`$f12`):

```
lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `$f0` as the return result, and then return:

```
mul.s$f0,  $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr  $ra # return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

EXAMPLE

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of X = X + Y * Z. Let's assume X, Y, and Z are all square matrices with 32 elements in each dimension.

```
void mm (double x[][], double y[][], double z[][])
{
        int i, j, k;

        for (i = 0; i! = 32; i = i + 1)
        for (j = 0; j! = 32; j = j + 1)
        for (k = 0; k! = 32; k = k + 1)
          x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

The array starting addresses are parameters, so they are in `$a0`, `$a1`, and `$a2`. Assume that the integer variables are in `$s0`, `$s1`, and `$s2`, respectively. What is the MIPS assembly code for the body of the procedure?

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000046

ANSWER

Note that x[i][j] is used in the innermost loop above. Since the loop index is k, the index does not affect x[i][j], so we can avoid loading and storing x[i][j] each iteration. Instead, the compiler loads x[i][j] into a register outside the loop, accumulates the sum of the products of y[i][k] and z[k][j] in that same register, and then stores the sum into x[i][j] upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions li (which loads a constant into a register), and l.d and s.d (which the assembler turns into a pair of data transfer instructions, lwc1 or swc1, to a pair of floating-point registers).

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
mm:...
        li      $t1, 32   # $t1 = 32 (row size/loop end)
        li      $s0, 0    # i = 0; initialize 1st for loop
L1:     li      $s1, 0    # j = 0; restart 2nd for loop
L2:     li      $s2, 0    # k = 0; restart 3rd for loop
```

To calculate the address of x[i][j], we need to know how a 32 □ 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the i "single-dimensional arrays," or rows, to get the one we want. Thus we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
sll  $t2, $s0, 5 # $t2 = i * 2^5 (size of row of x)
```

Now we add the second index to select the jth element of the desired row:

```
addu  $t2, $t2, $s1 # $t2 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

```
sll  $t2, $t2, 3 # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of x, giving the address of x[i][j], and then load the double precision number x[i][j] into $f4:

```
addu  $t2, $a0, $t2 # $t2 = byte address of x[i][j]
l.d   $f4, 0($t2) # $f4 = 8 bytes of x[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number z[k][j].

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000047

```
L3:   sll $t0, $s2, 5 # $t0 = k * 2^5 (size of row of z)
      addu $t0, $t0, $s1 # $t0 = k * size(row) + j
      sll $t0, $t0, 3 # $t0 = byte offset of [k][j]
      addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
      l.d $f16, 0($t0) # $f16 = 8 bytes of z[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number `y[i][k]`.

```
sll     $t2, $s0, 5 # $t0 = i * 2^5 (size of row of y)
addu    $t0, $t0, $s2 # $t0 = i * size(row) + k
sll     $t0, $t0, 3 # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0 # $t0 = byte address of y[i][k]
l.d     $f18, 0($t0) # $f18 = 8 bytes of y[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `y` and `z` located in registers `$f18` and `$f16`, and then accumulate the sum in `$f4`.

```
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16 # f4 = x[i][j]+ y[i][k] * z[k][j]
```

The final block increments the index `k` and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in `$f4` into `x[i][j]`.

```
addiu  $s2, $s2, 1 # $k k + 1
bne    $s2, $t1, L3 # if (k != 32) go to L3
s.d    $f4, 0($t2) # x[i][j] = $f4
```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
addiu  $s1, $s1, 1 # $j = j + 1
bne    $s1, $t1, L2 # if (j != 32) go to L2
addiu  $s0, $s0, 1 # $i = i + 1
bne    $s0, $t1, L1 # if (i != 32) go to L1
...
```

Elaboration: The array layout discussed in the example, called *row major order,* is used by C and many other programming languages. Fortran instead uses *column major order,* whereby the array is stored column by column.

Only 16 of the 32 MIPS floating-point registers could originally be used for single precision operations: `$f0`, `$f2`, `$f4`, . . . , `$f30`. Double precision is computed using pairs

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000048

of these registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added `l.d` and `s.d` to the instruction set. MIPS-32 also added "paired single" versions of all floating-point instructions, where a single instruction results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers. For example, `add.ps F0, F2, F4` is equivalent to `add.s F0, F2, F4` followed by `add.s F1, F3, F5`.

Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence the floating-point unit, including the floating-point registers, were optionally available as a second chip. Such optional accelerator chips are called *coprocessors,* and explain the acronym for floating-point loads in MIPS: `lwc1` means load word to coprocessor 1, the floating-point unit. (Coprocessor 0 deals with virtual memory, described in Chapter 7.) Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and hence the term "coprocessor" joins "accumulator" and "core memory" as quaint terms that date the speaker.

Elaboration: Although there are many ways to throw hardware at floating-point multiply to make it go fast, floating-point division is considerably more challenging to make fast and accurate. Slow divides in early computers led to removal of divides from many algorithms, but parallel computers have inspired rediscovery of divide-intensive algorithms that work better on these computers. Hence, we may need faster divides.

One technique to leverage a fast multiplier is *Newton's iteration,* where division is recast as finding the zero of a function to find the reciprocal $1/x$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solves this problem by calculating an extra-precise reciprocal.

Elaboration: Java embraces IEEE 754 by name in its definition of Java floating-point data types and operations. Thus, the code in the first example could have well been generated for a class method that converted Fahrenheit to Celsius.

The second example uses multiple dimensional arrays, which are not explicitly supported in Java. Java allows arrays of arrays, but each array may have its own length, unlike multiple dimensional arrays in C. Like the examples in Chapter 2, a Java version of this second example would require a good deal of checking code for array bounds, including a new length calculation at the end of row. It would also need to check that the object reference is not null.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000049

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

exactly in double precision floating point. The best we can do is get the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps 2 extra bits on the right during intermediate additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate the value of these extra digits.

Rounding with Guard Digits

EXAMPLE

Add $2.56_{ten} \square 10^0$ to $2.34_{ten} \square 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

ANSWER

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \square 10^0$ becomes $0.0256_{ten} \square 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{ten} \\ +\ 0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Thus the sum is $2.3656_{ten} \square 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tie-breaker. Rounding the sum up with three significant digits yields $2.37_{ten} \square 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{ten} \\ +\ 0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

The answer is $2.36_{ten} \square 10^2$, off by 1 in the last digit from the sum above.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000050

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number was off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

units in the last place (ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be prepresented.

Elaboration: Although the example above really needed just one extra digit, multiply can need two. A binary product may have one leading 0 bit; hence, the normalizing step must shift the product 1 bit left. This shifts the guard digit into the least significant bit of the product, leaving the round bit to help accurately round the product.

There are four rounding modes: always round up (toward +□), always round down (toward –□), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The Internal Revenue Service always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

The goal of the extra rounding bits is to allow the computer to get the same results as if the intermediate results were calculated to infinite precision and then rounded.

To support this goal and rounding to the nearest even, the standard has a third bit in addition to guard and round; it is set whenever there are nonzero bits to the right of the round bit. This **sticky bit** allows the computer to see the difference between $0.50 \ldots 00_{ten}$ and $0.50 \ldots 01_{ten}$ when rounding.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \square 10^{-1}$ to $2.34_{ten} \square 10^{2}$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000...00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000...00, we round instead to 2.35.

Summary

The Big Picture below reinforces the stored-program concept from Chapter 2; the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

$$(-1)^{S} \square (1 + \text{Fraction}) \square 2^{(\text{Exponent} - \text{bias})}$$

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000051

is almost always an approximation of the real number. Computer systems must take care to minimize this gap between computer arithmetic and arithmetic in the real world, and programmers at times need to be aware of the implications of this approximation.

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

The BIG Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size, hence limited precision; it's possible to calculate a number too big or too small to be represented in a word. Programmers must remember these limits and write programs accordingly.

Hardware Software Interface

In the last chapter we presented the storage classes of the programming language C (see the Hardware Software Interface section on page 142). The table above shows some of the C and Java data types together with the MIPS data transfer instructions and instructions that operate on those types that appear in Chapters 2 and 3. Note that Java omits unsigned integers.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000052

Check Yourself

Suppose there was a 16-bit IEEE 754 floating-point format with 5 exponent bits. What would be the likely range of numbers it could represent?

1. $1.0000\ 0000\ 00 \square 2^{0}$ to $1.1111\ 1111\ 11 \square 2^{31}, 0$
2. $\pm 1.0000\ 0000\ 0 \square 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \square 2^{15}, \pm 0, \pm \square$, NaN
3. $\pm 1.0000\ 0000\ 00 \square 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \square 2^{15}, \pm 0, \pm \square$, NaN
4. $\pm 1.0000\ 0000\ 00 \square 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \square 2^{14}, \pm 0, \pm \square$, NaN

Elaboration: To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. Hence the full MIPS instruction set has many flavors of compares to support NaNs. (Java does not support unordered compares.)

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero significand. They allow a number to degrade in significance until it becomes 0, called *gradual underflow.* For example, the smallest positive single precision normalized number is

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{two} \square 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{two} \square 2^{-126}, \text{ or } 1.0_{two} \square 2^{-149}$$

For double precision, the denorm gap goes from $1.0 \square 2^{-1022}$ to $1.0 \square 2^{-1074}$.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Also, if programmers do not expect denorms, their programs may be surprised.

3.7 Real Stuff: Floating Point in the IA-32

The IA-32 has regular multiply and divide instructions that operate entirely on registers, unlike the reliance on Hi and Lo in MIPS. (In fact, later versions of the MIPS instruction set have added similar instructions.)

The main differences are found in floating-point instructions. The IA-32 floating-point architecture is different from all other computers in the world.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000053

The IA-32 Floating-Point Architecture

The Intel 8087 floating-point coprocessor was announced in 1980. This architecture extended the 8086 with about 60 floating-point instructions.

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the two top elements of the stacks, and stores can pop elements off the stack. Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers on-chip below the top of the stack. Thus, a complete stack instruction set is supplemented by a limited set of register-memory instructions.

This hybrid is still a restricted register-memory model, however, since loads always move data to the top of the stack while incrementing the top-of-stack pointer and stores can only move the top of stack to memory. Intel uses the notation `ST` to indicate the top of stack, and `ST(i)` to represent the *i*th register below the top of stack.

Another novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Unlike the maximum of 64 bits on MIPS, the IA-32 floating-point operands on the stack are 80 bits wide. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. This *double extended precision* is not supported by programming languages, although it has been useful to programmers of mathematical software.

Memory data can be 32-bit (single precision) or 64-bit (double precision) floating-point numbers. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to floating point, and vice versa, for integer loads and stores.

The IA-32 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer processor so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure 3.21 shows some of the 60 floating-point operations. Note that we get even more combinations when including the operand modes for these operations. Figure 3.22 shows the many options for floating-point add.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000054

Data transfer	Arithmetic	Compare	Transcendental
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

FIGURE 3.21 The floating-point instructions of the IA-32. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the order of the operands in this operation. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations in the first column push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations. Not all combinations suggested by the notation are provided. Hence, F{I}SUB{R}{P} operations represents these instructions found in the IA-32: FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP. For the integer subtract instructions, there is no pop (FISUBP) or reverse pop (FISUBRP).

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is *i*th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces *i*th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.

FIGURE 3.22 The variations of operands for floating-point add in the IA-32.

The floating-point instructions are encoded using the ESC opcode of the 8086 and the postbyte address specifier (see Figure 2.46 on page 143). The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit floating point or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000055

Floating-point performance of the IA-32 family has traditionally lagged far behind other computers. Whether it is simply a lack of attention by Intel engineers or a flaw with its architecture is hard to know. We can say that many new architectures have been announced since 1980, and none have followed in Intel's footsteps. In addition, Intel created a more traditional floating-point architecture as part of SSE2.

The Intel Streaming SIMD Extension 2 (SSE2) Floating-Point Architecture

Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture, including double precision floating-point registers and operations. It includes eight registers that can be used for floating-point operands, giving the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE2 registers as floating-point registers like those found in other computers. AMD expanded the number to 16 as part of AMD64, which Intel relabled EM64T for its use.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles or two doubles. This new architecture can more than double performance over the stack architecture.

Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

Bertrand Russell, *Recent Words on the Principles of Mathematics*, 1901

3.8 Fallacies and Pitfalls

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

Fallacy: Floating-point addition is associative; that is, $x + (y + z) = (x + y) + z$.

Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000056

For example, suppose $x = -1.5_{ten} \square 10^{38}$, $y = 1.5_{ten} \square 10^{38}$, and $z = 1.0$, and that these are all single precision numbers. Then

$$\begin{aligned} x + (y + z) &= -1.5_{ten} \square 10^{38} + (1.5_{ten} \square 10^{38} + 1.0) \\ &= -1.5_{ten} \square 10^{38} + (1.5_{ten} \square 10^{38}) = 0.0 \\ (x + y) + z &= (-1.5_{ten} \square 10^{38} + 1.5_{ten} \square 10^{38}) + 1.0 \\ &= (0.0_{ten}) + 1.0 \\ &= 1.0 \end{aligned}$$

Therefore, $x + \square y + z \square \neq \square x + y \square + z$.

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{ten} \square 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{ten} \square 10^{38} + 1.0$ is still $1.5_{ten} \square 10^{38}$. That is why the sum of x, y, and z is 0.0 or 1.0, depending on the order of the floating-point additions, and hence floating-point add is *not* associative.

Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.

Recall that a binary number x, where xi means the ith bit, represents the number

$$\ldots + (x3 \square 2^3) + (x2 \square 2^2) + (x1 \square 2^1) + (x0 \square 2^0)$$

Shifting the bits of x right by n bits would seem to be the same as dividing by 2^n. And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}. The two's complement representation of -5_{ten} is

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1{,}073{,}741{,}822_{ten}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift (see **In More Depth:** Booth's Algorithm) that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

The result is -2_{ten} instead of -1_{ten}; close, but no cigar.

The PowerPC, however, does have a fast shift instruction (*shift right algebraic*) that in conjunction with a special add (add with carry) gives the same answer as dividing by a power of 2.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000057

Pitfall: The MIPS instruction add immediate unsigned `addiu` *sign-extends its 16-bit immediate field.*

Despite its name, add immediate unsigned (`addiu`) is used to add constants to signed integers when we don't care about overflow. MIPS has no subtract immediate instruction and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.23). The following is the inside story behind the headlines.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing –2, –1, 0, +1, or +2. The guess is multiplied by the divisor and

FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times, San Jose Mercury News, San Francisco Chronicle,* and *Infoworld.* The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000058

subtracted from the remainder to generate a new remainder. Like nonrestoring division (see Exercise 3.29), if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently there were five elements of the table from the 80486 that Intel thought could never be accessed, and they optimized the PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

The following is a time line of the Pentium bug morality play:

- *July 1994:* Intel discovers the bug in the Pentium. The actual cost to fix the bug was several hundred thousand dollars. Following normal bug fix procedures, it will take months to make the change, reverify, and put the corrected chip into production. Intel planned to put good chips into production in January 1995, estimating that 3 to 5 million Pentiums would be produced with the bug.
- *September 1994:* A math professor at Lynchburg College in Virginia, Thomas Nicely, discovers the bug. After calling Intel technical support and getting no official reaction, he posts his discovery on the Internet. It quickly gained a following, and some pointed out that even small errors become big when multiplying by big numbers: the fraction of people with a rare disease times the population of Europe, for example, might lead to the wrong estimate of the number of sick people.
- *November 7, 1994: Electronic Engineering Times* puts the story on its front page, which is soon picked up by other newspapers.
- *November 22, 1994:* Intel issues a press release, calling it a "glitch." The Pentium "can make errors in the ninth digit. . . . Even most engineers and financial analysts require accuracy only to the fourth or fifth decimal point. Spreadsheet and word processor users need not worry. . . . There are maybe several dozen people that this would affect. So far, we've only heard from one. . . . [Only] theoretical mathematicians (with Pentium computers purchased before the summer) should be concerned." What irked many was that customers were told to describe their application to Intel, and then *Intel* would decide whether or not their application merited a new Pentium without the divide bug.
- *December 5, 1994:* Intel claims the flaw happens once in 27,000 years for the typical spreadsheet user. Intel assumes a user does 1000 divides per day and multiplies the error rate assuming floating-point numbers are random, which is one in 9 billion, and then gets 9 million days, or 27,000 years.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000059

Things begin to calm down, despite Intel neglecting to explain why a typical customer would access floating-point numbers randomly.

- *December 12, 1994:* IBM Research Division disputes Intel's calculation of the rate of errors (you can access this article by visiting *www.mkp.com/books_catalog/cod/links.htm*). IBM claims that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. IBM assumes 5000 divides per second, for 15 minutes, yielding 4.2 million divides per day, and does not assume random distribution of numbers, instead calculating the chances as one in 100 million. As a result, IBM immediately stops shipment of all IBM personal computers based on the Pentium. Things heat up again for Intel.
- *December 21, 1994:* Intel releases the following, signed by Intel's president, chief executive officer, chief operating officer, and chairman of the board: "We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer."Analysts estimate that this recall cost Intel $500 million, and Intel employees did not get a Christmas bonus that year.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

In April 1997 another floating-point bug was revealed in the Pentium Pro and Pentium II microprocessors. When the floating-point-to-integer store instructions (`fist`, `fistp`) encounter a negative floating-point number that is too large to fit in a 16- or 32-bit word after being converted to integer, they set the wrong bit in the FPO status word (precision exception instead of invalid operation exception). To Intel's credit, this time they publicly acknowledged the bug and offered a software patch to get around it—quite a different reaction from what they did in 1994.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000060

3.9 Concluding Remarks

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapter 5 discusses exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation are part of the inspiration for the field of numerical analysis.

Over the years, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic and IEEE 754 binary floating-point arithmetic are found in the vast majority of computers sold today. For example, every desktop computer sold since this book was first printed follows these conventions.

A side effect of the stored-program computer is that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, instruction, and so on. It is the instruction that operates on the word that determines its meaning.

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. The next two figures try to make this clear.

Figure 3.24 lists the MIPS instructions covered in this chapter and Chapter 2. We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of Figure 3.25 are the instructions the MIPS processor executes that are not found in Figure 3.24. We call the full set of hardware instructions *MIPS-32*. On the right of Figure 3.25 are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000061

Instruction subset	Integer	Fl. pt.
MIPS core	95%	57%
MIPS arithmetic core	0%	41%
Remaining MIPS-32	5%	2%

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
and	and	R	move from system control (EPC)	mfc0	R
and immediate	andi	I	floating-point add single	add.s	R
or	or	R	floating-point add double	add.d	R
or immediate	ori	I	floating-point subtract single	sub.s	R
nor	nor	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
branch on equal	beq	I	branch on floating-point false	bc1f	I
branch on not equal	bne	I	floating-point compare single	c.x.s	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J	floating-point compare double	c.x.d	R
jump register	jr	R	(x = eq, neq, lt, le, gt, ge)		
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 3.24 The MIPS instruction set covered so far. This book concentrates on the instructions in the left column.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000062

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
shift right arithmetic	sra	R	not (□ *rs*)	not	rd,rs
shift left logical variable	sllv	R	negate *(signed or unsigned)*	negs	rd,rs
shift right logical variable	srlv	R	rotate left	rol	rd,rs,rt
shift right arithmetic variable	srav	R	rotate right	ror	rd,rs,rt
move to Hi	mthi	R	multiply and don't check oflw *(signed or uns.)*	muls	rd,rs,rt
move to Lo	mtlo	R	multiply and check oflw *(signed or uns.)*	mulos	rd,rs,rt
load halfword	lh	I	divide and check overflow	div	rd,rs,rt
load byte	lb	I	divide and don't check overflow	divu	rd,rs,rt
load word left *(unaligned)*	lwl	I	remainder *(signed or unsigned)*	rems	rd,rs,rt
load word right *(unaligned)*	lwr	I	load immediate	li	rd,imm
store word left *(unaligned)*	swl	I	load address	la	rd,addr
store word right *(unaligned)*	swr	I	load double	ld	rd,addr
load linked (*atomic update*)	ll	I	store double	sd	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load word	ulw	rd,addr
move if zero	movz	R			
move if not zero	movn	R	unaligned store word	usw	rd,addr
multiply and add (S or *uns.*)	madds	R			
multiply and subtract (S or *uns.*)	msubs	I	unaligned load halfword *(signed or uns.)*	ulhs	rd,addr
branch on ≥ zero and link	bgezal	I	unaligned store halfword	ush	rd,addr
branch on < zero and link	bltzal	I	branch	b	Label
jump and link register	jalr	R	branch on equal zero	beqz	rs,L
branch compare to zero	bxz	I	branch on compare *(signed or unsigned)*	bxs	rs,rt,L
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			set equal	seq	rd,rs,rt
branch compare reg likely	bxl	I	set not equal	sne	rd,rs,rt
trap if compare reg	tx	R	set on compare *(signed or unsigned)*	sxs	rd,rs,rt
trap if compare immediate	txi	I	(x = lt, le, gt, ge)		
(x = eq, neq, lt, le, gt, ge)			load to floating point *(s or d)*	l.f	rd,addr
return from exception	rfe	R	store from floating point *(s or d)*	s.f	rd,addr
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move *(s or d)*	mov.f	R			
FP move if zero *(s or d)*	movz.f	R			
FP move if not zero *(s or d)*	movn.f	R			
FP square root *(s or d)*	sqrt.f	R			
FP absolute value *(s or d)*	abs.f	R			
FP negate *(s or d)*	neg.f	R			
FP convert *(w, s, or d)*	cvt.f.f	R			
FP compare un *(s or d)*	c.xn.f	R			

FIGURE 3.25 Remaining MIPS-32 and "Pseudo MIPS" instruction sets. *f* means single (s) or double precision (d) floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.*f*/msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*).

(page 227)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000063

Figure 3.26 gives the popularity of the MIPS instructions for SPEC2000 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 1% of the instructions executed. The following table summarizes that information.

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC2000 execution, and the integer core plus arithmetic core dominate SPEC2000 floating point.

For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As we can see, the MIPS core includes the most popular MIPS instructions, and be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers.

Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	add	0%	0%	FP add double	add.d	0%	8%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	7%	21%	FP multiply double	mul.d	0%	8%
add immediate unsigned	addiu	12%	2%	FP divide double	div.d	0%	0%
subtract unsigned	subu	3%	2%	load word to FP double	l.d	0%	15%
and	and	1%	0%	store word to FP double	s.d	0%	7%
and immediate	andi	3%	0%	shift right arithmetic	sra	1%	0%
or	or	7%	2%	load half	lhu	1%	0%
or immediate	ori	2%	0%	branch less than zero	bltz	1%	0%
nor	nor	3%	1%	branch greater or equal zero	bgez	1%	0%
shift left logical	sll	1%	1%	branch less or equal zero	blez	0%	1%
shift right logical	srl	0%	0%	multiply	mul	0%	1%
load upper immediate	lui	2%	5%				
load word	lw	24%	15%				
store word	sw	9%	2%				
load byte	lbu	1%	0%				
store byte	sb	1%	0%				
branch on equal (zero)	beq	6%	2%				
branch on not equal (zero)	bne	5%	1%				
jump and link	jal	1%	0%				
jump register	jr	1%	0%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURE 3.26 The frequency of the MIPS instructions for SPEC2000 integer and floating point. All instructions that accounted for at least 1% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here. This data is from Chapter 2 of *Computer Architecture: A Quantitative Approach*, third edition.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000064

3.10 Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, 1992

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the IA-32. See Section 3.10.

3.11 Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941, *Abroad*, 1869

3.1 [3] <§3.2> Convert 4096_{ten} into a 32-bit two's complement binary number.

3.2 [3] <§3.2> Convert -2047_{ten} into a 32-bit two's complement binary number.

3.3 [5] <§3.2> Convert $-2{,}000{,}000_{ten}$ into a 32-bit two's complement binary number.

3.4 [5] <§3.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000\ 0110_{two}$?

3.5 [5] <§3.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{two}$?

3.6 [5] <§3.2> What decimal number does this two's complement binary number represent: $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{two}$?

3.7 [10] <§3.2> Find the shortest sequence of MIPS instructions to determine the absolute value of a two's complement integer. Convert this instruction (accepted by the MIPS assembler):

```
abs   $t2,$t3
```

This instruction means that register `$t2` has a copy of register `$t3` if register `$t3` is positive, and the two's complement of register `$t3` if `$t3` is negative. (Hint: It can be done with three instructions.)

3.8 [10] <§3.2> For More Practice: Number Representations

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000065

3.9 [10] <§3.2> If A is a 32-bit address, typically an instruction sequence such as

```
lui $t0, A_upper
ori $t0, $t0, A_lower
lw $s0, 0($t0)
```

can be used to load the word at A into a register (in this case, $s0). Consider the following alternative, which is more efficient:

```
lui $t0, A_upper_adjusted
lw $s0, A_lower($t0)
```

Describe how A_upper is adjusted to allow this simpler code to work. (Hint: A_upper needs to be adjusted because A_lower will be sign-extended.)

3.10 [10] <§3.3> Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, registers $t3 and $t4. Place a 0 or 1 in register $t2 if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions.)

3.11 [15] <§3.3> For More Practice: Writing MIPS Code to Perform Arithmetic

3.12 [15] <§3.3> Suppose that all of the conditional branch instructions except beq and bne were removed from the MIPS instruction set along with slt and all of its variants (slti, sltu, sltui). Show how to perform

```
slt $t0, $s0, $s1
```

using the modified instruction set in which slt is not available. (Hint: It requires more than two instructions.)

3.13 [10] <§3.3> Draw the gates for the Sum bit of an adder, given the equation on page B-28.

3.14 [5] <§3.4> For More Practice: Writing MIPS Code to Perform Arithmetic

3.15 [20] <§3.4> For More Practice: Writing MIPS Code to Perform Arithmetic

3.16 [2 weeks] <§3.4> For More Practice: Simulating MIPS Machines

3.17 [1 week] <§3.4> For More Practice: Simulating MIPS Machines

3.18 [5] <§3.4> For More Practice: Carry Lookahead Adders

3.19 [15] <§3.4> For More Practice: Carry Lookahead Adders

3.20 [10] <§3.4> For More Practice: Relative Performance of Adders

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000066

3.21 [15] <§3.4> For More Practice: Relative Performance of Adders

3.22 [15] <§3.4> For More Practice: Relative Performance of Adders

3.23 [30] <§3.4> In More Depth: Booth's Algorithm

3.24 [15][30] <§3.4> For More Practice: Special MIPS Registers

3.25 [10] <§§3.5, 3.4> In More Depth: The Power PC's Multiply-Add Instruction

3.26 [20] <§3.5> In More Depth: The Power PC's Multiply-Add Instruction

3.27 <§§3.3, 3.4, 3.5> With $x = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 1011_{two}$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two}$ representing two's complement signed integers, perform, showing all work:

a. $x + y$

b. $x - y$

c. $x * y$

d. x/y

3.28 [20] <§§3.3, 3.4, 3.5> Perform the same operations as Exercise 3.27, but with $x = 1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{two}$.

3.29 [30] <§3.5> The division algorithm in Figure 3.11 on page 185 is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shift left is the same as multiplying by 2. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \square 2 - \text{Divisor}$$

This value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

$$\text{Remainder} \square 2 + \text{Divisor} \square 2 - \text{Divisor} = \text{Remainder} \square 2 + \text{Divisor}$$

Based on this observation, write a *nonrestoring division* algorithm using the notation of Figure 3.11 that does not add the Divisor to the Remainder in step 3b. Show that your algorithm works by dividing $0000\ 1011_{two}$ by 0011_{two}.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000067

3.30 [15] <§§3.2, 3.6> The Big Picture on page 216 mentions that bits have no inherent meaning. Given the bit pattern:

```
1010 1101 0001 0000 0000 0000 0000 0010
```

what does it represent, assuming that it is

a. a two's complement integer?

b. an unsigned integer?

c. a single precision floating-point number?

d. a MIPS instruction?

You may find Figures 3.20 (page 208) and A.10.2 (page A-50) useful.

3.31 <§§3.2, 3.6> This exercise is similar to Exercise 3.30, but this time use the bit pattern

```
0010 0100 1001 0010 0100 1001 0010 0100
```

3.32 [10] [10] <§3.6> For More Practice: Floating Point Number Representations

3.33 [10] <§3.6> For More Practice: Floating Point Number Representations

3.34 [10] <§3.6> For More Practice: Writing MIPS Code to Perform FP Arithmetic

3.35 [5] <§3.6> Add $2.85_{ten} \square 10^3$ to $9.84_{ten} \square 10^4$, assuming that you have only three significant digits, first with guard and round digits and then without them.

3.36 [5] <§3.6> This exercise is similar to Exercise 3.35, but this time use the numbers $3.63_{ten} \square 10^4$ and $6.87_{ten} \square 10^3$.

3.37 [5] <§3.6> Show the IEEE 754 binary representation for the floating-point number 20_{ten} in single and double precision.

3.38 [5] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 20.5_{ten}.

3.39 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 0.1_{ten}.

3.40 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with the decimal fraction –5/6.

3.41 [10] <§3.6> Suppose we introduce a new instruction that adds three floating-point numbers. Assuming we add them together with a triple adder, with guard, round, and sticky bits, are we guaranteed results within 1 ulp of the results using two distinct add instructions?

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000068

3.42 [15] <§3.6> With $x = 0100\ 0110\ 1101\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$ and $y = 1011\ 1110\ 1110\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. $x * y$

3.43 [15] <§3.6> With $x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$, $y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$, and $z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. (result of a) + z

c. Why is this result counterintuitive?

3.44 [20] <§§3.6, 3.7> The IEEE 754 floating-point standard specifies 64-bit double precision with a 53-bit significand (including the implied 1) and an 11-bit exponent. IA-32 offers an extended precision option with a 64-bit significand and a 16-bit exponent.

a. Assuming extended precision is similar to single and double precision, what is the bias in the exponent?

b. What is the range of numbers that can be represented by the extended precision option?

c. How much greater is this accuracy compared to double precision?

3.45 [5] <§§3.6, 3.7> The internal representation of floating point numbers in IA-32 is 80 bits wide. This contains a 16 bit exponent. However it also advertises a 64 bit significand. How is this possible?

3.46 [10] <§3.7> While the IA-32 allows 80-bit floating-point numbers internally, only 64-bit floating-point numbers can be loaded or stored. Starting with only 64-bit numbers, how many operations are required before the full range of the 80-bit exponents are used? Give an example.

3.47 [25] <§3.8> For More Practice: Floating Point on Algorithms

3.48 [30] <§3.8> For More Practice: Floating Point Rounding Modes

3.49 [30] <§3.8> For More Practice: Denormalized Numbers

3.50 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

3.51 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000069

3.52 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

3.53 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

3.54 [15] <§3.9> For More Practice: Evaluating Performance

3.55 [15] <§3.9> For More Practice: Evaluating Performance

Answers To Check Yourself

§3.2. page 168: 3, since each character in a Java string takes 16 bits plus one word for length.
§3.3, page 174: 2.
§3.6, page 217: 3.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000070

This Page Intentionally Left Blank

Computers in the Real World

Reconstructing the Ancient World

Problem: Analyzing and understanding archeological sites is challenging. Can we find ways to use computers to help researchers to explore archeological sites and artifacts discovered at those sites?

Solution: Archeology is undergoing a revolution with the use of digital tools for mapping out ancient sites, reconstructing damaged artifacts, and recreating ancient sites in three dimensions. Among the important new techniques being used to analyze and recreate sites are the following:

The use of geographical information systems (GIS) to help accurately measure sites. GIS uses global positioning systems (GPS) to accurately pinpoint locations, allowing fast and precise measurements of a site.

Laser range finding to obtain accurate measurements of the two- and three-dimensional structure of objects. Laser range-finding is even being used with low-flying aircraft to obtain height measurements.

A digital photograph taken from a virtual reality model of the new temple at Chavín de Huántar.

Digital photography to obtain accurate images of sites as well as individual objects.

Virtual reality and three-dimensional visualization systems can use digital photographic data and accurate geospatial information to create realistic versions of archeological sites, allowing archeolo-

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000072

gists to gain new insights, as well as to share their work with other researchers and the general public.

These techniques have been used to explore and create an interactive virtual reality model of an archeological site called Chavín de Huántar, which is in the Peruvian highlands. The image on the top of the previous page is a digital still photograph taken from the virtual reality model. Chavín de Huántar was occupied from about 1000 BCE and predates the classical Incan civilization by more than 1000 years. Highly detailed photos together with measurements of over 25,000 points allows a reconstruction of an accurate virtual model. The image on the top of the previous page is from the new temple at Chavín, which played a key role in the establishment of formalized religious authority in the New World.

Three-dimensional modeling and reconstruction have also been used in the reconstruction of artifacts from fragments. The images on the left below are fragments of pottery found at Petra, the famous archeological site in Jordan. On the right is a computer reconstruction of the original vessel, highlighting the position of one of the fragments.

Images of pottery fragments found at Petra, Jordan

To learn more see these references on the library

Reconstructing objects from fragments at the SHAPE Laboratory at Brown University

The Chavín de Huántar exploration (includes virtual reality tour of the site)

A computer reconstruction from the fragments in the previous photo

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000073

4

Assessing and Understanding Performance

Time discovers truth.

Seneca

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000074

4.1 **Introduction** 240

4.2 **CPU Performance and Its Factors** 246

4.3 **Evaluating Performance** 254

4.4 **Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors** 259

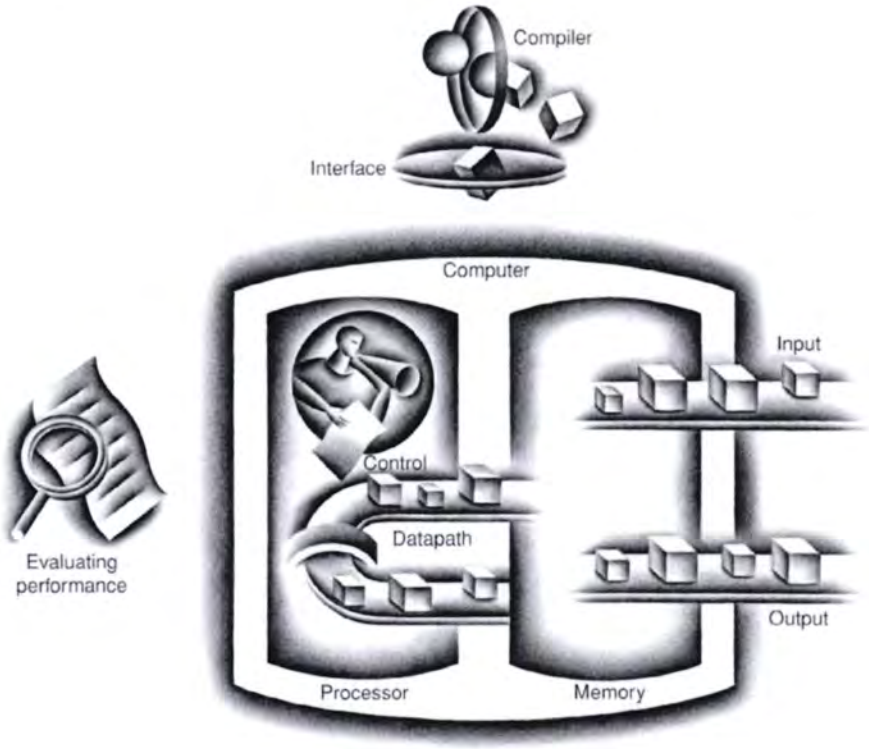
4.5 **Fallacies and Pitfalls** 266

4.6 **Concluding Remarks** 270

4.7 **Historical Perspective and Further Reading** 272

4.8 **Exercises** 272

The Five Classic Components of a Computer



4.1 Introduction

This chapter discusses how to measure, report, and summarize performance and describes the major factors that determine the performance of a computer. A primary reason for examining performance is that hardware performance is often key to the effectiveness of an entire system of hardware and software. Much of the material in this chapter, especially this section and the next, will be critical to understanding the next three chapters. The rest of the chapter provides important insights and principles that anyone seeking to evaluate the performance of a computer system should know. The material in Sections 4.3–4.5, however, is not necessary for the immediately following chapters and can be returned to later.

Assessing the performance of such a system 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. It is simply impossible to sit down with an instruction set manual and a significant software system and determine, simply by analysis, how fast the software will run on the computer. In fact, for different types of applications, different performance metrics may be appropriate, and different aspects of a computer system may be the most significant in determining overall performance.

Of course, in trying to choose among different computers, performance is almost always 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. In some cases, claims are made about computers that don't provide useful insight for any real applications. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a computer.

Our interest in performance, however, goes beyond issues of assessing performance only from the outside of a computer. To understand why a piece of software performs as it does, why one instruction set can be implemented to perform better than another, or how some hardware feature affects performance, we need to understand what determines the performance of a computer. For example, to improve the performance of a software system, we may need to understand what factors in the hardware contribute to the overall system performance and the relative importance of these factors. These factors may include how well the program uses the instructions of the computer, how well the underlying hardware

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000076

implements the instructions, and how well the memory and I/O systems perform. Understanding how to determine the performance impact of these factors is crucial to understanding the motivation behind the design of particular aspects of the computer, as we will see in the chapters that follow.

The rest of this section describes different ways in which performance can be determined. In Section 4.2, 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. Sections 4.3 and 4.4 describe how best to choose benchmarks to evaluate computers and how to accurately summarize the performance of a group of programs. Section 4.4 also describes one set of commonly used CPU benchmarks and examines measurements for a variety of Intel processors using those benchmarks. Finally, in Section 4.5, we'll examine some of the many pitfalls that have trapped designers and those who analyze and report performance.

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 4.1 shows 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 is the Concorde, the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

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, how-

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x 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 4.1 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).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000077

ever, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different 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 data center 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**. Data center managers are often interested in increasing *throughput*—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 desktop computers versus servers, and embedded computers require yet other metrics and applications. We will see examples of this in section 4.4 when we look at different SPEC benchmarks: one meant to measure CPU performance (SPEC CPU) and one meant to measure Web server performance (SPECweb99).

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.

EXAMPLE

Throughput and Response Time

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 World Wide 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. (In Chapter 8, on input/output systems, we will discuss throughput-related measures.) 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:

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000078

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

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

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

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 faster than Y, then the execution time on Y is n times longer than 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 faster than 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 faster than B.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000079

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 *faster than* 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."

Elaboration: Performance in embedded systems is often characterized by real-time constraints: that is, certain application-specific events must occur within a limited amount of time. There are two common types of real-time constraints: hard real time and soft real time. Hard real time defines a fixed bound on the time to respond to or process some event. For example, the embedded processor that controls an antilock brake system must respond within a hard limit from the time it receives a signal that the wheels are locked. In soft real-time systems, an average response or a response within a limited time to a large fraction of the events suffices. For example, handling video frames in a DVD playback system would be an example of a soft real-time constraint, since dropping a frame is permissible, if it happens very rarely. In embedded real-time applications, once the response-time performance constraint is met, designers often optimize throughput or try to reduce cost.

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. But 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.

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

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 that the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000080

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 among operating systems.

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 use *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 to CPU time measurements.

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 runs at a constant rate and 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., 0.25 nanoseconds, 0.25 ns, 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 section, 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, 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.

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, and 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 look for the 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.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000081

Check Yourself

1. Suppose we know that an application that uses both a desktop client and a remote server 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 client and the server, 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 4 times better than the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

4.2 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 seen 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:

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

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

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

This formula makes it clear that the hardware designer can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program. As we will see in this chapter and later in Chapters 5, 6, and 7, the designer often faces a trade-off between the number of clock cycles

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000082

needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles also increase the clock cycle time.

Improving Performance

EXAMPLE

Our favorite program runs in 10 seconds on computer A, which has a 4 GHz clock. We are trying to help a computer designer build a computer, B, that 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:

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

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 4 \times 10^9 \frac{\text{cycles}}{\text{second}} = 40 \times 10^9 \text{cycles}$$

CPU time for B can be found using this equation:

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

$$6 \text{ seconds} = \frac{1.2 \times 40 \times 10^9 \text{cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 40 \times 10^9 \text{cycles}}{6 \text{ seconds}} = \frac{8 \times 10^9 \text{cycles}}{\text{second}} = 8 \text{ GHz}$$

computer B must therefore have twice the clock rate of A to run the program in 6 seconds.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000083

Hardware Software Interface

The equations in our previous examples do 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

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

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

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 same instruction set architecture, since the instruction count required for a program will, of course, be the same.

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:

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

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000084

Now we can compute the CPU time for each computer:

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

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.

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

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

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

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 on many 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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000085

The BIG Picture

Figure 4.2 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:

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

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 that offsets the improvement in instruction count. Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

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 4.2 The basic components of performance and how each is measured.

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 Chapters 5, 6, and 7), 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.

Designers often obtain CPI by a detailed simulation of an implementation or by using hardware counters, when a CPU is operational. Sometimes it is possible to compute the CPU clock cycles by looking at the different types of instructions and using their individual clock cycle counts. In such cases, the following formula is useful:

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000086

where C_i is the count of the number of instructions of class i executed, CPI_i is the average number of cycles per instruction for that instruction class, and n is the number of instruction classes. Remember that overall CPI for a program will depend on both the number of cycles for each instruction type and the frequency of each instruction type in the program execution.

Understanding Program Performance

As we described in Chapter 1, 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, possibly 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 floating-point operations, 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 complex 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.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000087

EXAMPLE

Comparing Code Segments

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

	CPI for this 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 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. So 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

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

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

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000088

So code sequence 2 is faster, even though it actually 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

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

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2$$

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

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 identical. Several of the exercises ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In the next section, 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.

Two of the major factors that affect CPI are the performance of the pipeline, which is the technique used by all modern processors to execute instructions, and the performance of the memory system. In Chapter 6, we will see how pipeline performance adds to the CPI through stalls, and in Chapter 7 we will see how the performance of the caches can increase the CPI due to stalls in the memory system.

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?

a. $\frac{15 \square 0.6}{1.1} = 8.2\,\text{sec}$

b. $15 \square 0.6 \square 1.1 = 9.9\,\text{sec}$

c. $\frac{15 \square 1.1}{0.6} = 27.5\,\text{sec}$

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000089

4.3 Evaluating Performance

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

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.

Today, it is widely understood that the best type of programs to use for benchmarks are real applications. These may be applications that the user employs regularly or simply applications that are typical. For example, in an environment where the users are primarily engineers, you might use a set of benchmarks containing several typical engineering or scientific applications. If the user community were primarily software development engineers, the best benchmarks would probably include such applications as a compiler or document processing system. Using real applications as benchmarks makes it much more difficult to find trivial ways to speed up the execution of the benchmark. Furthermore, when techniques are found to improve performance, such techniques are much more likely to help other programs in addition to the benchmark.

The use of benchmarks whose performance depends on very small code segments encourages optimizations in either the architecture or compiler that target these segments. The compiler optimizations might recognize special code fragments and generate an instruction sequence that is particularly efficient for this code fragment. Likewise, a designer might try to make some sequence of instructions run especially fast because the sequence occurs in a benchmark. In fact, several companies have introduced compilers with special-purpose optimizations targeted at specific benchmarks. Often these optimizations must be explicitly enabled with a specific compiler option, which would not be used when compiling other programs. Whether the compiler would produce good code, or even *correct* code, if a real application program used these switches, is unclear.

Sometimes in the quest to produce highly optimized code for benchmarks, engineers introduce erroneous optimizations. For example, in late 1995, Intel published a new performance rating for the integer SPEC benchmarks running on a Pentium processor and using an internal compiler, not used outside of Intel.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000090

Unfortunately, the code produced for one of the benchmarks was wrong, a fact that was discovered when a competitor read through the binary to understand how Intel had sped up one of the programs in the benchmark suite so dramatically. In January 1996, Intel admitted the error and restated the performance. Small programs or programs that spend almost all their execution time in a very small code fragment are especially vulnerable to such efforts.

So why doesn't everyone run real programs to measure performance? One reason is that small benchmarks are attractive when beginning a design, since they are small enough to compile and simulate easily, sometimes by hand. They are especially tempting when designers are working on a novel computer because compilers may not be available until much later in the design. Although the use of such small benchmarks early in the design process may be justified, there is no valid rationale for using them to evaluate working computer systems.

As mentioned earlier, different classes and applications of computers will require different types of benchmarks. For desktop computers, the most common benchmarks are either measures of CPU performance or benchmarks focusing on a specific task, such as DVD playback or graphics performance for games. In Section 4.4, we will examine the SPEC CPU benchmarks, which focus on CPU performance and measure response time to complete a benchmark. For servers, the decision of which benchmark to use depends heavily on the nature of the intended application. For scientific servers, CPU-oriented benchmarks with scientific applications are typically used, and response time to complete a benchmark is the metric. For other server environments, benchmarks of Web serving, file serving, and databases are commonly used. These server benchmarks usually emphasize throughput, albeit with possible requirements on response time to individual events, such as a database query or Web page request. Section 4.4 examines the SPECweb99 benchmark designed to test Web server performance. In embedded computing, good benchmarks are much more rare. Often customers use their specific embedded application or a segment of it for benchmarking purposes. The one major benchmark suite developed for embedded computers is EEMBC, and we discuss those benchmarks in the In More Depth section on the CD.

Once we have selected a set of suitable benchmarks and obtained performance measurements, we can write a performance report. The guiding principle in reporting performance measurements should be *reproducibility*—we should list everything another experimenter would need to duplicate the results. This list must include the version of the operating system, compilers, and the input, as well as the computer configuration. As an example, the system description section of a SPEC CPU2000 benchmark report is in Figure 4.3.

One important element of reproducibility is the choice of input. Different inputs can generate quite different behavior. For example, an input can trigger certain execution paths that may be typical, or it may exercise rarely used, and

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000091

Hardware	
Hardware vendor	Dell
Model number	Precision WorkStation 360 (3.2 GHz Pentium 4 Extreme Edition)
CPU	Intel Pentium 4 (800 MHz system bus)
CPU MHz	3200
FPU	Integrated
CPU(s) enabled	1
CPU(s) orderable	1
Parallel	No
Primary cache	12K(I) micro-ops + 8KB(D) on chip
Secondary cache	512KB(I+D) on chip
L3 cache	2048KB(I+D) on chip
Other cache	N/A
Memory	4 x 512MB ECC DDR400 SDRAM CL3
Disk subsystem	1x 80GB ATA/100 7200 RPM
Other hardware	

Software	
Operating system	Windows XP Professional SP1
Compiler	Intel C++ Compiler 7.1 (20030402Z) Microsoft Visual Studio.NET (7.0.9466) MicroQuill SmartHeap Library 6.01
File system type	NTFS
System state	Default

FIGURE 4.3 System description of a desktop system using the fastest Pentium 4 available in 2003. In addition to this formatted mandatory description, there are 23 lines of notes describing special flag settings used for portability (4), optimization (2), tuning (12), base timing (2), a special library (2), and BIOS setting (1).

hence less important, parts of an application. Some of the most important effects from the input set are in the memory system. Larger input sets tend to stress the memory system to a greater extent, and the use of realistically sized workloads in servers both for commercial and scientific applications is critical if a benchmark is intended to predict what real applications may see.

Comparing and Summarizing Performance

Once we have selected programs to use as benchmarks and agreed on whether we are measuring response time or throughput, you might think that performance comparison would be straightforward. However, we must still decide how to summarize the performance of a group of benchmarks. Although summarizing a set of

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1022, Page 000092