**Lecture 1**

During their 50-year life span, computers have become the perfect example of

modern convenience. Living memory is strained to recall the days of steno

pools, carbon paper, and mimeograph machines. It sometimes seems that

these magical computing machines were developed instantaneously in the

form that we now know them. But the developmental path of computers is

paved with accidental discovery, commercial coercion, and whimsical fancy.

And occasionally computers have even improved through the application of

solid engineering practices! Despite all of the twists, turns, and technological

dead ends, computers have evolved at a pace that defies comprehension. We

can fully appreciate where we are today only when we have seen where we’ve

come from.

In the sections that follow, we divide the evolution of computers into generations,

each generation being defined by the technology used to build the machine.

We have provided approximate dates for each generation for reference purposes

only. You will find little agreement among experts as to the exact starting and

ending times of each technological epoch.

Every invention reflects the time in which it was made, so one might wonder

whether it would have been called a computer if it had been invented in the

late 1990s. How much computation do we actually see pouring from the mysterious

boxes perched on or beside our desks? Until recently, computers served us

only by performing mind-bending mathematical manipulations. No longer limited

to white-jacketed scientists, today’s computers help us to write documents,

keep in touch with loved ones across the globe, and do our shopping chores.

Modern business computers spend only a minuscule part of their time performing

accounting calculations. Their main purpose is to provide users with a

bounty of strategic information for competitive advantage. Has the word *computer*

now become a misnomer? An anachronism? What, then, should we call

them, if not computers?

We cannot present the complete history of computing in a few pages. Entire

books have been written on this subject and even they leave their readers wanting

for more detail. If we have piqued your interest, we refer you to look at some of

the books cited in the list of references at the end of this chapter.

**Generation Zero: Mechanical Calculating Machines (1642–1945)**

Prior to the 1500s, a typical European businessperson used an abacus for calculations

and recorded the result of his ciphering in Roman numerals. After the

decimal numbering system finally replaced Roman numerals, a number of people

invented devices to make decimal calculations even faster and more accurate. Wilhelm Schickard (1592–1635) has been credited with the invention of the

first mechanical calculator, the Calculating Clock (exact date unknown). This

device was able to add and subtract numbers containing as many as six digits. In

1642, Blaise Pascal (1623–1662) developed a mechanical calculator called the

Pascaline to help his father with his tax work. The Pascaline could do addition

with carry and subtraction. It was probably the first mechanical adding device

actually used for a practical purpose. In fact, the Pascaline was so well conceived

that its basic design was still being used at the beginning of the twentieth

century, as evidenced by the Lightning Portable Adder in 1908, and the Addometer

in 1920. Gottfried Wilhelm von Leibniz (1646–1716), a noted mathematician,

invented a calculator known as the Stepped Reckoner that could add,

subtract, multiply, and divide. None of these devices could be programmed or

had memory. They required manual intervention throughout each step of their

calculations.

Although machines like the Pascaline were used into the twentieth century,

new calculator designs began to emerge in the nineteenth century. One of the

most ambitious of these new designs was the Difference Engine by Charles Babbage

(1791–1871). Some people refer to Babbage as “the father of computing.”

By all accounts, he was an eccentric genius who brought us, among other things,

the skeleton key and the “cow catcher,” a device intended to push cows and other

movable obstructions out of the way of locomotives.

Babbage built his Difference Engine in 1822. The Difference Engine got its

name because it used a calculating technique called the *method of differences*. The

machine was designed to mechanize the solution of polynomial functions and was

actually a calculator, not a computer. Babbage also designed a general-purpose

machine in 1833 called the Analytical Engine. Although Babbage died before he

could build it, the Analytical Engine was designed to be more versatile than his

earlier Difference Engine. The Analytical Engine would have been capable of performing

any mathematical operation. The Analytical Engine included many of the

components associated with modern computers: an arithmetic processing unit to

perform calculations (Babbage referred to this as the *mill*), a memory (the *store*),

and input and output devices. Babbage also included a conditional branching

operation where the next instruction to be performed was determined by the result

of the previous operation. Ada, Countess of Lovelace and daughter of poet Lord

Byron, suggested that Babbage write a plan for how the machine would calculate

numbers. This is regarded as the first computer program, and Ada is considered to

be the first computer programmer. It is also rumored that she suggested the use of

the binary number system rather than the decimal number system to store data.

A perennial problem facing machine designers has been how to get data into

the machine. Babbage designed the Analytical Engine to use a type of punched

card for input and programming. Using cards to control the behavior of a machine

did not originate with Babbage, but with one of his friends, Joseph-Marie

Jacquard (1752–1834). In 1801, Jacquard invented a programmable weaving

loom that could produce intricate patterns in cloth. Jacquard gave Babbage a tapestry

that had been woven on this loom using more than 10,000 punched cards.

To Babbage, it seemed only natural that if a loom could be controlled by cards,

then his Analytical Engine could be as well. Ada expressed her delight with this

idea, writing, “[T]he Analytical Engine weaves algebraical patterns just as the

Jacquard loom weaves flowers and leaves.”

The punched card proved to be the most enduring means of providing input to a

computer system. Keyed data input had to wait until fundamental changes were

made in how calculating machines were constructed. In the latter half of the nineteenth

century, most machines used wheeled mechanisms, which were difficult to

integrate with early keyboards because they were levered devices. But levered

devices could easily punch cards and wheeled devices could easily read them. So a

number of devices were invented to encode and then “tabulate” card-punched data.

The most important of the late-nineteenth-century tabulating machines was the one

invented by Herman Hollerith (1860–1929). Hollerith’s machine was used for

encoding and compiling 1890 census data. This census was completed in record

time, thus boosting Hollerith’s finances and the reputation of his invention. Hollerith

later founded the company that would become IBM. His 80-column punched card,

the *Hollerith card*, was a staple of automated data processing for over 50 years.

**The First Generation: Vacuum Tube Computers (1945–1953)**

Although Babbage is often called the “father of computing,” his machines were

mechanical, not electrical or electronic. In the 1930s, Konrad Zuse (1910–1995)

picked up where Babbage left off, adding electrical technology and other improvements

to Babbage’s design. Zuse’s computer, the Z1, used electromechanical

relays instead of Babbage’s hand-cranked gears. The Z1 was programmable and

had a memory, an arithmetic unit, and a control unit. Because money and resources

were scarce in wartime Germany, Zuse used discarded movie film instead of

punched cards for input. Although his machine was designed to use vacuum tubes,

Zuse, who was building his machine on his own, could not afford the tubes. Thus,

the Z1 correctly belongs in the first generation, although it had no tubes.

Zuse built the Z1 in his parents’ Berlin living room while Germany was at

war with most of Europe. Fortunately, he couldn’t convince the Nazis to buy his

machine. They did not realize the tactical advantage such a device would give

them. Allied bombs destroyed all three of Zuse’s first systems, the Z1, Z2, and

Z3. Zuse’s impressive machines could not be refined until after the war and ended

up being another “evolutionary dead end” in the history of computers.

Digital computers, as we know them today, are the outcome of work done by

a number of people in the 1930s and 1940s. Pascal’s basic mechanical calculator

was designed and modified simultaneously by many people; the same can be said

of the modern electronic computer. Notwithstanding the continual arguments

about who was first with what, three people clearly stand out as the inventors of

modern computers: John Atanasoff, John Mauchly, and J. Presper Eckert.

John Atanasoff (1904–1995) has been credited with the construction of the

first completely electronic computer. The Atanasoff Berry Computer (ABC) was

a binary machine built from vacuum tubes. Because this system was built specifically to solve systems of linear equations, we cannot call it a general-purpose

computer. There were, however, some features that the ABC had in common with

the general-purpose ENIAC (Electronic Numerical Integrator and Computer),

which was invented a few years later. These common features caused considerable

controversy as to who should be given the credit (and patent rights) for the

invention of the electronic digital computer. (The interested reader can find more

details on a rather lengthy lawsuit involving Atanasoff and the ABC in Mollenhoff

[1988].)

John Mauchly (1907–1980) and J. Presper Eckert (1929–1995) were the two

principle inventors of the ENIAC, introduced to the public in 1946. The ENIAC

is recognized as the first all-electronic, general-purpose digital computer. This

machine used 17,468 vacuum tubes, occupied 1,800 square feet of floor space,

weighed 30 tons, and consumed 174 kilowatts of power. The ENIAC had a memory

capacity of about 1,000 information bits (about 20 10-digit decimal numbers)

and used punched cards to store data.

John Mauchly’s vision for an electronic calculating machine was born from

his lifelong interest in predicting the weather mathematically. While a professor

of physics at Ursinus College near Philadelphia, Mauchly engaged dozens of

adding machines and student operators to crunch mounds of data that he believed

would reveal mathematical relationships behind weather patterns. He felt that if

he could have only a little more computational power, he could reach the goal

that seemed just beyond his grasp. Pursuant to the Allied war effort, and with

ulterior motives to learn about electronic computation, Mauchly volunteered for a

crash course in electrical engineering at the University of Pennsylvania’s Moore

School of Engineering. Upon completion of this program, Mauchly accepted a

teaching position at the Moore School, where he taught a brilliant young student,

J. Presper Eckert. Mauchly and Eckert found a mutual interest in building an

electronic calculating device. In order to secure the funding they needed to build

their machine, they wrote a formal proposal for review by the school. They portrayed

their machine as conservatively as they could, billing it as an “automatic

calculator.” Although they probably knew that computers would be able to function

most efficiently using the binary numbering system, Mauchly and Eckert

designed their system to use base 10 numbers, in keeping with the appearance of

building a huge electronic adding machine. The university rejected Mauchly and

Eckert’s proposal. Fortunately, the United States Army was more interested.

During World War II, the army had an insatiable need for calculating the trajectories

of its new ballistic armaments. Thousands of human “computers” were

engaged around the clock cranking through the arithmetic required for these firing

tables. Realizing that an electronic device could shorten ballistic table calculation

from days to minutes, the army funded the ENIAC. And the ENIAC did indeed

shorten the time to calculate a table from 20 hours to 30 seconds. Unfortunately,

the machine wasn’t ready before the end of the war. But the ENIAC had shown

that vacuum tube computers were fast and feasible. During the next decade, vacuum

tube systems continued to improve and were commercially successful.

**The Second Generation: Transistorized Computers (1954–1965)**

The vacuum tube technology of the first generation was not very dependable. In

fact, some ENIAC detractors believed that the system would never run because

the tubes would burn out faster than they could be replaced. Although system reliability

wasn’t as bad as the doomsayers predicted, vacuum tube systems often

experienced more downtime than uptime.

In 1948, three researchers with Bell Laboratories—John Bardeen, Walter Brattain,

and William Shockley—invented the transistor. This new technology not only

revolutionized devices such as televisions and radios, but also pushed the computer

industry into a new generation. Because transistors consume less power than vacuum

tubes, are smaller, and work more reliably, the circuitry in computers consequently

became smaller and more reliable. Despite using transistors, computers of this generation

were still bulky and quite costly. Typically only universities, governments, and

large businesses could justify the expense. Nevertheless, a plethora of computer

makers emerged in this generation; IBM, Digital Equipment Corporation (DEC),

and Univac (now Unisys) dominated the industry. IBM marketed the 7094 for scientific

applications and the 1401 for business applications. DEC was busy manufacturing

the PDP-1. A company founded (but soon sold) by Mauchly and Eckert built the

Univac systems. The most successful Unisys systems of this generation belonged to

its 1100 series. Another company, Control Data Corporation (CDC), under the supervision

of Seymour Cray, built the CDC 6600, the world’s first supercomputer. The

$10 million CDC 6600 could perform 10 million instructions per second, used 60-bit words, and had an astounding 128 kilowords of main memory.

**The Third Generation: Integrated Circuit Computers (1965–1980)**

The real explosion in computer use came with the integrated circuit generation.

Jack Kilby invented the integrated circuit (IC) or *microchip*, made of germanium.

Six months later, Robert Noyce (who had also been working on integrated circuit

design) created a similar device using silicon instead of germanium. This is the

silicon chip upon which the computer industry was built. Early ICs allowed

dozens of transistors to exist on a single silicon chip that was smaller than a single

“discrete component” transistor. Computers became faster, smaller, and

cheaper, bringing huge gains in processing power. The IBM System/360 family

of computers was among the first commercially available systems to be built

entirely of solid-state components. The 360 product line was also IBM’s first

offering where all of the machines in the family were compatible, meaning they

all used the same assembly language. Users of smaller machines could upgrade to

larger systems without rewriting all of their software. This was a revolutionary

new concept at the time.

The IC generation also saw the introduction of time-sharing and multiprogramming

(the ability for more than one person to use the computer at a time).

Multiprogramming, in turn, necessitated the introduction of new operating systems

for these computers. Time-sharing minicomputers such as DEC’s PDP-8 and

PDP-11 made computing affordable to smaller businesses and more universities.

**The Fourth Generation: VLSI Computers (1980–????)**

In the third generation of electronic evolution, multiple transistors were integrated

onto one chip. As manufacturing techniques and chip technologies

advanced, increasing numbers of transistors were packed onto one chip. There are

now various levels of integration: SSI (small scale integration), in which there are

0 to 100 components per chip; MSI (medium scale integration), in which there

are 100 to 1,000 components per chip; LSI (large scale integration), in which

there are 1,000 to 10,000 components per chip; and finally, VLSI (very large

scale integration), in which there are more than 10,000 components per chip. This

last level, VLSI, marks the beginning of the fourth generation of computers.

To give some perspective to these numbers, consider the ENIAC-on-a-chip

project. In 1997, to commemorate the fiftieth anniversary of its first public

demonstration, a group of students at the University of Pennsylvania constructed

a single-chip equivalent of the ENIAC. The 1,800 square-foot, 30-ton beast that

devoured 174 kilowatts of power the minute it was turned on had been reproduced

on a chip the size of a thumbnail. This chip contained approximately

174,569 transistors—an order of magnitude fewer than the number of components

typically placed on the same amount of silicon in the late 1990s.

VLSI allowed Intel, in 1971, to create the world’s first microprocessor, the

4004, which was a fully functional, 4-bit system that ran at 108KHz. Intel also

introduced the random access memory (RAM) chip, accommodating four kilobits

of memory on a single chip. This allowed computers of the fourth generation to

become smaller and faster than their solid-state predecessors.

VLSI technology, and its incredible shrinking circuits, spawned the development

of microcomputers. These systems were small enough and inexpensive

enough to make computers available and affordable to the general public. The

premiere microcomputer was the Altair 8800, released in 1975 by the Micro

Instrumentation and Telemetry (MITS) corporation. The Altair 8800 was soon

followed by the Apple I and Apple II, and Commodore’s PET and Vic 20. Finally,

in 1981, IBM introduced its PC (Personal Computer).

The Personal Computer was IBM’s third attempt at producing an “entry-level”

computer system. Its Datamaster as well as its 5100 Series desktop computers

flopped miserably in the marketplace. Despite these early failures, IBM’s John Opel

convinced his management to try again. He suggested forming a fairly autonomous

“independent business unit” in Boca Raton, Florida, far from IBM’s headquarters in

Armonk, New York. Opel picked Don Estridge, an energetic and capable engineer,

to champion the development of the new system, code-named the Acorn. In light of

IBM’s past failures in the small-systems area, corporate management held tight rein

on the Acorn’s timeline and finances. Opel could get his project off of the ground

only after promising to deliver it within a year, a seemingly impossible feat.

Estridge knew that the only way that he could deliver the PC within the

wildly optimistic 12-month schedule would be to break with IBM convention and

use as many “off-the-shelf” parts as possible. Thus, from the outset, the IBM PC

was conceived with an “open” architecture. Although some people at IBM may

have later regretted the decision to keep the architecture of the PC as nonproprietary

as possible, it was this very openness that allowed IBM to set the standard

for the industry. While IBM’s competitors were busy suing companies for copying

their system designs, PC clones proliferated. Before long, the price of “IBM compatible”

microcomputers came within reach for just about every small business. Also, thanks to the clone makers, large numbers of these systems soon

began finding true “personal use” in people’s homes.

IBM eventually lost its microcomputer market dominance, but the genie was

out of the bottle. For better or worse, the IBM architecture continues to be the de

facto standard for micro computing, with each year heralding bigger and faster

systems. Today, the average desktop computer has many times the computational

power of the mainframes of the 1960s.

Since the 1960s, mainframe computers have seen stunning improvements in

price-performance ratios owing to VLSI technology. Although the IBM System/

360 was an entirely solid-state system, it was still a water-cooled, powergobbling

behemoth. It could perform only about 50,000 instructions per second

and supported only 16 megabytes of memory (while usually having *kilobytes* of

physical memory installed). These systems were so costly that only the largest

businesses and universities could afford to own or lease one. Today’s mainframes—

now called “enterprise servers”—are still priced in the millions of dollars,

but their processing capabilities have grown several thousand times over,

passing the billion-instructions-per-second mark in the late 1990s. These systems,

often used as Web servers, routinely support hundreds of thousands of transactions

*per minute!*

The processing power brought by VLSI to supercomputers defies comprehension.

The first supercomputer, the CDC 6600, could perform 10 million instructions

per second, and had 128 kilobytes of main memory. By contrast,

supercomputers of today contain thousands of processors, can address terabytes

of memory, and will soon be able to perform a *quadrillion* instructions per second.

What technology will mark the beginning of the fifth generation? Some say

that the fifth generation will mark the acceptance of parallel processing and the

use of networks and single-user workstations. Many people believe we have

already crossed into this generation. Some people characterize the fifth generation

as being the generation of neural network, DNA, or optical computing systems.

It’s possible that we won’t be able to define the fifth generation until we have

advanced into the sixth or seventh generation, and whatever those eras will bring.

**Moore’s Law**

So where does it end? How small can we make transistors? How densely can we

pack chips? No one can say for sure. Every year, scientists continue to thwart

prognosticators’ attempts to define the limits of integration. In fact, more than

one skeptic raised an eyebrow when, in 1965, Intel founder Gordon Moore

stated, “The density of transistors in an integrated circuit will double every year.”

The current version of this prediction is usually conveyed as “the density of silicon

chips doubles every 18 months.” This assertion has become known as

*Moore’s Law*. Moore intended this postulate to hold for only 10 years. However,

advances in chip manufacturing processes have allowed this law to hold for

almost 40 years (and many believe it will continue to hold well into the 2010s).

Yet, using current technology, Moore’s Law cannot hold forever. There are

physical and financial limitations that must ultimately come into play. At the current rate of miniaturization, it would take about 500 years to put the entire solar

system on a chip! Clearly, the limit lies somewhere between here and there. Cost

may be the ultimate constraint. Rock’s Law, proposed by early Intel capitalist

Arthur Rock, is a corollary to Moore’s law: “The cost of capital equipment to

build semiconductors will double every four years.” Rock’s Law arises from the

observations of a financier who has seen the price tag of new chip facilities escalate

from about $12,000 in 1968 to $12 million in the late 1990s. At this rate, by

the year 2035, not only will the size of a memory element be smaller than an

atom, but it would also require the entire wealth of the world to build a single

chip! So even if we continue to make chips smaller and faster, the ultimate question

may be whether we can afford to build them.

Certainly, if Moore’s Law is to hold, Rock’s Law must fall. It is evident that for

these two things to happen, computers must shift to a radically different technology.

Research into new computing paradigms has been proceeding in earnest during the

last half decade. Laboratory prototypes fashioned around organic computing, superconducting,

molecular physics, and quantum computing have been demonstrated.

Quantum computers, which leverage the vagaries of quantum mechanics to solve

computational problems, are particularly exciting. Not only would quantum systems

compute exponentially faster than any previously used method, they would

also revolutionize the way in which we define computational problems. Problems

that today are considered ludicrously infeasible could be well within the grasp of the

next generation’s schoolchildren. These schoolchildren may, in fact, chuckle at our

“primitive” systems in the same way that we are tempted to chuckle at the ENIAC.

**THE COMPUTER LEVEL HIERARCHY**

If a machine is to be capable of solving a wide range of problems, it must be able

to execute programs written in different languages, from FORTRAN and C to

Lisp and Prolog. As we shall see in Chapter 3, the only physical components we

have to work with are wires and gates. A formidable open space—*a semantic*

*gap*—exists between these physical components and a high-level language such

as C++. For a system to be practical, the semantic gap must be invisible to most

of the users of the system.

Programming experience teaches us that when a problem is large, we should

break it down and use a “divide and conquer” approach. In programming, we

divide a problem into modules and then design each module separately. Each

module performs a specific task and modules need only know how to interface

with other modules to make use of them.

Computer system organization can be approached in a similar manner. Through

the principle of abstraction, we can imagine the machine to be built from a hierarchy

of levels, in which each level has a specific function and exists as a distinct hypothetical

machine.We call the hypothetical computer at each level a *virtual machine.* Each

level’s virtual machine executes its own particular set of instructions, calling upon

machines at lower levels to carry out the tasks when necessary. By studying computer

organization, you will see the rationale behind the hierarchy’s partitioning, as well as how these layers are implemented and interface with each other.

**THE VON NEUMANN MODEL**

In the earliest electronic computing machines, programming was synonymous

with connecting wires to plugs. No layered architecture existed, so programming a computer was as much of a feat of electrical engineering as it was an exercise in

algorithm design. Before their work on the ENIAC was complete, John W.

Mauchly and J. Presper Eckert conceived of an easier way to change the behavior

of their calculating machine. They reckoned that memory devices, in the form of

mercury delay lines, could provide a way to store program instructions. This

would forever end the tedium of rewiring the system each time it had a new problem

to solve, or an old one to debug. Mauchly and Eckert documented their idea,

proposing it as the foundation for their next computer, the EDVAC. Unfortunately,

while they were involved in the top secret ENIAC project during World

War II, Mauchly and Eckert could not immediately publish their insight.

No such proscriptions, however, applied to a number of people working at the

periphery of the ENIAC project. One of these people was a famous Hungarian

mathematician named John von Neumann (pronounced *von noy-man*). After reading

Mauchly and Eckert’s proposal for the EDVAC, von Neumann published and

publicized the idea. So effective was he in the delivery of this concept that history

has credited him with its invention. All stored-program computers have come to

be known as *von Neumann systems* using the *von Neumann architecture*.

Although we are compelled by tradition to say that stored-program computers use

the von Neumann architecture, we shall not do so without paying proper tribute to

its true inventors: John W. Mauchly and J. Presper Eckert.

Today’s version of the stored-program machine architecture satisfies at least

the following characteristics:

• Consists of three hardware systems: A *central processing unit* (*CPU*) with a

control unit, an *arithmetic logic unit* (*ALU*), *registers* (small storage areas),

and a program counter; a *main-memory system,* which holds programs that

control the computer’s operation; and an *I/O system.*

• Capacity to carry out sequential instruction processing

• Contains a single path, either physically or logically, between the main memory

system and the control unit of the CPU, forcing alternation of instruction and execution

cycles. This single path is often referred to as the *von Neumann bottleneck*.

Figure 1.4 shows how these features work together in modern computer systems.

Notice that the system shown in the figure passes all of its I/O through the arithmetic

logic unit (actually, it passes through the accumulator, which is part of the

ALU). This architecture runs programs in what is known as the *von Neumann*

*execution cycle* (also called the *fetch-decode-execute cycle*), which describes how

the machine works. One iteration of the cycle is as follows:

**1.** The control unit fetches the next program instruction from the memory, using

the program counter to determine where the instruction is located.

**2.** The instruction is decoded into a language the ALU can understand.

**3.** Any data operands required to execute the instruction are fetched from memory

and placed into registers within the CPU.

**4.** The ALU executes the instruction and places the results in registers or memory.

The ideas present in the von Neumann architecture have been extended so

that programs and data stored in a slow-to-access storage medium, such as a hard

disk, can be copied to a fast-access, volatile storage medium such as RAM prior

to execution. This architecture has also been streamlined into what is currently

called the *system bus model*, which is shown in Figure 1.5. The data bus moves

data from main memory to the CPU registers (and vice versa). The address bus

holds the address of the data that the data bus is currently accessing. The control

bus carries the necessary control signals that specify how the information transfer

is to take place.

Other enhancements to the von Neumann architecture include using index

registers for addressing, adding floating point data, using interrupts and asynchronous

I/O, adding virtual memory, and adding general registers.

**NON–VON NEUMANN MODELS**

Until recently, almost all general-purpose computers followed the von Neumann

design. However, the von Neumann bottleneck continues to baffle engineers looking

for ways to build fast systems that are inexpensive and compatible with the

vast body of commercially available software. Engineers who are not constrained

by the need to maintain compatibility with von Neumann systems are free to use

many different models of computing. A number of different subfields fall into the

non-von Neumann category, including neural networks (using ideas from models

of the brain as a computing paradigm), genetic algorithms (exploiting ideas from

biology and DNA evolution), quantum computation (previously discussed), and

parallel computers. Of these, parallel computing is currently the most popular.

Today, parallel processing solves some of our biggest problems in much the

same way as settlers of the Old West solved their biggest problems using parallel

oxen. If they were using an ox to move a tree and the ox was not big enough

or strong enough, they certainly didn’t try to grow a bigger ox—they used two

oxen. If a computer isn’t fast enough or powerful enough, instead of trying to

develop a faster, more powerful computer, why not simply use multiple computers?

This is precisely what parallel computing does. The first parallel-processing

systems were built in the late 1960s and had only two processors. The

1970s saw the introduction of supercomputers with as many as 32 processors,

and the 1980s brought the first systems with over 1,000 processors. Finally, in

1999, IBM announced the construction of a supercomputer called the Blue

Gene. This massively parallel computer contains over 1 million processors,

each with its own dedicated memory. Its first task is to analyze the behavior of

protein molecules.

Even parallel computing has its limits, however. As the number of processors

increases, so does the overhead of managing how tasks are distributed to

those processors. Some parallel-processing systems require extra processors

just to manage the rest of the processors and the resources assigned to them.

No matter how many processors are placed in a system, or how many

resources are assigned to them, somehow, somewhere, a bottleneck is bound to

develop. The best that we can do to remedy this is to make sure that the slowest

parts of the system are the ones that are used the least. This is the idea

behind *Amdahl’s Law*. This law states that the performance enhancement possible

with a given improvement is limited by the amount that the improved feature is used. The underlying premise is that every algorithm has a sequential part that ultimately limits the speedup that can be achieved by multiprocessor

implementation.

**Lecture 2**

A *computer system* consists of hardware and systems software that work together

to run application programs. Specific implementations of systems change over

time, but the underlying concepts do not. All computer systems have similar

hardware and software components that perform similar functions. This book is

written for programmers who want to get better at their craft by understanding

how these components work and how they affect the correctness and performance

of their programs.

You are poised for an exciting journey. If you dedicate yourself to learning the

concepts in this book, then you will be on your way to becoming a rare “power programmer,”

enlightened by an understanding of the underlying computer system

and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical

errors caused by the way that computers represent numbers. You will learn how

to optimize your C code by using clever tricks that exploit the designs of modern

processors and memory systems. You will learn how the compiler implements

procedure calls and how to use this knowledge to avoid the security holes from

buffer overflow vulnerabilities that plague network and Internet software. You will

learn how to recognize and avoid the nasty errors during linking that confound

the average programmer. You will learn how to write your own Unix shell, your

own dynamic storage allocation package, and even your own Web server. You will

learn the promises and pitfalls of concurrency, a topic of increasing importance as

multiple processor cores are integrated onto single chips.

In their classic text on the C programming language, Kernighan and

Ritchie introduce readers to C using the hello program shown in Figure 1.1.

Although hello is a very simple program, every major part of the system must

work in concert in order for it to run to completion. In a sense, the goal of this

book is to help you understand what happens and why, when you run hello on

your system.

We begin our study of systems by tracing the lifetime of the hello program,

from the time it is created by a programmer, until it runs on a system, prints its

simple message, and terminates. As we follow the lifetime of the program, we will

briefly introduce the key concepts, terminology, and components that come into

play. Later chapters will expand on these ideas.

*code/intro/hello.c*

1 #include <stdio.h>

2

3 int main()

4 {

5 printf("hello, world\n");

6 }

Our hello program begins life as a *source program* (or *source file*) that the

programmer creates with an editor and saves in a text file called hello.c. The

source program is a sequence of bits, each with a value of 0 or 1, organized

in 8-bit chunks called *bytes*. Each byte represents some text character in the

program.

Most modern systems represent text characters using the ASCII standard that

represents each character with a unique byte-sized integer value. For example,

Figure 1.2 shows the ASCII representation of the hello.c program.

The hello.c program is stored in a file as a sequence of bytes. Each byte has

an integer value that corresponds to some character. For example, the first byte

has the integer value 35, which corresponds to the character ‘#’. The second byte

has the integer value 105, which corresponds to the character ‘i’, and so on. Notice

that each text line is terminated by the invisible *newline* character ‘\n’, which is

represented by the integer value 10. Files such as hello.c that consist exclusively

of ASCII characters are known as *text files*. All other files are known as *binary*

*files*.

The representation of hello.c illustrates a fundamental idea: All information

in a system—including disk files, programs stored in memory, user data stored in

memory, and data transferred across a network—is represented as a bunch of bits.

The only thing that distinguishes different data objects is the context in which

we view them. For example, in different contexts, the same sequence of bytes

might represent an integer, floating-point number, character string, or machine

instruction.

The hello program begins life as a high-level C program because it can be read

and understood by human beings in that form. However, in order to run hello.c

on the system, the individual C statements must be translated by other programs

into a sequence of low-level *machine-language* instructions. These instructions are

then packaged in a form called an *executable object program* and stored as a binary

disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed

by a *compiler driver*:

unix> *gcc -o hello hello.c*

. *Preprocessing phase.*The preprocessor (cpp) modifies the original C program

according to directives that begin with the # character. For example, the

#include <stdio.h> command in line 1 of hello.c tells the preprocessor

to read the contents of the system header file stdio.h and insert it directly

into the program text. The result is another C program, typically with the .i

suffix.

. *Compilation phase.* The compiler (cc1) translates the text file hello.i into

the text file hello.s, which contains an *assembly-language program*. Each

statement in an assembly-language program exactly describes one low-level

machine-language instruction in a standard text form. Assembly language is

useful because it provides a common output language for different compilers

for different high-level languages. For example, C compilers and Fortran

compilers both generate output files in the same assembly language.

. *Assembly phase.* Next, the assembler (as) translates hello.s into machine language

instructions, packages them in a form known as a *relocatable object*

*program*, and stores the result in the object file hello.obj. The hello.obj file is

a binary file whose bytes encode machine language instructions rather than

characters. If we were to view hello.o with a text editor, it would appear to

be gibberish.

. *Linking phase.*Notice that our hello program calls the printf function, which

is part of the *standard C library* provided by every C compiler. The printf

function resides in a separate precompiled object file called printf.o, which

must somehow be merged with our hello.o program. The linker (ld) handles

this merging. The result is the hello file, which is an *executable object file* (or

simply *executable*) that is ready to be loaded into memory and executed by

the system. GCC is one of many useful tools developed by the GNU (short for GNU’s Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how

it can be modified or distributed. The GNU project has developed an environment with all the major

components of a Unix operating system, except for the kernel, which was developed separately by

the Linux project. The GNU environment includes the emacs editor, gcc compiler, gdb debugger,

assembler, linker, utilities for manipulating binaries, and other components. The gcc compiler has

grown to support many different languages, with the ability to generate code for many different

machines. Supported languages include C, C++, Fortran, Java, Pascal, Objective-C, and Ada.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open source

movement (commonly associated with Linux) owes its intellectual origins to the GNU project’s

notion of *free software* (“free” as in “free speech,” not “free beer”). Further, Linux owes much of its

popularity to the GNU tools, which provide the environment for the Linux kernel.

For simple programs such as hello.c, we can rely on the compilation system to

produce correct and efficient machine code. However, there are some important

reasons why programmers need to understand how compilation systems work:

. *Optimizing program performance.* Modern compilers are sophisticated tools

that usually produce good code. As programmers, we do not need to know

the inner workings of the compiler in order to write efficient code. However,

in order to make good coding decisions in our C programs, we do need a

basic understanding of machine-level code and how the compiler translates

different C statements into machine code. For example, is a switch statement

always more efficient than a sequence of if-else statements? How much

overhead is incurred by a function call? Is a while loop more efficient than

a for loop? Are pointer references more efficient than array indexes? Why

does our loop run so much faster if we sum into a local variable instead of an

argument that is passed by reference? How can a function run faster when we

simply rearrange the parentheses in an arithmetic expression?

*Understanding link-time errors.* In our experience, some of the most perplexing

programming errors are related to the operation of the linker, especially

when you are trying to build large software systems. For example, what does

it mean when the linker reports that it cannot resolve a reference? What is the

difference between a static variable and a global variable? What happens if

you define two global variables in different C files with the same name? What

is the difference between a static library and a dynamic library? Why does it

matter what order we list libraries on the command line? And scariest of all,

why do some linker-related errors not appear until run time? You will learn

the answers to these kinds of questions in Chapter 7.

. *Avoiding security holes.* For many years, *buffer overflow vulnerabilities* have

accounted for the majority of security holes in network and Internet servers.

These vulnerabilities exist because too few programmers understand the need

to carefully restrict the quantity and forms of data they accept from untrusted

sources. A first step in learning secure programming is to understand the consequences

of the way data and control information are stored on the program

stack. We cover the stack discipline and buffer overflow vulnerabilities in

Chapter 3 as part of our study of assembly language.We will also learn about

methods that can be used by the programmer, compiler, and operating system

to reduce the threat of attack.

**Processors Read and Interpret Instructions**

**Stored in Memory**

At this point, our hello.c source program has been translated by the compilation

system into an executable object file called hello that is stored on disk. To run

the executable file on a Unix system, we type its name to an application program

known as a *shell*:

unix> *./hello*

hello, world

unix>

The shell is a command-line interpreter that prints a prompt, waits for you to type a

command line, and then performs the command. If the first word of the command

line does not correspond to a built-in shell command, then the shell assumes that

it is the name of an executable file that it should load and run. So in this case,

the shell loads and runs the hello program and then waits for it to terminate. The

hello program prints its message to the screen and then terminates. The shell then

prints a prompt and waits for the next input command line.

Buses

Running throughout the system is a collection of electrical conduits called *buses*

that carry bytes of information back and forth between the components. Buses

are typically designed to transfer fixed-sized chunks of bytes known as *words*. The

number of bytes in a word (the *word size*) is a fundamental system parameter that

varies across systems. Most machines today have word sizes of either 4 bytes (32

bits) or 8 bytes (64 bits). For the sake of our discussion here, we will assume a word

size of 4 bytes, and we will assume that buses transfer only one word at a time.

I/O Devices

Input/output (I/O) devices are the system’s connection to the external world. Our

example system has four I/O devices: a keyboard and mouse for user input, a

display for user output, and a disk drive (or simply disk) for long-term storage of

data and programs. Initially, the executable hello program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*.

The distinction between the two is mainly one of packaging. Controllers are chip

sets in the device itself or on the system’s main printed circuit board (often called

the *motherboard*). An adapter is a card that plugs into a slot on the motherboard.

Regardless, the purpose of each is to transfer information back and forth between

the I/O bus and an I/O device.

Main Memory

The *main memory* is a temporary storage device that holds both a program and

the data it manipulates while the processor is executing the program. Physically,

main memory consists of a collection of *dynamic random access memory*(DRAM)

chips. Logically, memory is organized as a linear array of bytes, each with its own

unique address (array index) starting at zero. In general, each of the machine

instructions that constitute a program can consist of a variable number of bytes.

The sizes of data items that correspond to C program variables vary according to

type. For example, on an IA32 machine running Linux, data of type short requires

two bytes, types int, float, and long four bytes, and type double eight bytes.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets

(or *executes*) instructions stored in main memory. At its core is a word-sized

storage device (or *register*) called the *program counter* (PC). At any point in time,

the PC points at (contains the address of) some machine-language instruction in

main memory.1

From the time that power is applied to the system, until the time that the

power is shut off, a processor repeatedly executes the instruction pointed at by the

program counter and updates the program counter to point to the next instruction.

A processor *appears to* operate according to a very simple instruction execution

model, defined by its *instruction set architecture*. In this model, instructions execute

in strict sequence, and executing a single instruction involves performing a series

of steps. The processor reads the instruction from memory pointed at by the

program counter (PC), interprets the bits in the instruction, performs some simple

operation dictated by the instruction, and then updates the PC to point to the next

instruction, which may or may not be contiguous in memory to the instruction that

was just executed.

There are only a few of these simple operations, and they revolve around

main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register

file is a small storage device that consists of a collection of word-sized registers,

each with its own unique name.

**Lecture 3**

Modern computers store and process information represented as 2-valued signals.

These lowly binary digits, or *bits*, form the basis of the digital revolution. The

familiar decimal, or base-10, representation has been in use for over 1000 years,

having been developed in India, improved by Arab mathematicians in the 12th

century, and brought to theWest in the 13th century by the Italian mathematician

Leonardo Pisano (c. 1170 – c. 1250), better known as Fibonacci. Using decimal

notation is natural for ten-fingered humans, but binary values work better when

building machines that store and process information. Two-valued signals can

readily be represented, stored, and transmitted, for example, as the presence or

absence of a hole in a punched card, as a high or low voltage on a wire, or as a

magnetic domain oriented clockwise or counterclockwise. The electronic circuitry

for storing and performing computations on 2-valued signals is very simple and

reliable, enabling manufacturers to integrate millions, or even billions, of such

circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and

apply some *interpretation* that gives meaning to the different possible bit patterns,

however, we can represent the elements of any finite set. For example, using a

binary number system, we can use groups of bits to encode nonnegative numbers.

By using a standard character code, we can encode the letters and symbols in a

document.We cover both of these encodings in this chapter, as well as encodings

to represent negative numbers and to approximate real numbers.

We consider the three most important representations of numbers. *Unsigned*

encodings are based on traditional binary notation, representing numbers greater

than or equal to 0. *Two’s-complement* encodings are the most common way to

represent *signed* integers, that is, numbers that may be either positive or negative.

*Floating-point* encodings are a base-two version of scientific notation for

representing real numbers. Computers implement arithmetic operations, such as

addition and multiplication, with these different representations, similar to the

corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number,

and hence some operations can *overflow* when the results are too large to be represented.

This can lead to some surprising results. For example, on most of today’s

computers (those using a 32-bit representation of data type int), computing the

expression

200 \* 300 \* 400 \* 500

yields −884,901,888. This runs counter to the properties of integer arithmetic—

computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar

properties of true integer arithmetic. For example, multiplication is associative

and commutative, so that computing any of the following C expressions yields

−884,901,888:

(500 \* 400) \* (300 \* 200)

((500 \* 400) \* 300) \* 200

((200 \* 500) \* 300) \* 400

400 \* (200 \* (300 \* 500))

The computer might not generate the expected result, but at least it is consistent!

Floating-point arithmetic has altogether different mathematical properties.

The product of a set of positive numbers will always be positive, although overflow

will yield the special value +∞. Floating-point arithmetic is not associative,

due to the finite precision of the representation. For example, the C expression

(3.14+1e20)-1e20 will evaluate to 0*.*0 on most machines, while 3.14+(1e20-

1e20) will evaluate to 3*.*14. The different mathematical properties of integer

vs. floating-point arithmetic stem from the difference in how they handle the finiteness

of their representations—integer representations can encode a comparatively

small range of values, but do so precisely, while floating-point representations can

encode a wide range of values, but only approximately.

By studying the actual number representations, we can understand the ranges

of values that can be represented and the properties of the different arithmetic

operations. This understanding is critical to writing programs that work correctly

over the full range of numeric values and that are portable across different combinations

of machine, operating system, and compiler.Aswe will describe, a number

of computer security vulnerabilities have arisen due to some of the subtleties of

computer arithmetic. Whereas in an earlier era program bugs would only inconvenience

people when they happened to be triggered, there are now legions of

hackers who try to exploit any bug they can find to obtain unauthorized access

to other people’s systems. This puts a higher level of obligation on programmers

to understand how their programs work and how they can be made to behave in

undesirable ways.

Rather than accessing individual bits in memory, most computers use blocks

of eight bits, or *bytes*, as the smallest addressable unit of memory. A machinelevel

program views memory as a very large array of bytes, referred to as *virtual*

*memory*. Every byte of memory is identified by a unique number, known as its

*address*, and the set of all possible addresses is known as the *virtual address space*.

As indicated by its name, this virtual address space is just a conceptual image

presented to the machine-level program. The actual implementation (presented

in Chapter 9) uses a combination of random-access memory (RAM), disk storage,

special hardware, and operating system software to provide the program with what

appears to be a monolithic byte array.

In subsequent chapters, we will cover how the compiler and run-time system

partitions this memory space into more manageable units to store the different

*program objects*, that is, program data, instructions, and control information.

Various mechanisms are used to allocate and manage the storage for different

parts of the program. This management is all performed within the virtual address

space. For example, the value of a pointer in C—whether it points to an integer,

a structure, or some other program object—is the virtual address of the first byte

of some block of storage. The C compiler also associates *type* information with

each pointer, so that it can generate different machine-level code to access the

value stored at the location designated by the pointer depending on the type of

that value. Although the C compiler maintains this type information, the actual

machine-level program it generates has no information about data types. It simply

treats each program object as a block of bytes, and the program itself as a sequence

of bytes.

Binary notationis too verbose, while with decimal notation, it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or *hexadecimal* numbers.

Hexadecimal (or simply “hex”) uses digits ‘0’ through ‘9’ along with characters

‘A’ through ‘F’ to represent 16 possible values. Figure 2.2 shows the decimal and

binary values associated with the 16 hexadecimal digits. Written in hexadecimal,

the value of a single byte can range from 0016 to FF16.

In C, numeric constants starting with 0x or 0X are interpreted as being in

hexadecimal. The characters ‘A’ through ‘F’ may be written in either upper or

lower case. For example, we could write the number FA1D37B16 as 0xFA1D37B,

as 0xfa1d37b, or even mixing upper and lower case, e.g., 0xFa1D37b.We will use

the C notation for representing hexadecimal values in this book.

A common task in working with machine-level programs is to manually convert

between decimal, binary, and hexadecimal representations of bit patterns.

Converting between binary and hexadecimal is straightforward, since it can be

performed one hexadecimal digit at a time. Digits can be converted by referring

to a chart such as that shown in Figure 2.2. One simple trick for doing the conversion

in your head is to memorize the decimal equivalents of hex digits A, C, and F.

The hex values B, D, and E can be translated to decimal by computing their values

relative to the first three.

Every computer has a *word size*, indicating the nominal size of integer and pointer

data. Since a virtual address is encoded by such a word, the most important system

parameter determined by the word size is the maximum size of the virtual address

space. That is, for a machine with a *w*-bit word size, the virtual addresses can range

from 0 to 2*w* − 1, giving the program access to at most 2*w* bytes.

Most personal computers today have a 32-bit word size. This limits the virtual

address space to 4 gigabytes (written 4 GB), that is, just over 4 × 109 bytes. Although

this is ample space for most applications, we have reached the point where

many large-scale scientific and database applications require larger amounts of

storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly

common as storage costs decrease. As hardware costs drop over time,

even desktop and laptop machines will switch to 64-bit word sizes, and so we will

consider the general case of a *w*-bit word size, as well as the special cases of *w* = 32

and *w* = 64.

Computers and compilers support multiple data formats using different ways to

encode data, such as integers and floating point, as well as different lengths. For

example, many machines have instructions for manipulating single bytes, as well

as integers represented as 2-, 4-, and 8-byte quantities. They also support floating point

numbers represented as 4- and 8-byte quantities.

For ordering the bytes representing an object, there are two common conventions.

Consider a*w*-bit integer having a bit representation [*xw*−1*, xw*−2*, . . . , x*1*, x*0],

where *xw*−1 is the most significant bit and *x*0 is the least. Assuming *w* is a multiple

of 8, these bits can be grouped as bytes, with the most significant byte having bits

[*xw*−1*, xw*−2*, . . . , xw*−8], the least significant byte having bits [*x*7*, x*6*, . . . , x*0], and

the other bytes having bits from the middle. Some machines choose to store the object

in memory ordered from least significant byte to most, while other machines

store them from most to least. The former convention—where the least significant

byte comes first—is referred to as *little endian*. This convention is followed

by most Intel-compatible machines. The latter convention—where the most significant

byte comes first—is referred to as *big endian*. This convention is followed

by most machines from IBM and Sun Microsystems. Note that we said “most.”

The conventions do not split precisely along corporate boundaries. For example,

both IBM and Sun manufacture machines that use Intel-compatible processors

and hence are little endian. Many recent microprocessors are *bi-endian*, meaning

that they can be configured to operate as either little- or big-endian machines.

Continuing our earlier example, suppose the variable x of type int and at

address 0x100 has a hexadecimal value of 0x01234567. The ordering of the bytes

within the address range 0x100 through 0x103 depends on the type of

Note that in the word 0x01234567 the high-order byte has hexadecimal value

0x01, while the low-order byte has value 0x67.

People get surprisingly emotional about which byte ordering is the proper one.

In fact, the terms “little endian” and “big endian” come from the book *Gulliver’s*

*Travels* by Jonathan Swift, where two warring factions could not agree as to how a

soft-boiled egg should be opened—by the little end or by the big. Just like the egg

issue, there is no technological reason to choose one byte ordering convention over

the other, and hence the arguments degenerate into bickering about socio-political

issues. As long as one of the conventions is selected and adhered to consistently,

the choice is arbitrary.

**Lecture 4**

A *number* is an intangible, abstract, concept. It is an intellectual device

that we use to denote quantity. Let’s say I were to tell you that “some book

has one hundred pages.” You could touch the pages — they are tangible. You

could even count those pages to verify that there are one hundred of them.

However, “one hundred” is simply an abstraction that I would be applying to

the book as a way of describing its size.

The important thing to realize is that the following is *not* one hundred:

100

This is nothing more than ink on paper forming certain lines and curves.

You might recognize this sequence of symbols as a representation of

one hundred, but this is not the actual value 100. It’s just three symbols

appearing on this page. It isn’t even the only representation for one

hundred — consider the following, which are all different representations

of the value one hundred:

The representation of a number is (generally) some sequence of symbols.

For example, the common representation of the value one hundred, “100,”

is really a sequence of three numeric digits: the digit *1* followed by the digit *0*

followed by a second *0* digit. Each of these digits has some specific meaning,

but we could have just as easily used the sequence “64” to represent the value

one hundred. Even the individual digits that comprise this representation

of 100 are not numbers. They are numeric digits, tools we use to represent

numbers, but they are not numbers themselves.

Now you may be wondering why we should even care whether a

sequence of symbols like “100” is the actual value one hundred or just the

representation of this value. The reason for this distinction is that you’ll

**100** decimal representation

**C** Roman numeral representation

**6416** base 16/hexadecimal representation

**11001002** base two/binary representation

**1448** base eight/octal representation

**A *numbering system***is a mechanism we use to represent numeric values. In

today’s society, people most often use the *decimal numbering system* (base 10)

and most computer systems use binary representation. Confusion between

the two can lead to poor coding practices. So to write great code, you must

eliminate this confusion.

To appreciate the difference between numbers and their representations,

let’s start with a concrete discussion of the decimal numbering

system. The Arabs developed the decimal numbering system we commonly

use today (indeed, the ten decimal digits are known as *Arabic numerals*).

The Arabic system uses a *positional notation system* to represent values with a

relatively small number of different symbols. That is, the Arabic representation

takes into consideration not only the symbol itself, but the position

of the symbol in a sequence of symbols, a scheme that is far superior toother, nonpositional, representations.

The decimal positional numbering system uses powers of ten and ten

unique symbols for each digit position. Because decimal numbers use powers

of ten, we call such numbers “base-10” numbers. By substituting a different

set of numeric digits and multiplying those digits by powers of some base

other than 10, we can devise a different numbering system to represent our

numbers. The base, or *radix*, is the value that we raise to successive powers for

each digit to the left of the *radix point* (note that the term *decimal point* only

applies to decimal numbers).

In order to allow human beings to work with decimal representation, the

computer has to convert between the decimal notation that humans use and

the binary format that computers use. To appreciate what the computer does

for you, it’s useful to learn how to do these conversions manually.

To convert a binary value to decimal, we add 2*i* for each “1” in the binary

string, where *i* is the zero-based position of the binary digit. For example, the

binary value 110010102 represents:

1 × 27 + 1 × 26 + 0 × 25 + 0 × 24 + 1 × 23 + 0 × 22 + 1 × 21 + 0 × 20

or

128 + 64 + 8 + 2

or

202**10**

To convert decimal to binary is almost as easy. Here’s an algorithm that

converts decimal representation to the corresponding binary representation:

1. If the number is even, emit a zero. If the number is odd, emit a one.

2. Divide the number by two and throw away any fractional component or

remainder.

3. If the quotient is zero, the algorithm is complete.

4. If the quotient is not zero and the number is odd, insert a one before the

current string. If the quotient is not zero and the number is even, prefix

your binary string with zero.

5. Go back to step 2 and repeat.

Generally, only assembly language compilers (“assemblers”) allow the

use of literal binary constants in a program. Because there is a wide variety

of assemblers out there, it should come as no surprise that there are many

different ways to represent binary literal constants in an assembly language

program. Because this text presents examples using MASM and HLA, it

makes sense to adopt the conventions these two assemblers use.

MASM treats any sequence of binary digits (zero and one) that ends with

a “b” or “B” as a binary value. The “b” suffix differentiates binary values like

“1001” and the decimal value of the same form (one thousand and one).

Therefore, the binary representation for nine would be “1001b” in a MASM

source file.

HLA prefixes binary values with the percent symbol *(%)*.

Binary number representation is verbose. Because reading and writing

binary values is awkward, programmers often avoid binary representation in

program source files, preferring hexadecimal notation. Hexadecimal

representation offers two great features: it’s very compact, and it’s easy to

convert between binary and hexadecimal. Therefore, software engineers

generally use hexadecimal representation rather than binary to make their

programs more readable.

Because hexadecimal representation is base 16, each digit to the left of

the hexadecimal point represents some value times a successive power of 16.

Hexadecimal representation uses the letters *A* through *F* for the additional

six digits it requires (above and beyond the ten standard decimal digits, 0–9).

One problem with hexadecimal representation is that it is difficult to

differentiate hexadecimal values like “dead” from standard program

identifiers. Therefore, most programming languages use a special prefix or

suffix character to denote the hexadecimal radix for constants appearing in

your source files. Here’s how you specify literal hexadecimal constants in

several popular languages:

􀀀 The C, C++, C#, Java, and other C-derivative programming languages use

the prefix “0x” to denote a hexadecimal value. Therefore, you’d use the

character sequence “0xdead” for the hexadecimal value DEAD16.

􀀀 The MASM assembler uses an “h” or “H” suffix to denote a hexadecimal

value. This doesn’t completely resolve the ambiguity between certain

identifiers and literal hexadecimal constants; “deadh” still looks like an

identifier to MASM. Therefore, MASM also requires that a hexadecimal

value begin with a numeric digit. So for hexadecimal values that don’t

already begin with a numeric digit, you would add “0” to the beginning

of the value (adding a zero to the beginning of any numeric representation

does not alter the value of that representation). For example, use

“0deadh” to unambiguously represent the hexadecimal value DEAD16.

􀀀 Visual Basic uses the “&H” or “&h” prefix to denote a hexadecimal value.

Continuing with our current example (DEAD16), you’d use “&Hdead” to

represent this hexadecimal value in Visual Basic.

􀀀 Pascal (Delphi/Kylix) uses the symbol *$* as a prefix for hexadecimal

values. So you’d use “$dead” to represent our current example in

Delphi/Kylix.

􀀀 HLA similarly uses the symbol *$* as a prefix for hexadecimal values. So

you’d also use “$dead” to represent DEAD16 with HLA. HLA allows you

to insert underscores into the middle of a hexadecimal number to make

it easier to read, for example “$FDEC\_A012.”

***The Octal (Base-8) Numbering System***

Octal (base-8) representation was common in early computer systems. As a

result, you may still see people use the octal representation now and then.

Octal is great for 12-bit and 36-bit computer systems (or any other size that is

a multiple of three). However, it’s not particularly great for computer

systems whose bit size is some power of two (8-bit, 16-bit, 32-bit, and 64-bit

computer systems). As a result, octal has fallen out of favor over the past

several decades. Nevertheless, some programming languages provide the

ability to specify numeric values in octal notation, and you can still find some

older Unix applications that use octal, so octal is worth discussing here.

**Octal Representation in Programming Languages**

The C programming language (and derivatives like C++ and Java), Visual

Basic, and MASM support octal representation. You should be aware of the

notation various programming languages use for octal numbers in case you

come across it in programs written in these languages.

􀀀 In C, you specify the octal base by prefixing a numeric string with a zero.

For example, “0123” is equivalent to the decimal value 8310 and is definitely

not equivalent to the decimal value 12310.

􀀀 MASM uses a “Q” or “q” suffix to denote an octal number

(Microsoft/Intel probably chose “Q” because it looks like an “O” and

they didn’t want to use “O” or “o” because of the possible confusion with

zero).

􀀀 Visual Basic uses the “&O” (that’s the letter *O*, not a zero) prefix to

denote an octal value. For example, you’d use “&O123” to represent the

decimal value 8310.

Because most programming languages (or their libraries) provide automatic

numeric/string conversions, beginning programmers are often unaware that

this conversion is even taking place. In this section we’ll consider two

conversions: from string to numeric form and from numeric form to string.

Consider how easy it is to convert a string to numeric form in various

languages. In each of the following statements, the variable i can hold some

integer number. The input from the user’s console, however, is a string of

characters. The programming language’s run-time library is responsible for

converting that string of characters to the internal binary form the CPU

requires.

cin >> i; // C++

readln( i ); // Pascal

input i // BASIC

stdin.get(i); // HLA

Because these statements are so easy to use, most programmers don’t

consider the cost of using such statements in their programs. Unfortunately,

if you have no idea of the cost of these statements, you’ll not realize how

they can impact your program when performance is critical. The reason

for exploring these conversion algorithms here is to make you aware of the

work involved so you will not make frivolous use of these conversions.

To simplify things, we’ll discuss unsigned integer values and ignore

the possibility of illegal characters and numeric overflow. Therefore, the

following algorithms actually understate the actual work involved (by a

small amount).

Use this algorithm to convert a string of decimal digits to an integer

value:

1. Initialize a variable with zero; this will hold the final value.

2. If there are no more digits in the string, then the algorithm is complete,

and the variable holds the numeric value.

3. Fetch the next digit (going from left to right) from the string.

4. Multiply the variable by ten, and then add in the digit fetched in step 3.

5. Go to step 2 and repeat.

Converting an integer value to a string of characters takes even more effort.

The algorithm for the conversion is the following:

1. Initialize a string to the empty string.

2. If the integer value is zero, then output a *0*, and the algorithm is

complete.

3. Divide the current integer value by ten, computing the remainder and

quotient.

4. Convert the remainder (always in the range 0..9) to a character, and

concatenate the character to the end of the string.

5. If the quotient is not zero, make it the new value and repeat steps 3–5.

6. Output the characters in the reverse order they were placed into the

string.

The particulars of these algorithms are not important. What is important to

note is that these steps execute once for each output character and division is

very slow. So a simple statement like one of the following can hide a fair

amount of work from the programmer:

printf( "%d", i ); // C

cout << i; // C++

print i // BASIC

write( i ); // Pascal

stdout.put( i ); // HLA

To write great code you don’t need to eschew the use of numeric/string

conversions. They are an important part of computation, and great code will

need to do these conversions. However, a great programmer will be careful

about the use of numeric/string conversions and only use them as necessary.

Note that these algorithms are only valid for unsigned integers. Signed

integers require a little more effort to process (though the extra work is

almost negligible). Floating-point values, however, are far more difficult to

convert between string and numeric form. That’s something to keep in mind

when writing code that uses floating-point arithmetic.

**Internal Numeric Representation**

Most modern computer systems are binary computer systems and, therefore,

use an internal binary format to represent values and other objects. However,

most systems cannot represent just any binary value. Instead, they are only

capable of efficiently representing binary values of a given size. If you want to

write great code, you need to make sure that your programs use data objects

that the machine can represent efficiently. The following sections describe

how computers physically represent values.

***Bits***

The smallest unit of data on a binary computer is a single *bit*. Because a bit

can represent only two different values (typically zero or one) you may get

the impression that there are a very small number of items you can represent

with a single bit. Not true! There are an infinite number of two-item combinations

you can represent with a single bit. Here are some examples (with

arbitrary binary encodings I’ve created):

Zero (0) or one (1)

􀀀 False (0) or true (1)

􀀀 Off (0) or on (1)

􀀀 Male(0) or female (1)

􀀀 Wrong (0) or right (1)

You are *not* limited to representing binary data types (that is, those objects

that have only two distinct values). You could use a single bit to represent any

two distinct items:

􀀀 The numbers 723 (0) and 1,245 (1)

􀀀 The colors red (0) and blue (1)

You could even represent two unrelated objects with a single bit. For

example, you could use the bit value zero to represent the color red and

the bit value one to represent the number 3,256. You can represent *any*

two different values with a single bit. However, you can represent *only two*

different values with a single bit. As such, individual bits aren’t sufficient

for most computational needs. To overcome the limitations of a single bit,

we create *bit strings* from a sequence of multiple bits.

***Bit Strings***

As you’ve already seen in this chapter, by combining bits into a sequence, we

can form binary representations that are equivalent to other representations

of numbers (like hexadecimal and octal). Most computer systems, however,

do not let you collect together an arbitrary number of bits. Instead, you have

to work with strings of bits that have certain fixed lengths. In this section

we’ll discuss some of the more common bit string lengths and the names

computer engineers have given them.

A *nibble* is a collection of four bits. Most computer systems do not provide

efficient access to nibbles in memory. However, nibbles are interesting to us

because it takes exactly one nibble to represent a single hexadecimal digit.

A *byte* is eight bits and it is the smallest addressable data item on many

CPUs. That is, the CPU can efficiently retrieve data on an 8-bit boundary

from memory. For this reason, the smallest data type that many languages

support consumes one byte of memory (regardless of the actual number of

bits the data type requires).

Because the byte is the smallest unit of storage on most machines, and

many languages use bytes to represent objects that require fewer than eight

bits, we need some way of denoting individual bits within a byte. To describe

the bits within a byte, we’ll use *bit numbers*.

The term *word* has a different meaning depending on the CPU. On some

CPUs a word is a 16-bit object. On others a word is a 32-bit or 64-bit object.

In this text, we’ll adopt the 80x86 terminology and define a word to be a

16-bit quantity. Like bytes, we’ll number the bits in a word starting with bit

number zero for the LO bit and work our way up to the HO bit.

A *double word* is exactly what its name implies, a pair of words (sometimes you

will see “double word” abbreviated as “dword”). Therefore, a double-word

quantity is 32 bits long.

Most CPUs efficiently handle objects up to a certain size (typically 32 or 64

bits on contemporary systems). That does not mean you can’t work with

larger objects — it simply becomes less efficient to do so. For that reason,

you typically won’t see programs handling numeric objects much larger

than about 128 or 256 bits. Because 64-bit integers are available in some

programming languages, and most languages support 64-bit floating-point

values, it does make sense to describe a 64-bit data type; we’ll call these *quad*

*words*. Just for the fun of it, we’ll use the term *long word* for 128-bit values. Few

languages today support 128 bit values,2 but this does give us some room to

grow.

Of course, we can break quad words down into 2 double words, 4 words,

8 bytes, or 16 nibbles. Likewise, we can break long words down into 2 quad

words, 4 double words, 8 words, or 16 bytes.

On Intel 80x86 platforms there is also an 80-bit type that Intel calls a *tbyte*

(for 10-byte) object. The 80x86 CPU family uses tbyte variables to hold

extended precision floating-point values and certain binary-coded decimal

(BCD) values.

The binary number 0…000003 represents zero, 0…00001 represents one,

0…00010 represents two, and so on towards infinity. But what about negative

numbers? To represent signed values, most computer systems use the two’s

complement numbering system. The representation of signed numbers

places some fundamental restrictions on those numbers, so it is important to

understand the difference in representation between signed and unsigned

numbers in a computer system to use them efficiently.

With *n* bits we can only represent 2 n different objects. As negative

values are objects in their own right, we’ll have to divide these 2*n* combinations

between negative and non-negative values. So, for example, a

byte can represent the negative values −128..−1 and the non-negative

values 0..127. With a 16-bit word we can represent signed values in the

range −32,768..+32,767. With a 32-bit double word we can represent values

in the range −2,147,483,648..+2,147,483,647. In general, with *n* bits we can

represent the signed values in the range −2n− 1  to +2 n-1 −1.

The two’s complement system uses the HO bit as a *sign bit*. If the HO bit

is zero, the number is non-negative; if the HO bit is one, the number is negative.

To negate a two’s complement number, you can use the following algorithm:

1. Invert all the bits in the number, that is, change all the zeros to ones and

all the ones to zeros.

2. Add one to the inverted result (ignoring any overflow).

**Some Useful Properties of Binary Numbers**

It’s worth learning a few interesting facts about binary values that you might

find useful in your programs. Here are some useful properties:

1. If bit position zero of a binary (integer) value contains one, the number

is an odd number; if this bit contains zero, then the number is even.

2. If the LO *n* bits of a binary number all contain zero, then the number is

evenly divisible by 2*n*.

3. If a binary value contains a one in bit position *n*, and zeros everywhere

else, then that number is equal to 2*n*.

4. If a binary value contains all ones from bit position zero up to (but not

including) bit position *n*, and all other bits are zero, then that value is

equal to 2*n*−1.

5. Shifting all the bits in a number to the left by one position multiplies the

binary value by two.

6. Shifting all the bits of an unsigned binary number to the right by one

position effectively divides that number by two (this does not apply to

signed integer values). Odd numbers are rounded down.

7. Multiplying two *n*-bit binary values together may require as many as 2\**n*

bits to hold the result.

8. Adding or subtracting two *n*-bit binary values never requires more than

*n*+1 bits to hold the result.

9. Inverting all the bits in a binary number (that is, changing all the zeros

to ones and all the ones to zeros) is the same thing as negating (changing

the sign) of the value and then subtracting one from the result.

10. Incrementing (adding one to) the largest unsigned binary value for a

given number of bits always produces a value of zero.

11. Decrementing (subtracting one from) zero always produces the largest

unsigned binary value for a given number of bits.

12. An *n*-bit value provides 2*n* unique combinations of those bits.

13. The value 2*n*−1 contains *n* bits, each containing the value one.

The *binary-coded decimal* (BCD) format, as its name suggests, encodes decimal

values using a binary representation. The 80x86 CPU family provides several

machine instructions that convert between BCD and pure binary formats.

Common general-purpose high-level languages (like C/C++, Pascal, and

Java) rarely support decimal values. However, business-oriented programming

languages (like COBOL and many database languages) support

this data type. So if you’re writing code that interfaces with a database or

some language that supports decimal arithmetic, you may need to deal with

BCD representation.

BCD values consist of a sequence of nibbles, with each nibble representing

a value in the range 0..9. Of course, you can represent values in the

range 0..15 using a nibble; the BCD format, however, uses only 10 of the

possible 16 values. Each nibble represents a single decimal digit in a BCD

value, so with a single byte we can represent values containing two decimal

digits (0..99).

As you can see, BCD storage isn’t particularly efficient. An 8-bit BCD variable

can represent values in the range 0..99 while that same eight bits, holding a

binary value, could represent values in the range 0..255. Likewise, a 16-bit

binary value can represent values in the range 0..65535 while a 16-bit BCD

value can only represent about a sixth of those values (0..9999). Inefficient

storage isn’t the only problem with BCD, though. BCD calculations also tend

to be slower than binary calculations.

At this point, you’re probably wondering why anyone would ever use the

BCD format. The BCD format does have two saving graces: it’s very easy to

convert BCD values between the internal numeric representation and their

decimal string representations, and it’s also very easy to encode multidigit

decimal values in hardware when using BCD — for example, when using a

set of dials with each dial representing a single digit. For these reasons,

you’re likely to see people using BCD in embedded systems (such as toaster

ovens and alarm clocks) but rarely in general-purpose computer software.

A few decades ago people mistakenly thought that calculations involving

BCD (or just decimal) arithmetic were more accurate than binary calculations.

Therefore, they would often perform important calculations, like

those involving dollars and cents (or other monetary units) using decimalbased

arithmetic. While it is true that certain calculations can produce more

accurate results in BCD, this statement is not true in general. Indeed, for

most calculations the binary representation is more accurate. For this reason,

most modern computer programs represent all values (including decimal

values) in a binary form. For example, the Intel 80x86 floating-point unit

(FPU) supports a pair of instructions for loading and storing BCD values.

Internally, however, the FPU converts these BCD values to binary. It only uses

BCD as an external data format (external to the FPU, that is). This generally

produces more accurate results.

**Rational Representation**

One big problem with the fractional representations we’ve seen is that

they are not exact; that is, they provide a close approximation of real

values, but they cannot provide an exact representation for all rational

values.5 For example, in binary or decimal you cannot exactly represent

the value 1/3. You could switch to a ternary (base-3) numbering system and

exactly represent 1/3, but then you wouldn’t be able to exactly represent

fractional values like 1/2 or 1/10. What we need is a numbering system that

can represent any reasonable fractional value. Rational representation is a

possibility in such situations.

Rational representation uses pairs of integers to represent fractional

values. One integer represents the numerator (n) of a fraction, and the

other represents the denominator (d). The actual value is equal to n/d.

As long as n and d are “relatively prime” with respect to one another (that is,

they are not both evenly divisible by the same value) this scheme provides a

good representation for fractional values within the bounds of the integer

representation you’re using for n and d. In theory, arithmetic is quite easy;

you use the same algorithms to add, subtract, multiply, and divide fractional

values that you learned in grade school when dealing with fractions. The only

problem is that certain operations may produce really large numerators or

denominators (to the point where you get integer overflow in these values).

Other than this problem, however, you can represent a wide range of

fractional values using this scheme.

**Lecture 5**

Because computers use binary representation, programmers who write great

code often have to work with binary (and hexadecimal) values. Often, when

writing code, you may need to manually operate on two binary values in order

to use the result in your source code. Although calculators are available

to compute such results, you should be able to perform simple arithmetic

operations on binary operands by hand.

Hexadecimal arithmetic is sufficiently painful that a hexadecimal

calculator belongs on every programmer’s desk (or, at the very least, use a

software-based calculator that supports hexadecimal operations, such as the

Windows calculator). Arithmetic operations on binary values, however, are

actually easier than decimal arithmetic. Knowing how to manually compute

binary arithmetic results is essential because several important algorithms

use these operations (or variants of them). Therefore, the next several

subsections describe how to manually add, subtract, multiply, and divide

binary values, and how to perform various logical operations on them.

***Adding Binary Values***

Adding two binary values is easy; there are only eight rules to learn. (If this

sounds like a lot, just realize that you had to memorize approximately 200

rules for decimal addition!) Here are the rules for binary addition:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 with carry

Carry + 0 + 0 = 1

Carry + 0 + 1 = 0 with carry

Carry + 1 + 0 = 0 with carry

Carry + 1 + 1 = 1 with carry

Once you know these eight rules you can add any two binary values together.

Here are some complete examples of binary addition:

0101

+ 0011

------

Step 1: Add the LO bits (1 + 1 = 0 + carry).

c

0101

+ 0011

------

0

Step 2: Add the carry plus the bits in bit position one (carry + 0 + 1 = 0 +

carry).

c

0101

+ 0011

-------

00

Step 3: Add the carry plus the bits in bit position two (carry + 1 + 0 = 0 +

carry).

c

0101

+ 0011

------

000

Step 4: Add the carry plus the bits in bit position three (carry + 0 + 0 = 1).

0101

+ 0011

------

1000

Here are some more examples:

1100\_1101 1001\_1111 0111\_0111

+ 0011\_1011 + 0001\_0001 + 0000\_1001

----------- ----------- -----------

1\_0000\_1000 1011\_0000 1000\_0000

***Subtracting Binary Values***

Binary subtraction is also easy; like addition, binary subtraction has eight

rules:

0 − 0 = 0

0 − 1 = 1 with a borrow

1 − 0 = 1

1 − 1 = 0

0 − 0 − borrow = 1 with a borrow

0 − 1 − borrow = 0 with a borrow

1 − 0 − borrow = 0

1 − 1 − borrow = 1 with a borrow

Here are some complete examples of binary subtraction:

0101

− 0011

------

Step 1: Subtract the LO bits (1 − 1 = 0).

0101

− 0011

------

0

Step 2: Subtract the bits in bit position one (0 − 1 = 1 + borrow).

0101

− 0011

b

------

10

Step 3: Subtract the borrow and the bits in bit position two (1 − 0 − b = 0).

0101

− 0011

------

010

Step 4: Subtract the bits in bit position three (0 − 0 = 0).

0101

− 0011

------

0010

Here are some more examples:

1100\_1101 1001\_1111 0111\_0111

− 0011\_1011 − 0001\_0001 − 0000\_1001

----------- ----------- ------------------------------------------

1001\_0010 1000\_1110 0110\_1110

***Multiplying Binary Values***

Multiplication of binary numbers is also very easy. It’s just like decimal multiplication

involving only zeros and ones (which is trivial). Here are the rules

you need to know for binary multiplication:

0 × 0 = 0

0 × 1 = 0

1 × 0 = 0

1 × 1 = 1

Using these four rules, multiplication is done the same way you’d do decimal

multiplication (in fact, if you just follow the rules for decimal multiplication

on your binary values you’ll actually get the correct results, because the rules

for decimal multiplication involving the zero and one digits are identical).

Here are some examples of binary multiplication:

1010

× 0101

-------

Step 1: Multiply the LO bit of the multiplier times the multiplicand.

1010

× 0101

-------

1010 (1 × 1010)

Step 2: Multiply bit one of the multiplier times the multiplicand.

1010

× 0101

-------

1010 (1 × 1010)

0000 (0 × 1010)

-------

01010 (partial sum)

Step 3: Multiply bit two of the multiplier times the multiplicand.

1010

× 0101

-------

001010 (previous partial sum)

1010 (1 × 1010)

-------

110010 (partial sum)

Step 4: Multiply bit three of the multiplier times the multiplicand.

1010

× 0101

-------

110010 (previous partial sum)

0000 (0 × 1010)

-------

0110010 (product)

***Dividing Binary Values***

Like multiplication of binary numbers, binary division is actually easier than

decimal division. You use the same (longhand) division algorithm, but

binary division is easier because you can trivially determine whether the

divisor goes into the dividend during each step of the longhand division

algorithm. This algorithm is actually easier in binary because at each step you do not

have to guess how many times 12 goes into the remainder nor do you have to

multiply 12 by your guess to obtain the amount to subtract. At each step in

the binary algorithm, the divisor goes into the remainder exactly zero or one

times.

**Logical Operations on Bits**

There are four main logical operations we’ll need to perform on hexadecimal

and binary numbers: AND, OR, XOR (exclusive-or), and NOT.

Unlike the arithmetic operations, a hexadecimal calculator isn’t necessary

to perform these operations.

The logical AND, OR, and XOR operations accept two single-bit

operands and compute the following results:

**AND:**

0 and 0 = 0

0 and 1 = 0

1 and 0 = 0

1 and 1 = 1

**OR:**

0 or 0 = 0

0 or 1 = 1

1 or 0 = 1

1 or 1 = 1

**XOR:**

0 xor 0 = 0

0 xor 1 = 1

1 xor 0 = 1

1 xor 1 = 0

In plain English, the logical AND operation translates as, “If the first operand

is one and the second operand is one, the result is one; otherwise the result is

zero.” We could also state this as “If either or both operands are zero, the

result is zero.” The logical AND operation is useful for forcing a zero result.

If one of the operands is zero, the result is always zero regardless of the value

of the other operand. If one of the operands contains one, then the result is

the value of the other operand.

Colloquially, the logical OR operation is, “If the first operand or the

second operand (or both) is one, the result is one; otherwise the result is

zero.” This is also known as the *inclusive-OR* operation. If one of the operands

to the logical-OR operation is one, the result is always one. If an operand is

zero, the result is always the value of the other operand.

In English, the logical XOR operation is, “If the first or second operand,

but not both, is one, the result is one; otherwise the result is zero.” If one of

the operands is a one, the result is always the *inverse* of the other operand.

The logical NOT operation is unary (meaning it accepts only one

operand).

The logical functions work on single-bit operands. Because most programming

languages manipulate groups of 8, 16, or 32 bits, we need to

extend the definition of these logical operations beyond single-bit operands.

We can easily extend logical functions to operate on a *bit-by-bit* (or *bitwise*)

basis. Given two values, a bitwise logical function operates on bit zero of

both operands producing bit zero of the result; it operates on bit one of

both operands producing bit one of the result, and so on.

This bit-by-bit execution also applies to the other logical operations, as well.

The ability to force bits to zero or one using the logical AND and OR

operations, and the ability to invert bits using the logical XOR operation, is

very important when working with strings of bits (such as binary numbers).

These operations let you selectively manipulate certain bits within a value

while leaving other bits unaffected. For example, if you have an 8-bit binary

value *X* and you want to guarantee that bits four through seven contain

zeros, you could logically AND the value *X* with the binary value

%0000\_1111. This bitwise logical AND operation would force the HO four

bits of *X* to zero and leave the LO four bits of *X* unchanged. Likewise, you

could force the LO bit of *X* to one and invert bit number two of *X* by logically

ORing *X* with %0000\_0001 and then logically exclusive ORing (XORing) *X*

with %0000\_0100. Using the logical AND, OR, and XOR operations to

manipulate bit strings in this fashion is known as *masking* bit strings. We use

the term *masking* because we can use certain values (one for AND, zero for

OR and XOR) to “mask out” or “mask in” certain bits in an operand while

forcing other bits to zero, one, or their inverse.

Several languages provide operators that let you compute the bitwise

AND, OR, XOR, and NOT of their operands. The C/C++/Java language

family uses the ampersand (&) operator for bitwise AND, the pipe (|) operator

for bitwise OR, the caret (^) operator for bitwise XOR, and the tilde (~)

operator for bitwise NOT. The Visual Basic and Delphi/Kylix languages let

you use the and, or, xor, and not operators with integer operands. From 80x86

assembly language, you can use the AND, OR, NOT, and XOR instructions

to do these bitwise operations.

// Here's a C/C++ example:

i = j & k; // Bitwise AND

i = j | k; // Bitwise OR

i = j ^ k; // Bitwise XOR

i = ~j; // Bitwise NOT

Although bit operations may seem a bit abstract, they are quite useful for

many non-obvious purposes. The following subsections describe some of

their useful properties of using the logical operations in various languages.

**Shifts and Rotates**

Another set of logical operations on bit strings are the *shift* and *rotate* operations.

These functions can be further broken down into *shift lefts, rotate lefts,*

*shift rights,* and *rotate rights.* These operations turn out to be very useful in

many programs.

The shift left operation moves each bit in a bit string one position to the

left. Bit zero moves into bit position one, the previous

value in bit position one moves into bit position two, and so on.

There are two questions that arise: “What goes into bit zero?” and “Where

does the HO bit wind up?” We’ll shift a zero into bit zero, and the previous

value of the HO bit will be the *carry* out of this operation.

Several high-level languages (such as C/C++/C#, Java, and

Delphi/Kylix) provide a shift left operator. In the C language family, this

operator is <<. In Delphi/Kylix, you use the shl operator. Here are some

examples:

// C:

cLang = d << 1; // Assigns d shifted left one position to

// variable "cLang"

// Delphi:

Delphi := d shl 1; // Assigns d shifted left one position to

// variable "Delphi"

Shifting the binary representation of a number one position to the left is

equivalent to multiplying that value by two. Therefore, if you’re using a

programming language that doesn’t provide an explicit shift left operator,

you can usually simulate this by multiplying a binary integer value by two.

Although the multiplication operation is usually slower than the shift left

operation, most compilers are smart enough to translate a multiplication by

a constant power of two into a shift left operation. Therefore, you could write

code like the following in Visual Basic to do a shift left:

vb = d \* 2

A shift right operation is similar to a shift left, except we’re moving the data

in the opposite direction. Bit seven moves into bit six; bit six moves into bit

five; bit five moves into bit four; and so on. During a shift right, we’ll move

a zero into bit seven, and bit zero will be the carry out of the operation.

C, C++, C#, and Java use the >> operator for a shift right

operation. Delphi/Kylix uses the shr operator. Most assembly languages also

provide a shift right instruction (shr on the 80x86).

Shifting an unsigned binary value right divides that value by two. For

example, if you shift the unsigned representation of 254 ($FE) one place to

the right, you get 127 ($7F), exactly as you would expect. However, if you

shift the 8-bit two’s complement binary representation of −2 ($FE) one

position to the right, you get 127 ($7F), which is *not* correct. To divide a

signed number by two using a shift, we must define a third shift operation:

*arithmetic shift right*. An arithmetic shift right operation does not modify the

value of the HO bit. Figure 3-5 shows the arithmetic shift right operation for

an 8-bit operand.

This generally produces the result you expect for two’s complement signed

operands. For example, if you perform the arithmetic shift right operation

on −2 ($FE), you get −1 ($FF). Note, however, that this operation always

rounds the numbers to the closest integer that is *less than or equal to the actual*

*result*. If you arithmetically shift right −1 ($FF), the result is −1, not zero.

Because −1 is less than zero, the arithmetic shift right operation rounds

towards −1. This is not a “bug” in the arithmetic shift right operation; it just

uses a different (though valid) definition of integer division. The bottom

line, however, is that you probably won’t be able to use a signed division

operator as a substitute for arithmetic shift right in languages that don’t

support arithmetic shift right, because most integer division operators round

towards zero.

One problem with the shift right operation in high-level languages is

that it’s rare for a high-level language to support both the logical shift right

and the arithmetic shift right. Worse still, the specifications for certain

languages leave it up to the compiler’s implementer to decide whether to use

an arithmetic shift right or a logical shift right operation. Therefore, it’s only

safe to use the shift right operator on values whose HO bit will cause both

forms of the shift right operation to produce the same result. If you need to

guarantee that a shift right is a logical shift right or an arithmetic shift right

operation, then you’ll either have to drop down into assembly language or

you’ll have to handle the HO bit manually. Obviously, the high-level code

gets ugly really fast, so a quick in-line assembly statement might be a better

solution if your program doesn’t need to be portable across different CPUs.

The following code demonstrates how to simulate a 32-bit logical shift right

and arithmetic shift right in languages that don’t guarantee the type of shift

they use:

// Written in C/C++, assuming 32-bit integers, logical shift right:

// Compute bit 30.

Bit30 = ((ShiftThisValue & 0x800000000) != 0) ? 0x40000000 : 0;

// Shifts bits 0..30.

ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;

// Merge in Bit #30.

ShiftThisValue = ShiftThisValue | Bit30;

// Arithmetic shift right operation

Bits3031 = ((ShiftThisValue & 0x800000000) != 0) ? 0xC0000000 : 0;

// Shifts bits 0..30.

ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;

// Merge bits 30/31.

ShiftThisValue = ShiftThisValue | Bits3031;

Many assembly languages also provide various rotate instructions that recirculate

bits through an operand by taking the bits shifted out of one end of

the operation and shifting them into the other end of the operand. Few

high-level languages provide this operation; fortunately, you won’t need

it very often. If you do, you can synthesize this operation using the shift

operators available in your high-level language:

// Pascal/Delphi/Kylix Rotate Left, 32-bit example:

// Puts bit 31 into bit 0, clears other bits.

CarryOut := (ValueToRotate shr 31);

ValueToRotate := (ValueToRotate shl 1) or CarryOut;

Assembly language programmers typically have access to a wide variety

of shift and rotate instructions.

**Topic № 6**

Floating-Point Representation in Computer Systems

Floating-point arithmetic is an approximation

of real arithmetic that solves the

major problem with integer data types —

the inability to represernt fractional values.

Although floating-point arithmetic is often slower than integer arithmetic, modern CPUs incorporate well-designed floating-point units, thus reducing the

performance difference between integer and floating point arithmetic.

For this reason, the stigma surrounding floating-point arithmetic has diminished

since the days when floating-point results were computed using software

(rather than hardware). One unfortunate aspect of floating-point’s increasing

popularity is that many programmers do not understand the inherent limitations

of the floating-point format. Floating-point arithmetic is but an approximation

of real arithmetic. The inaccuracies present in this approximation can lead to serious defects in application software if an engineer is not aware of the

problems associated with these approximations. In order to write great

software that produces correct results when using floating-point arithmetic,

programmers must be aware of the machine’s underlying numeric representation

and of how floating-point arithmetic approximates real arithmetic.

Floating-point numbers provide only an approximation of real numbers.

This is because there is an infinite number of possible real values, while

floating-point representation uses a finite number of bits (and, therefore,

can only represent a finite number of different values). When a given

floating-point format cannot exactly represent some real value, the floating point

number must instead use the closest value that it can exactly represent.

This section describes how the floating-point format works so you can better

understand the problems with these approximations.

Consider a couple of problems with integer and fixed-point formats.

Integers, of course, cannot represent any fractional values. Another problem

with most integer representations is that they can only represent values in the

range 0..2n−1 or -2n−1..2n−1−1. Fixed-point formats provide the ability to

represent fractional values, but at the expense of the range of integer values

you can represent. This problem, which the floating-point format solves, is

the issue of *dynamic range*.

Consider a simple 16-bit unsigned fixed-point format that uses 8 bits

for fractional component and 8 bits for the integer component of the

number. The integer component can represent values in the range 0..255,

and the fractional component can represent the values zero and fractions

between 2−8 and 1 (with a resolution of about 2−8). Suppose, now, that for

a string of calculations you only need two bits to represent the fractional

values 0.0, 0.25, 0.5, and 0.75. Unfortunately, the extra six bits in the fractional

part of the number go to waste. Wouldn’t it be nice if we could utilize

those bits in the integer portion of the number to extend its range from

0..255 to 0..16,383? Well, that’s the basic concept behind the floating-point

representation. In a floating-point value, the radix point (binary point) can

freely float between digits in the number as needed. So if in a 16-bit binary

number you only need two bits of precision for the fractional component of

the number, the binary point can float down between bits 1 and 2 in the

number, allowing the format to utilize bits 2 through 15 for the integer

portion. In order to support a floating-point format, the numeric representation

needs one additional field — a field that specifies the position of the

radix point within the number. This extra field is equivalent to the *exponent*

present when using scientific notation.

To represent real numbers, most floating-point formats use some

number of bits to represent a *mantissa* and a smaller number of bits to

represent an *exponent*. The mantissa is a base value, that usually falls within

a limited range (for example, between zero and one). The exponent is a

multiplier that when applied to the mantissa produces values outside this

range. The result of separating the number into these two parts is that

floating-point numbers can only represent numbers with a specific number

of *significant* digits. As you will soon see, if the difference between the smallest

and largest exponent is greater than the number of significant digits in

the mantissa (and it usually is), then the floating-point representation

cannot exactly represent all the integers between the smallest and largest

values the floating-point format can represent.

To easily see the impact of limited-precision arithmetic, we will adopt a

simplified *decimal* floating-point format for our examples. Our floating-point

format will provide a mantissa with three significant digits and a decimal

exponent with two digits. The mantissa and exponents are both signed

values. Note that this particular floating-point representation can approximate all

the values between 0.00 and 9.99 × 1099. However, this format certainly

cannot exactly represent all values in this range (that would take 100 digits

of precision!). To represent a value like 9,876,543,210, the floating-point

format would have to approximate this value with 9.88 × 109 (or

9.88e + 9 in programming language notation, which this book will generally

use from this point forward).

The big advantage of the mantissa/exponent configuration is that a

floating-point format can represent values across a wide range. There is a

subtle disadvantage to this scheme, however: you cannot *exactly* represent as

many different values with a floating-point format as you can with an integer

format. This is because the floating-point format can provide multiple

representations (that is, different bit patterns) for the same value. In the

simplified decimal floating-point format shown in Figure 4-1, for example,

1.00e + 1 and 0.10e + 2 are different representations of the same value. As

there are a finite number of different representations possible (given a finite

number of bits or digits), whenever a single value has two possible

representations, that’s one less different value the format can represent.

Furthermore, the floating-point format, a form of scientific notation,

complicates arithmetic somewhat. When adding and subtracting two

numbers in scientific notation, you must adjust the two values so that their

exponents are the same. For example, when adding 1.23e1 and 4.56e0, you

could convert 4.56e0 to 0.456e1 and then add them. This produces 1.686e1.

Unfortunately, the result does not fit into the three significant digits of our

current format, so we must either *round* or *truncate* the result to three

significant digits. Rounding generally produces the most accurate result, so

let’s round the result to obtain 1.69e1. As you can see, the lack of *precision*

(the number of digits or bits maintained in a computation) affects the

*accuracy* (the correctness of the computation).

In the previous example, we were able to round the result because we

maintained *four* significant digits *during* the calculation. If our floating-point

calculation were limited to three significant digits *during* computation, we

would have had to truncate the last digit of the smaller number, obtaining

1.68e1, which is even less correct. Therefore, to improve the accuracy of

floating-point calculations, it is necessary to use extra digits during the

calculation. These extra digits are known as *guard digits* (or *guard bits* in the

case of a binary format). They greatly enhance accuracy during a long chain

of computations.

The accuracy lost during a single computation usually isn’t bad unless

you are greatly concerned about the accuracy of your computations. However,

if you compute a value that is the result of a sequence of floating-point

operations, the error can *accumulate* and greatly affect the computation itself.

For example, suppose we add 1.23e3 and 1.00e0. Adjusting the numbers so

their exponents are the same before the addition produces 1.23e3 + 0.001e3.

The sum of these two values, even after rounding, is 1.23e3. This might seem

perfectly reasonable to you; after all, if we can only maintain three significant

digits, adding in a small value shouldn’t affect the result. However, suppose

we were to add 1.00e0 to 1.23e3 *ten times*. The first time we add 1.00e0 to

1.23e3 we get 1.23e3. Likewise, we get this same result the second, third,

fourth . . . and tenth time we add 1.00e0 to 1.23e3. On the other hand, had

we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3,

we would obtain a different result, 1.24e3. This is an important thing to know

about limited-precision arithmetic:

*The order of evaluation can affect the accuracy of the result.*

Your results will be better when adding or subtracting numbers if their

relative magnitudes (that is, the sizes of the exponents) are similar. If you

are performing a chain calculation involving addition and subtraction, you

should attempt to group the operations so that you can add or subtract

values whose magnitudes are close to one another before adding or subtracting

values whose magnitudes are not as close.

Another problem with addition and subtraction is that you can wind up

with *false precision*. Consider the computation 1.23e0 − 1.22e0. This produces

0.01e0. Although this is mathematically equivalent to 1.00e − 2, this latter

form suggests that the last two digits are both exactly zero. Unfortunately,

we only have a single significant digit after this computation, which is in the

hundredths place. Indeed, some FPUs or floating-point software packages

might actually insert random digits (or bits) into the LO positions. This

brings up a second important rule concerning limited-precision arithmetic:

*Whenever subtracting two numbers with the same signs or adding two numbers*

*with different signs, the accuracy of the result may be less than the precision available*

*in the floating-point format.*

Multiplication and division do not suffer from these same problems because

you do not have to adjust the exponents before the operation; all you need

to do is add the exponents and multiply the mantissas (or subtract the

exponents and divide the mantissas). By themselves, multiplication and

division do not produce particularly poor results. However, they tend to

exacerbate any accuracy error that already exists in a value. For example, if

you multiply 1.23e0 by 2, when you should be multiplying 1.24e0 by 2, the

result is even less accurate than it was. This brings up a third important rule

when working with limited-precision arithmetic:

*When performing a chain of calculations involving addition, subtraction,*

*multiplication, and division, try to perform the multiplication and division*

*operations first.*

Often, by applying normal algebraic transformations, you can arrange a

calculation so the multiplication and division operations occur first. For

example, suppose you want to compute the following:

x × (y + z)

Normally you would add y and z together and multiply their sum by x.

However, you will get a little more accuracy if you first transform the

previous equation to get the following:

x × y + x × z

This way you can compute the result by performing the multiplications first.1

Multiplication and division have other problems, as well. When multiplying

two very large or very small numbers, it is quite possible for *overflow* or

*underflow* to occur. The same situation occurs when dividing a small number

by a large number, or when dividing a large number by a small number. This

brings up a fourth rule you should attempt to follow when multiplying or

dividing values:

*When multiplying and dividing sets of numbers, try to multiply and divide numbers*

*that have the same relative magnitudes.*

Comparing floating-point numbers is very dangerous. Given the inaccuracies

present in any computation (including converting an input string to a

floating-point value), you should *never* compare two floating-point values to

see if they are equal. In a binary floating-point format, different computations

that produce the same (mathematical) result may differ in their least

significant bits. For example, adding 1.31e0 + 1.69e0 should produce 3.00e0.

Likewise, adding 1.50e0 + 1.50e0 should produce 3.00e0. However, were you

to compare (1.31e0 + 1.69e0) against (1.50e0 + 1.50e0) you might find that

these sums are *not* equal to one another. The test for equality succeeds if and

only if all bits (or digits) in the two operands are the same. Because it is not

necessarily true that two seemingly equivalent floating-point computations

will produce exactly equal results, a straight comparison for equality may fail

when, algebraically, such a comparison should succeed.

The standard way to test for equality between floating-point numbers is

to determine how much error (or tolerance) you will allow in a comparison,

and then check to see if one value is within this error range of the other. The

straightforward way to do this is to use a test like the following:

if( (Value1 >= (Value2 − error)) and (Value1 <= (Value2 + error)) then . . .

A more efficient way to handle this is to use a statement of the form:

if( abs(Value1 − Value2) <= error ) then . . .

You must exercise care when choosing the value for *error*. This should be a

value slightly greater than the largest amount of error that will creep into

your computations. The exact value will depend upon the particular floatingpoint

format you use and the magnitudes of the values you are comparing.

So the final rule is this:

*When comparing two floating-point numbers for equality, always compare the values*

*to see if the difference between two values is less than some small error value.*

Checking two floating-point numbers for equality is a very famous problem,

and almost every introductory programming text discusses this issue. Perhaps

less well known is the fact that comparing for less than or greater than

creates the same problems. Suppose that a sequence of floating-point

calculations produces a result that is only accurate to within plus or minus

error, even though the floating-point representation provides better accuracy

than error suggests. If you compare such a result against some other calculation

computed with less accumulated error, and those two values are very

close to one other, then comparing them for less than or greater than may

produce incorrect results.

For example, suppose that some chain of calculations in our simplified

decimal representation produces the result 1.25, which is only accurate to

plus or minus 0.05 (that is, the real value could be somewhere between 1.20

and 1.30). Also assume that a second chain of calculations produces the

result 1.27, which is accurate to the full precision of our floating-point result

(that is, the actual value, before rounding, is somewhere between 1.265 and

1.275). Now, if we compare the result of the first calculation (1.25) against

the value of the second calculation (1.27), we will find that the first

calculation is less than the result of the second. Unfortunately, given the

inaccuracy present in the first calculation this might not be true. If the

correct result of the first computation happens to be in the range 1.27 to

1.30 (exclusive), then reporting that the first calculation is less than the

second is false. About the only reasonable test is to see if the two values are

within the *error* tolerance of one another. If so, treat the values as equal

(so one wouldn’t be considered less than or greater than the other). If you

determine that the values are not equal to one another within the desired

error tolerance, then you can compare them to see if one value is less than

or greater than the other. This is known as a *miserly approach* to comparing

for less than or greater than (that is, we try to find as few values that are less

than or greater than as possible).

The other possibility is to use an *eager approach* to the comparison. An

eager approach attempts to make the result of the comparison true as often

as possible. Given two values that you want to compare, and an error tolerance

you’re interested in achieving, here’s how you’d eagerly compare the

two values for less than or greater than:

if( A < (B + error)) then Eager\_A\_lessthan\_B;

if( A > (B − error)) then Eager\_A\_greaterthan\_B;

Don’t forget that calculations like (B + error) are subject to their own inaccuracies,

depending on the relative magnitudes of the values B and error, and

the inaccuracy of this calculation may very well affect the final result that you

achieve in the comparison.

There are other problems that can occur when using floating-point

values. This book can only point out some of the major problems and

make you aware that you cannot treat floating-point arithmetic like real

arithmetic — the inaccuracies present in limited-precision arithmetic can

get you into trouble if you are not careful. A good text on numerical analysis

or even scientific computing can help fill in the details that are beyond

the scope of this book. If you are going to be working with floating-point

arithmetic, *in any language*, you should take the time to study the effects of

limited-precision arithmetic on your computations.

**IEEE Floating-Point Formats**

When Intel’s 80x86 designers planned to introduce a floating-point unit

(FPU) for its original 8086 microprocessor, they were smart enough to

realize that the electrical engineers and solid-state physicists who design

chips probably didn’t have the necessary numerical analysis background to

design a good floating-point representation. So Intel went out and hired the

best numerical analyst it could find to design a floating-point format for its

8087 FPU. That person then hired two other experts in the field, and the

three of them (Kahn, Coonan, and Stone) designed Intel’s floating-point

format. They did such a good job designing the KCS Floating-Point Standard

that the IEEE organization used this format as the basis for the IEEE floatingpoint

format.

To handle a wide range of performance and accuracy requirements,

Intel actually introduced *three* floating-point formats: single precision, double

precision, and extended precision. The single- and double-precision formats

Although there is an infinite number of values between one and two, we can

only represent eight million (223) of them because we use a 23-bit mantissa

(the 24th bit is always one). This is the reason for inaccuracy in floating-point

arithmetic — we only have 23 bits of precision in computations involving

single-precision floating-point values.

The mantissa uses a *one’s complement* format rather than two’s complement.

This means that the 24-bit value of the mantissa is simply an unsigned

binary number, and the sign bit, in bit position 31, determines whether that

value is positive or negative. One’s complement numbers have the unusual

property that there are two representations for zero (with the sign bit set or

clear). Generally, this is important only to the person designing the floatingpoint

software or hardware system. We will assume that the value zero always

has the sign bit clear.

than and vice versa). On some CPUs a 32-bit unsigned comparison is much

faster than a 32-bit floating-point comparison. In such situations, it’s

probably worthwhile to do the comparison using integer arithmetic rather

than floating-point arithmetic.

With a 24-bit mantissa, you will get approximately 6 1/2 decimal digits of

precision (one half digit of precision means that the first six digits can all be

in the range 0..9 but the seventh digit can only be in the range 0..x where

x < 9 and is generally close to 5). With an 8-bit excess-127 exponent, the

dynamic range of single-precision floating-point numbers is approximately

2±128 or about 10±38.

Although single-precision floating-point numbers are perfectly suitable

for many applications, the dynamic range is somewhat limited and is unsuitable

for many financial, scientific, and other applications. Furthermore,

during long chains of computations, the limited accuracy of the single

precision format may introduce serious error. For serious calculations,

a floating-point format with more precision is necessary.

***Double- Precision Floating-Point Format***

The double-precision format helps overcome the problems of the singleprecision

floating-point. Using twice the space, the double-precision format

has an 11-bit excess-1,023 exponent and a 53-bit mantissa (including an

implied HO bit of one) plus a sign bit. This provides a dynamic range

of about 10± **308** and 14 1/2 digits of precision, which is sufficient for most

applications.

**Topic № 7**

Character Representation in Computer Systems

Although computers are famous for their “number-crunching” capabilities, the truth is that most computer systems process character data far more often than numbers. Given the importance of character manipulation in modern software, a thorough understanding of character and string data is necessary if you’re going

to write great code.

The term *character* refers to a human or machine-readable symbol that is

typically a nonnumeric entity. In general, a character is any symbol that you

can type on a keyboard or display on a video display. Note that in addition to

alphabetic characters, character data includes punctuation symbols, numeric

digits, spaces, tabs, carriage returns (the ENTER key), other control characters,

and other special symbols.

Most computer systems use a 1- or 2-byte binary sequence to encode the

various characters. Windows and Linux certainly fall into this category,

using the ASCII or Unicode character sets, whose members can all be

represented using 1- or 2-byte binary sequences. The EBCDIC character

set, in use on IBM mainframes and minicomputers, is another example

of a single-byte character code.

***The ASCII Character Set***

The ASCII (American Standard Code for Information Interchange)

character set maps 128 characters to the unsigned integer values 0..127

($0..$7F). Although the exact mapping of characters to numeric values is

arbitrary and unimportant, a standardized mapping allows you to communicate

between programs and peripheral devices. The standard ASCII codes

are useful because nearly everyone uses them. Therefore, if you use the

ASCII code 65 to represent the character *A*, then you know that some

peripheral device (such as a printer) will correctly interpret this value as

the character *A*.

Because the ASCII character set provides only 128 different characters,

an interesting question arises: “What do we do with the additional 128 values

($80..$FF) that we can represent with a byte?” One answer is to ignore those

extra values. That will be the primary approach of this book. Another possibility

is to extend the ASCII character set by an additional 128 characters. Of

course, unless you can get everyone to agree upon one particular extension

of the character set,1 the whole purpose of having a standardized character

set will be defeated. And getting everyone to agree is a difficult task.

Despite some major shortcomings, ASCII data is *the* standard for data

interchange across computer systems and programs. Most programs can

accept ASCII data, and most programs can produce ASCII data. Because you

will probably be dealing with ASCII characters in your programs, it would be

wise to study the layout of the character set and memorize a few key ASCII

codes (such as those for *0*, *A*, *a*, and so on).

The ASCII character set is divided into four groups of 32 characters.

The first 32 characters, ASCII codes $0 through $1F (0 through 31), form

a special set of nonprinting characters called the *control characters*. We call

them control characters because they perform various printer and display

control operations rather than displaying actual symbols. Examples of

control characters include *carriage return*, which positions the cursor at the

beginning of the current line of characters;2 line feed, which moves the

cursor down one line on the output device; and backspace, which moves

the cursor back one position to the left. Unfortunately, different control

characters perform different operations on different output devices. There

is very little standardization among output devices. To find out exactly how

a particular control character affects a particular device, you will need to

consult its manual.

The second group of 32 ASCII character codes comprises various

punctuation symbols, special characters, and the numeric digits. The most

notable characters in this group include the space character (ASCII code

$20) and the numeric digits (ASCII codes $30..$39).

The third group of 32 ASCII characters contains the uppercase alphabetic

characters. The ASCII codes for the characters *A* through *Z* lie in the

range $41..$5A. Because there are only 26 different alphabetic characters,

the remaining six codes hold various special symbols.

The fourth and final group of 32 ASCII character codes represents the

lowercase alphabetic symbols, five additional special symbols, and another

control character (delete). Note that the lowercase character symbols use the

ASCII codes $61..$7A. If you convert the codes for the upper- and lowercase

characters to binary, you will notice that the uppercase symbols differ from

their lowercase equivalents in exactly one bit position.

The only place these two codes differ is in bit five. Uppercase alphabetic

characters always contain a zero in bit five; lowercase alphabetic characters

always contain a one in bit five. You can use this fact to quickly convert an

alphabetic character between upper- and lowercase by simply inverting bit

five. If you have an uppercase character, you can force it to lowercase by

setting bit five to one. If you have a lowercase character and you wish to force

it to uppercase, you can do so by setting bit five to zero.

Bits five and six determine the character’s group . Therefore, you can convert any upper- or lowercase (or special) character to its corresponding control character by setting bits five and six to zero.

Despite the fact that it is a “standard,” simply encoding your data using ASCII

characters does not guarantee compatibility across systems. While it’s true

that an *A* on one machine is most likely an *A* on another system, there is very

little standardization across machines with respect to the use of the control

characters. Indeed, of the 32 control codes in the first group of ASCII codes,

plus the delete code in the last group, there are only 4 control codes

commonly supported by most devices and applications — backspace (BS),

tab, carriage return (CR), and line feed (LF). Worse still, different machines

often use these “supported” control codes in different ways. End-of-line is a

particularly troublesome example. Windows, MS-DOS, CP/M, and other

systems mark end-of-line by the two-character sequence CR/LF. The Apple

Macintosh, and many other systems, mark end-of-line by a single CR

character. Linux, BeOS, and other Unix systems mark end-of-line with a

single LF character.

Attempting to exchange simple text files between such different systems

can be an experience in frustration. Even if you use standard ASCII characters

in all your files on these systems, you will still need to convert the

data when exchanging files between them. Fortunately, such conversions

are rather simple, and many text editors automatically handle files with

different line endings (there are also many available freeware utilities that

will do this conversion for you). Even if you have to do this in your own

software, all that the conversion involves is copying all characters except the

end-of-line sequence from one file to another, and then emitting the new

end-of-line sequence whenever you encounter an old end-of-line sequence

in the input file.

***The EBCDIC Character Set***

Although the ASCII character set is, unquestionably, the most popular

character representation, it is certainly not the only format available. For

example, IBM uses the EBCDIC code on many of its mainframe and minicomputer

lines. Because EBCDIC appears mainly on IBM’s big iron and

you’ll rarely encounter it on personal computer systems, we’ll only consider

it briefly in this book.

EBCDIC (pronounced EB-suh-dic) is an acronym that stands for *Extended*

*Binary Coded Decimal Interchange Code*. If you’re wondering if there was an

unextended version of this character code, the answer is yes. Earlier IBM

systems and keypunch machines used a character set known as BCDIC

*(Binary Coded Decimal Interchange Code)*. This was a character set based on

punched cards and decimal representation (for IBM’s older decimal

machines).

The first thing to note about EBCDIC is that it is not a single character

set; rather, it is a family of character sets. While the EBCDIC character sets

have a common core (for example, the encodings for the alphabetic

characters are usually the same), different versions of EBCDIC (known as

*code pages*) have different encodings for punctuation and special characters.

Because there are a limited number of encodings available in a single byte,

different code pages reuse some of the character encodings for their own

special set of characters. So, if you’re given a file that contains EBCDIC

characters and someone asks you to translate it to ASCII, you’ll quickly

discover that this is not a trivial task.

Before you ever look at the EBCDIC character set, you should first

realize that the forerunner of EBCDIC (BCDIC) was in existence long

before modern digital computers. BCDIC was born on old-fashioned IBM

keypunches and tabulator machines. EBCDIC was simply an extension of

that encoding to provide an extended character set for IBM’s computers.

However, EBCDIC inherited several peculiarities from BCDIC that seem

strange in the context of modern computers. For example, the encodings of

the alphabetic characters are not contiguous. This is probably a direct result

of the fact that the original character encodings really did use a decimal

(BCD) encoding. Originally (in BCD/decimal), the alphabetic characters

probably did have a sequential encoding. However, when IBM expanded the

character set, they used some of the binary combinations that are not present

in the BCD format (values like %1010..%1111). Such binary values appear

between two otherwise sequential BCD values, which explains why certain

character sequences (such as the alphabetic characters) do not use sequential

binary codes in the EBCDIC encoding.

Unfortunately, because of the weirdness of the EBCDIC character set,

many common algorithms that work well on ASCII characters simply don’t

work with EBCDIC. This chapter will not consider EBCDIC beyond a token

mention here or there. However, keep in mind that EBCDIC functional

equivalents exist for most ASCII characters. Check out the IBM literature for

more details.

***Double-Byte Character Sets***

Because of the encoding limitations of an 8-bit byte (which has a maximum

of 256 characters) and the need to represent more than 256 characters,

some computer systems use special codes to indicate that a particular

character consumes two bytes rather than a single byte. Such double-byte

character sets (DBCSs) do not encode every character using 16 bits —

instead, they use a single byte for most character encodings and use twobyte

codes only for certain characters.

A typical double-byte character set utilizes the standard ASCII character

set along with several additional characters in the range $80..$FF. Certain

values in this range are extension codes that tell the software that a second

byte immediately follows. Each extension byte allows the DBCS to support

another 256 different character codes. With three extension values, for

example, the DBCS can support up to 1,021 different characters. You get

256 characters with each of the extension bytes, and you get 253 (256 – 3)

characters in the standard single-byte set (minus three because the three

extension byte values each consume one of the 256 combinations, and they

don’t count as characters).

Back in the days when terminals and computers used memory-mapped

character displays, double-byte character sets weren’t very practical. Hardware

character generators really want each character to be the same size, and

they want to process a limited number of characters. However, as bitmapped

displays with software character generators became prevalent (Windows,

Macintosh, and Unix/XWindows machines), it became possible to process

DBCSs.

Although DBCSs can compactly represent a large number of characters,

they demand more computing resources in order to process text in a DBCS

format. For example, if you have a zero-terminated string containing DBCS

characters (typical in the C/C++ languages), then determining the number

of characters in the string can be considerable work. The problem is that

some characters in the string consume two bytes while most others consume

only one byte. A string length function has to scan byte-by-byte through

each character of the string to locate any extension values that indicate

that a single character consumes two bytes. This extra comparison more

than doubles the time a high-performance string length function takes to

execute. Worse still, many common algorithms that people use to manipulate

string data fail when they apply them to DBCSs. For example, a

common C/C++ trick to step through characters in a string is to either increment

or decrement a pointer to the string using expressions like ++ptrChar

or --ptrChar. Unfortunately, these tricks don’t work with DBCSs. While someone

using a DBCS probably has a set of standard C library routines available

that work properly on DBCSs, it’s also quite likely that other useful character

functions they’ve written (or that others have written) don’t work properly

with the extended characters in a DBCS. For this and other reasons, you’re

far better off using the Unicode character set if you need a standardized

character set that supports more than 256 characters. For all the details,

keep reading.

***The Unicode Character Set***

A while back, engineers at Apple Computer and Xerox realized that their

new computer systems with bitmapped displays and user-selectable fonts

could display far more than 256 different characters at one time. Although

DBCSs were a possibility, those engineers quickly discovered the compatibility

problems associated with double-byte character sets and sought a

different route. The solution they came up with was the Unicode character

set. Unicode has since become an international standard adopted and

supported by nearly every major computer manufacturer and operating

system provider (Mac OS, Windows, Linux, Unix, and many other operating

systems support Unicode).

Unicode uses a 16-bit word to represent each character. Therefore,

Unicode supports up to 65,536 different character codes. This is obviously a

huge advance over the 256 possible codes we can represent with an 8-bit byte.

Furthermore, Unicode is upward compatible from ASCII; if the HO 9 bits3 of

a Unicode character’s binary representation contain zero, then the LO 7 bits

use the standard ASCII code. If the HO 9 bits contain some nonzero value,

then the 16 bits form an extended character code (extended from ASCII,

that is). If you’re wondering why so many different character codes are

necessary, simply note that certain Asian character sets contain 4,096

characters (at least, in their Unicode character subset). The Unicode character

set even provides a set of codes you can use to create an applicationdefined

character set. At the time of this writing, approximately half of the

65,536 possible character codes have been defined; the remaining character

encodings are reserved for future expansion.

Today, many operating systems and language libraries provide excellent

support for Unicode. Microsoft Windows, for example, uses Unicode internally.

4 So operating system calls will actually run faster if you pass them

Unicode strings rather than ASCII strings. (When you pass an ASCII string

to a modern version of Windows, the OS first converts the string from ASCII

to Unicode and then proceeds with the OS API function.) Likewise, whenever

Windows returns a string to an application, that string is in Unicode

form; if the application needs it in ASCII form, then Windows must convert

the string from Unicode to ASCII before returning.

There are two big disadvantages to Unicode, however. First, Unicode

character data requires twice as much memory to represent as ASCII or other

single-byte encodings do. Although machines have far more memory today

(both in RAM and on disk where text files usually reside), doubling the size

of text files, databases, and in-memory strings (such as those for a text editor

or word processor) can have a significant impact on the system. Worse,

because strings are now twice as long, it takes almost twice as many instructions

to process a Unicode string as it does to process a string encoded with

single-byte characters. This means that string functions may run at half the

speed of those functions that process byte-sized character data.5 The second

disadvantage to Unicode is that most of the world’s data files out there are in

ASCII or EBCDIC form, so if you use Unicode within an application, you

wind up spending considerable time converting between Unicode and those

other character sets.

Although Unicode is a widely accepted standard, it still is not seeing

widespread use (though it is becoming more popular every day). Quite soon,

Unicode will hit “critical mass” and really take off. However, that point is still

in the future, so most of the examples in this text will continue to use ASCII

characters. Still, at some point in the not-too-distant future, it wouldn’t be

unreasonable to emphasize Unicode rather than ASCII in a book .

***Character String Formats***

Different languages use different data structures to represent strings. Some

string formats use less memory, others allow faster processing, some are

more convenient to use, and some provide additional functionality for the

programmer and operating system. To better understand the reasoning

behind the design of character strings, it is instructive to look at some

common string representations popularized by various high-level languages.

**Zero-Terminated Strings**

Without question, *zero-terminated strings* are probably the most common

string representation in use today, because this is the native string format

for C, C++, Java, and several other languages. In addition, you’ll find zeroterminated

strings in use in programs written in languages that don’t have

a specific native string format, such as assembly language.

A zero-terminated ASCII string is a sequence containing zero or more

8-bit character codes ending with a byte containing zero (or, in the case of

Unicode, a sequence containing zero or more 16-bit character codes ending

with a 16-bit word containing zero). For example, in C/C++, the ASCII string

“abc” requires four bytes: one byte for each of the three characters *a*, *b*, and *c*,

followed by a zero byte.

Zero-terminated strings have a couple of advantages over other string

formats:

Zero-terminated strings can represent strings of any practical length

with only one byte of overhead (two bytes in Unicode).

Given the popularity of the C/C++ programming languages, highperformance

string processing libraries are available that work well

with zero-terminated strings.

Zero-terminated strings are easy to implement. Indeed, except for dealing

with string literal constants, the C/C++ programming languages

don’t provide native string support. As far as the C and C++ languages

are concerned, strings are just arrays of characters. That’s probably why

C’s designers chose this format in the first place — so they wouldn’t have

to clutter up the language with string operators.

This format allows you to easily represent zero-terminated strings in any

language that provides the ability to create an array of characters.

However, despite these advantages, zero-terminated strings also have

disadvantages — they are not always the best choice for representing

character string data. These disadvantages are as follows:

String functions often aren’t very efficient when operating on zeroterminated

strings. Many string operations need to know the length

of the string before working on the string data. The only reasonable

way to compute the length of a zero-terminated string is to scan the

string from the beginning to the end. The longer your strings are,

the slower this function runs. Therefore, the zero-terminated string

format isn’t the best choice if you need to process long strings.

Though this is a minor problem, with the zero-terminated string format

you cannot easily represent any character whose character code is zero

(such as the ASCII NUL character).

With zero-terminated strings there is no information contained within

the string data itself that tells you how long a string can grow beyond the

terminating zero byte. Therefore, some string functions, like concatenation,

can only extend the length of an existing string variable and check

for overflow if the caller explicitly passes in the maximum length.

**Length-Prefixed Strings**

A second string format, *length-prefixed strings*, overcomes some of the

problems with zero-terminated strings. Length-prefixed strings are common

in languages like Pascal; they generally consist of a single byte that specifies

the length of the string, followed by zero or more 8-bit character codes.

In a length-prefixed scheme, the string “abc” would consist of four bytes:

the length byte ($03), followed by *a*, *b*, and *c*.

Length-prefixed strings solve two of the problems associated with zeroterminated

strings. First, it is possible to represent the NUL character in a

length-prefixed string, and second, string operations are more efficient.

Another advantage to length-prefixed strings is that the length is usually

sitting at position zero in the string (when viewing the string as an array of

characters), so the first character of the string begins at index one in the

array representation of the string. For many string functions, having a onebased

index into the character data is much more convenient than a zerobased

index (which zero-terminated strings use).

Length-prefixed strings do suffer from their own drawbacks, the principal

drawback being that they are limited to a maximum of 255 characters

in length (assuming a 1-byte length prefix). One can remove this limitation

by using a 2- or 4-byte length value, but doing so increases the amount of

overhead data from one to two or four bytes.

**Seven-Bit Strings**

An interesting string format that works for 7-bit codes like ASCII involves

using the HO bit to indicate the end of the string. All but the last character

code in the string would have their HO bit clear (or set, your choice) and the

last character in the string would have its HO bit set (or clear, if all the other

HO bits are set).

This 7-bit string format has several disadvantages:

You have to scan the entire string in order to determine the length

of the string.

You cannot have zero-length strings in this format.

Few languages provide literal string constants for 7-bit strings.

You are limited to a maximum of 128 character codes, though this is

fine when using plain ASCII.

However, the big advantage of 7-bit strings is that they don’t require any

overhead bytes to encode the length. Assembly language (using a macro to

create literal string constants) is probably the best language to use when

dealing with 7-bit strings — because the advantage of 7-bit strings is their

compactness, and assembly language programmers tend to be the ones who

worry most about compactness, this is a good match.

**Character Sets**

Like strings, character sets are another composite data type built upon

the character data type. A *character set* is a mathematical set of characters.

Membership in a set is a binary relation. A character is either in the set or

it is not in the set; you cannot have multiple copies of the same character

in a character set. Furthermore, the concept of sequence (whether one

character comes before another, as in a string) is foreign to a character set.

If two characters are members of a set, their order in the set is irrelevant.

**Topic № 8**

Memory Organization and Access

**The Basic System Components**

The basic operational design of a computer system is called its *architecture*.

John von Neumann, a pioneer in computer design, is given credit for the

principal architecture in use today. For example, the 80x86 family uses the

*von Neumann architecture* (VNA). A typical von Neumann system has three

major components: the *central processing unit* (CPU), *memory,* and *input/output*

In VNA machines, like the 80x86, the CPU is where all the action takes place.

All computations occur within the CPU. Data and machine instructions

reside in memory until the CPU requires them, at which point the system

transfers the data into the CPU. To the CPU, most I/O devices look like

memory; the major difference between memory and I/O devices is the fact

that the latter are generally located in the outside world, whereas the former

is located within the same machine.

***The System Bus***

The *system bus* connects the various components of a VNA machine. Most

CPUs have three major buses: the *address* bus, the *data* bus, and the *control*

bus. A bus is a collection of wires on which electrical signals pass between

components of the system. These buses vary from processor to processor, but

each bus carries comparable information on most processors. For example,

the data buses on the Pentium and 80386 may have different implementations,

but both variants carry data between the processor, I/O, and

memory.

**The Data Bus**

CPUs use the *data bus* to shuffle data between the various components in a

computer system. The size of this bus varies widely among CPUs. Indeed, bus

size is one of the main attributes that defines the “size” of the processor.

Most modern, general-purpose CPUs employ a 32-bit-wide or 64-bit-wide

data bus. Some processors use 8-bit or 16-bit data buses, there may well be

some CPUs with 128-bit buses by the time you read this. For the most part,

however, the CPUs in personal computers tend to use 32-bit or 64-bit data

buses (and 64-bit data buses are the most prevalent).

You’ll often hear a processor called an *8-, 16-, 32-, or 64-bit processor.*

The smaller of the number of data lines on the processor and the size of

the largest general-purpose integer register determines the processor size.

For example, modern Intel 80x86 CPUs all have 64-bit buses, but only provide

32-bit general-purpose integer registers, so we’ll classify these devices

as 32-bit processors. The AMD x86-64 processors support 64-bit integer

registers and a 64-bit bus, so they’re 64-bit processors.

Although the 80x86 family members with 8-, 16-, 32-, and 64-bit data

buses can process data blocks up to the bit width of the bus, they can also

access smaller memory units of 8, 16, or 32 bits. Therefore, anything you can

do with a small data bus can be done with a larger data bus as well; the larger

data bus, however, may access memory faster and can access larger chunks of

data in one memory operation.

***The Address Bus***

The data bus on an 80x86 family processor transfers information between a

particular memory location or I/O device and the CPU. The only question

is, “Which memory location or I/O device?” The address bus answers that

question. To uniquely identify each memory location and I/O device, the

system designer assigns a unique memory address to each. When the software

wants to access a particular memory location or I/O device, it places the

corresponding address on the address bus. Circuitry within the device checks

this address and transfers data if there is an address match. All other memory

locations ignore the request on the address bus.

With a single address bus line, a processor could access exactly two

unique addresses: zero and one. With *n* address lines, the processor can

access 2*n* unique addresses (because there are 2*n* unique values in an *n*-bit

binary number). Therefore, the number of bits on the address bus will

determine the *maximum* number of addressable memory and I/O locations.

Early 80x86 processors, for example, provided only 20 lines on the address

bus. Therefore, they could only access up to 1,048,576 (or 220) memory

locations. Larger address buses can access more memory.

***The Control Bus***

The control bus is an eclectic collection of signals that control how the

processor communicates with the rest of the system. To illustrate its

importance, consider the data bus for a moment. The CPU uses the data bus

to move data between itself and memory. This prompts the question, “How

does the system know whether it is sending or receiving data?” Well, the

system uses two lines on the control bus, *read* and *write*, to determine the data

flow direction (CPU to memory, or memory to CPU). So when the CPU

wants to write data to memory, it *asserts* (places a signal on) the write control

line. When the CPU wants to read data from memory, it asserts the read

control line.

Although the exact composition of the control bus varies among processors,

some control lines are common to all processors and are worth a

brief mention. Among these are the system clock lines, interrupt lines, byte

enable lines, and status lines.

The *byte enable lines* appear on the control bus of some CPUs that support

byte-addressable memory*.* These control lines allow 16-, 32-, and 64-bit processors

to deal with smaller chunks of data by communicating the size of the

accompanying data. Additional details appear later in the sections on 16-bit

and 32-bit buses.

The control bus also contains a signal that helps distinguish between

address spaces on the 80x86 family of processors. The 80x86 family, unlike

many other processors, provides two distinct address spaces: one for memory

and one for I/O. However, it does not have two separate physical address

buses (for I/O and memory). Instead, the system shares the address bus for

both I/O and memory addresses. Additional control lines decide whether

the address is intended for memory or I/O. When such signals are active,

the I/O devices use the address on the LO 16 bits of the address bus. When

inactive, the I/O devices ignore the signals on the address bus, and the

memory subsystem takes over at that point.

A typical CPU addresses a maximum of 2 *n*different memory locations, where *n*is the number of bits on the address bus (most computer systems built around 80x86 family CPUs do not include the maximum addressable amount of memory). Of course, the first question you should ask is, 'What exactly is a memory location?' The 80x86, as an example, supports *byte-addressable memory*. Therefore, the basic memory unit is a byte. With address buses containing 20, 24, 32, or 36 address lines, the 80x86 processors can address 1 MB, 16 MB, 4 GB, or 64 GB of memory, respectively. Some CPU families do not provide byte-addressable memory (commonly, they only address memory in double-word or even quad-word chunks ). However, because of the vast amount of software written that assumes memory is byte-addressable (such as all those C/C++ programs out there), even CPUs that don't support byte-addressable memory in hardware still use byte addresses and simulate byte addressing in software. We'll return to this issue shortly.

Think of memory as an array of bytes. The address of the first byte is zero and the address of the last byte is 2 *n*ˆ’ 1. For a CPU with a 20-bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

Memory: array [0..1048575] of byte; // One-megabyte address space (20 bits)

Different computer systems have different solutions to this problem. The 80x86 family stores the LO byte of a word at the address specified and the HO byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, because a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations.

The address for a word or a double word is the address of its LO byte. The remaining bytes follow this LO byte, with the HO byte appearing at the address of the word plus one or the address of the double word plus three (see Figure 6-4). Note that it is quite possible for byte, word, and double-word values to overlap in memory.

Big Endian Versus Little Endian Organization

Earlier, you read that the 80x86 CPU family stores the LO byte of a word or double-word value at a particular address in memory and the successive HO bytes at successively higher addresses. There was also a vague statement to the effect that 'different processors handle this in different ways.' Well, now is the time to learn how different processors store multi-byte objects in byte-addressable memory.

Almost every CPU you'll use whose 'bit size ' is some power of two (8, 16, 32, 64, and so on) will number the bits and nibbles as shown in the previous chapters. There are some exceptions, but they are rare, and most of the time they represent a notational change, not a functional change (meaning you can safely ignore the difference). Once you start dealing with objects larger than eight bits, however, life becomes more complicated. Different CPUs organize the bytes in a multibyte object differently.

Consider the layout of the bytes in a double word on an 80x86 CPU (see Figure 6-10). The LO byte, which contributes the smallest component of a binary number, sits in bit positions zero through seven and appears at the lowest address in memory. It seems reasonable that the bits that contribute the least would be located at the lowest address in memory.

click to expand  
Figure : Byte layout in a double word on the 80x86 processor

Unfortunately, this is not the only organization that is possible. Some CPUs, for example, reverse the memory addresses of all the bytes in a double word, using the organization shown in Figure

click to expand  
Alternate byte layout in a double word

The byte organization that Intel uses is whimsically known as the *little endian byte organization*. The alternate form is known as *big endian byte organization*. If you're wondering, these terms come from Jonathan Swift's *Gulliver's Travels*; the Lilliputians were arguing over whether one should open an egg by cracking it on the little end or the big end, a parody of the arguments the Catholics and Protestants were having over their respective doctrines when Swift was writing.

The time for arguing over which format is better was back before there were several different CPUs created using different *byte genders*. (Many programmers refer to this as byte sex. Byte gender is a little less offensive, hence the use of that term in this book.) Today, we have to deal with the fact that different CPUs sport different byte genders, and we have to take care when writing software if we want that software to run on both types of processors. Arguing over whether one format is better than another is irrelevant at this point; regardless of which format is better or worse, we may have to put extra code in our programs to deal with both formats (including the worse of the two, whichever that is).

The big endian versus little endian problem occurs when we try to pass binary data between two computers. For example, the double-word binary representation of 256 on a little endian machine has the following byte values:

LO byte: 0 Byte #1: 1 Byte #2: 0 HO byte: 0

If you assemble these four bytes on a little endian machine, their layout takes this form:

Byte: 3 2 1 0 256: 0 0 1 0 (each digit represents an 8-bit value)

On a big endian machine, however, the layout takes the following form:

Byte: 3 2 1 0 256: 0 1 0 0 (each digit represents an 8-bit value)

This means that if you take a 32-bit value from one of these machines and attempt to use it on the other machine (whose byte gender is not the same), you won't get correct results. For example, if you take a big endian version of the value 256, you'll discover that it has the bit value one in bit position 16 in the little endian format. If you try to use this value on a little endian machine, that machine will think that the value is actually 65,536 (that is, %1\_0000\_0000\_0000\_0000). Therefore, when exchanging data between two different machines, the best solution is to convert your values to some canonical form and then, if necessary, convert the canonical form back to the local format if the local and canonical formats are not the same. Exactly what constitutes a 'canonical' format depends, usually, on the transmission medium. For example, when transmitting data across networks, the canonical form is usually big endian because TCP/IP and some other network protocols use the big endian format. This does not suggest that big endian is always the canonical form. For example, when transmitting data across the Universal Serial Bus (USB), the canonical format is little endian. Of course, if you control the software on both ends, the choice of canonical form is arbitrary; still, you should attempt to use the appropriate form for the transmission medium to avoid confusion down the road.

To convert between the endian forms, you must do a *mirror-image swap*of the bytes in the object. To cause a mirror-image swap, you must swap the bytes at opposite ends of the binary number, and then work your way towards the middle of the object swapping pairs of bytes as you go along. For example, to convert between the big endian and little endian format within a double word, you'd first swap bytes zero and three, then you'd swap bytes one and two.

CPU Memory Access

Most CPUs have two or three different ways to access memory. The most common *memory addressing modes*modern CPUs support are *direct, indirect,*and *indexed*. A few CPUs (like the 80x86) support additional addressing modes like *scaled indexed*, while some RISC CPUs only support indirect access to memory. Having additional memory addressing modes makes memory access more flexible. Sometimes a particular addressing mode can allow you to access data in a complex data structure with a single instruction, where two or more instructions would be required on a CPU without that addressing mode. Therefore, having a wide variety of ways to access memory is generally good as these complex addressing modes allow you to use fewer instructions.

It would seem that the 80x86 processor family (with many different typesof memory addressing modes) would be more efficient than a RISC processor that only supports a small number of addressing modes. In many respects, this is absolutely true; those RISC processors can often take three to five instructions to do what a single 80x86 instruction does. However, this does not mean that an 80x86 program will run three to five times faster. Don't forget that access to memory is very slow, usually requiring wait states. Whereas the 80x86 frequently accesses memory, RISC processors rarely do. Therefore, that RISC processor can probably execute the first four instructions, which do not access memory at all, while the single 80x86 instruction, which accesses memory, is spinning on some wait states. In the fifth instruction the RISC CPU might access memory and will incur wait states of its own. If both processors execute an average of one instruction per clock cycle and have to insert 30 wait states for a main memory access, we're talking about a difference of 31 clock cycles (80x86) versus 35 clock cycles (RISC), only about a 12 percent difference.

If an application must access slow memory, then choosing an appropriate addressing mode will often allow that application to compute the same result with fewer instructions and with fewer memory accesses, thus improving performance. Therefore, understanding how an application can use the different addressing modes a CPU provides is important if you want to write fast and compact code.

**Topic № 9**

Composite Data Types and Memory Objects

Composite data types are types that are composed of other, more primitive, types. Examples of composite data types commonly found in applications include pointers, arrays, records or structures, and unions. Many high-level languages provide syntactical abstractions for these composite data types that make them easy to declare and use, all the time hiding their underlying complexities.

Though the cost of using these composite data types is not terrible, it is very easy for a programmer to introduce inefficiencies into an application by using these data types without understanding the underlying costs. Great programmers, therefore, are cognizant of the costs associated with using composite data types so they can use them in an appropriate manner.

## Pointer Types

A *pointer*is a variable whose value refers to some other object. Now you've probably experienced pointers firsthand in Pascal, C/C++, or some other programming language, and you may be feeling a little anxious right now. Well, fear not! Pointers are actually *easy*to deal with.

Probably the best place to start is with the definition of a pointer. High-level languages like Pascal and C/C++ hide the simplicity of pointers behind a wall of abstraction. This added complexity tends to frighten programmers because *they don't understand what's going on behind the scenes*. However, a little knowledge can erase all your fears of pointers.

Let's just ignore pointers for a moment and work with something that's easier to understand: an array. Consider the following array declaration in Pascal:

M: array [0..1023] of integer;

Even if you don't know Pascal, the concept here is easy to understand. M is anarray with 1,024 integers in it, indexed from M[0] to M[1023] *.*Each one of these array elements can hold an integer value that is independent of the others. In other words, this array gives you 1,024 different integer variables each of which you access via an array index rather than by name .

If you have a program with the statement M[0]:=100; it probably wouldn't take you any time to figure out what this statement is doing. It stores the value 100 into the first element of the array M . Now consider the following two statements:

i := 0; (\* assume i is an integer variable \*) M [i] := 100;

You should agree, without too much hesitation, that these two statements do the same thing as M[0]:=100; . Indeed, you're probably willing to agree that you can use any integer expression in the range 0..1,023 as an index of this array. The following statements *still*perform the same operation as our earlier statement:

i := 5; (\* assume all variables are integers\*) j := 10; k := 50; m [i \* j - k] := 100;

Okay, how about the following:

M [1] := 0; M [ M [1] ] := 100;

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you'll discover that these two instructions perform the same operation as before. The first statement stores zero into array element M[1] . The second statement fetches the value of M[1] , which is zero, and uses that value to determine where it stores the value 100.

If you're willing to accept this example as reasonable - perhaps bizarre, but usable nonetheless - then you'll have no problems with pointers, *because M[1] is a pointer!*Well, not really, but if you were to change 'M' to 'memory' and treat each element of this array as a separate memory location, then this *is*the definition of a pointer: a pointer is a memory variable whose value is the address of some other memory object.

### Pointer Implementation

Although most languages implement pointers using memory addresses, a pointer is actually an abstraction of a memory address, and therefore a language could define a pointer using any mechanism that maps the value of the pointer to the address of some object in memory. Some implementations of Pascal, for example, use offsets from some fixed memory address as pointer values. Some languages (such as dynamic languages like LISP) might actually implement pointers by using double indirection. That is, the pointer object contains the address of some memory variable whose value is the address of the object to be accessed. This double indirection may seem somewhat convoluted, but it does offer certain advantages when using a complex memory management system. However, this chapter will assume that a pointer is a variable whose value is the address of some other object inmemory.

As you've seen in examples, you can indirectly access an object using a pointer with two 80x86 machine instructions (or with a similar sequence on other CPUs), as follows :

mov(PointerVariable, ebx); // Load the pointer variable into a register. mov([ebx], eax); // Use register indirect mode to access data.

Now consider the double-indirect pointer implementation described earlier. Access to data via double indirection is less efficient than the straight pointer implementation because it takes an extra machine instruction to fetch the data from memory. This isn't obvious in a high-level language like C/C++ or Pascal, where you'd use double indirection as follows (it looks very similar to single indirection):

i = \*\*cDblPtr; // C/C++ i := ^^pDblPtr; (\* Pascal \*)

In assembly language, however, you'll see the extra work involved:

mov(hDblPtr, ebx); // Get the pointer to a pointer. mov([ebx], ebx); // Get the pointer to the value. mov([ebx], eax); // Get the value.

Contrast this with the two assembly instructions (shown earlier) needed to access an object using single indirection. Because double indirection requires 50 percent more code than single indirection, you can see why many languages implement pointers using single indirection.

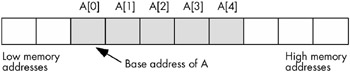
### Pointers and Dynamic Memory Allocation

Pointers typically reference anonymous variables that you allocate on the *heap*(a region in memory reserved for dynamic storage allocation) using memory allocation/deallocation functions in different languages like *malloc/free*(C), *new/dispose*(Pascal), and *new/delete*(C++). Objects you allocate on the heap are known as *anonymous variables*because you refer to them by their address, and you do not associate a name with them. And because the allocation functions return the address of an object on the heap, you would typically store the function's return result into a pointer variable. True, the pointer variable may have a name, but that name applies to the pointer's data (an address), not the name of the object referenced by this address.

## Arrays

After strings, arrays are probably the most common *composite data type*(a complex data type built up from smaller data objects). Yet few beginning programmers fully understand how arrays operate and know about their efficiency trade-offs. It's surprising how many novice programmers view arrays from a completely different perspective once they understand how arrays operate at the machine level.

Abstractly, an array is an aggregate data type whose members (elements) are all of the same type. A member is selected from the array by specifying the member's array index with an integer (or with some value whose underlying representation is an integer, such as character, enumerated, and Boolean types). This chapter assumes that all of the integer indexes of an array are numerically contiguous (though this is not required). That is, if both x and y are valid indexes of the array, and if x < y *,*then all i such that x < i < y are also valid indexes. In this book, we will assume that array elements occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 7-2.

  
Figure 7-2: Array layout in memory

The *base address*of an array is the address of the first element of the array and is at the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, and so on. Note that there is no requirement that the indexes start at zero. They may start with any number as long as they are contiguous. However, for the purposes of this discussion, it's easier to discuss array access if the first index is zero. We'll generally begin most arrays at index zero unless there is a good reason to do otherwise .

Whenever you apply the indexing operator to an array, the result is the unique array element specified by that index. For example, A[i] chooses the i thelement from array A .

### Array Declarations

Array declarations are very similar across many high-level languages. We'll look at some examples in many of these languages within this section.

C, C++, and Java all let you declare an array by specifying the total number of elements in an array. The syntax for an array declaration in these languages is as follows:

*data\_type array\_name*  [  *number\_of\_elements*  ];

Here are some sample C/C++ array declarations:

char CharArray[ 128 ]; int intArray[ 8 ]; unsigned char ByteArray[ 10 ]; int \*PtrArray[ 4 ];

If these arrays are declared as automatic variables , then C/C++ 'initializes' them with whatever bit patterns happen to be present in memory. If, on the other hand, you declare these arrays as static objects, then C/C++ zeros out each array element. If you want to initialize an array yourself, then you can use the following C/C++ syntax:

*data\_type array\_name*  [  *number\_of\_elements*  ] = {  *element\_list*  };

Here's a typical example:

int intArray[8] = {0,1,2,3,4,5,6,7};

HLA's array declaration syntax takes the following form, which is semantically equivalent to the C/C++ declaration:

*array\_name*  :  *data\_type*  [  *number\_of\_elements*  ];

Here are some examples of HLA array declarations, which all allocate storage for uninitialized arrays (the second example assumes that you have defined the integer data type in a type section of the HLA program):

static CharArray: char[128]; // Character array with elements // 0..127. IntArray: integer[ 8 ]; // Integer array with elements 0..7. ByteArray: byte[10]; // Byte array with elements 0..9. PtrArray: dword[4]; // Double-word array with elements 0..3.

You can also initialize the array elements using declarations like the following:

RealArray: real32[8] := [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 ]; IntegerAry: integer[8] := [ 8, 9, 10, 11, 12, 13, 14, 15 ];

Both of these definitions create arrays with eight elements. The first definition initializes each 4-byte real32 array element with one of the values in the range 0.0..7.0. The second declaration initializes each integer array element with one of the values in the range 8..15.

Pascal/Delphi/Kylix uses the following syntax to declare an array:

*array\_name*  : array[  *lower\_bound..upper\_bound*  ] of  *data\_type*  ;

As in the previous examples, *array\_name*is the identifier and *data\_type*is the type of each element in this array. Unlike C/C++, Java, and HLA, in Pascal/Delphi/Kylix you specify the upper and lower bounds of the array rather than the array's size . The following are typical array declarations in Pascal:

type ptrToChar = ^char; var CharArray: array[0..127] of char; // 128 elements IntArray: array[ 0..7 ] of integer; // 8 elements ByteArray: array[0..9] of char; // 10 elements PtrArray: array[0..3] of ptrToChar; // 4 elements

Although these Pascal examples start their indexes at zero, Pascal does not require a starting index of zero. The following is a perfectly valid array declaration in Pascal:

var ProfitsByYear : array[ 1998..2009 ] of real; // 12 elements

The program that declares this array would use indexes 1998 through 2009 when accessing elements of this array, not 0 through 11.

Many Pascal compilers provide an extra feature to help you locate defects in your programs. Whenever you access an element of an array, these compilers will automatically insert code that will verify that the array index is within the bounds specified by the declaration. This extra code will stop the program if the index is out of range. For example, if an index into ProfitsByYear is outside the range 1998..2009, the program would abort with an error. This is a very useful feature that helps verify the correctness of your program. [1]

Generally, array indexes are integer values, though some languages allow other *ordinal types*(those data types that use an underlying integer representation). For example, Pascal allows char and boolean array indexes. In Pascal, it's perfectly reasonable and useful to declare an array as follows:

alphaCnt : array[ 'A'..'Z' ] of integer;

You access elements of alphaCnt using a character expression as the array index. For example, consider the following Pascal code that initializes each element of alphaCnt to zero:

for ch := 'A' to 'Z' do alphaCnt[ ch ] := 0;

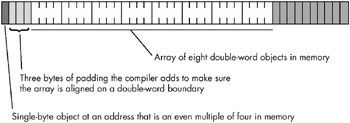
Assembly language and C/C++ treat most ordinal values as special instances of integer values, so they are certainly legal array indexes. Most implementations of BASIC will allow a floating-point number as an array index, though BASIC always truncates the value to an integer before using it as an index. [2]

### Array Representation in Memory

Abstractly, an array is a collection of variables that you access using an index. Semantically, we can define an array any way we please as long as it maps distinct indexes to distinct objects in memory and always maps the same index to the same object. In practice, however, most languages utilize a few common algorithms that provide efficient access to the array data.

The number of bytes of storage an array will consume is the product of the number of elements multiplied by the number of bytes per element in the array. Many languages also add a few additional bytes of padding at the end of the array so that the total length of the array is an even multiple of a nice value like four (on a 32-bit machine, a compiler may add bytes to the end of the array in order to extend its length to some multiple of four bytes). However, a program must *not*access these extra padding bytes because they may or may not be present. Some compilers will put them in, some will not, and some will only put them in depending on the type of object that immediately follows the array in memory.

Many optimizing compilers will attempt to place an array starting at a memory address that is an even multiple of some common size like two, four, or eight bytes. This effectively adds padding bytes before the beginning of the array or, if you prefer to think of it this way, it adds padding bytes to the end of the previous object in memory .

  
Figure : Adding padding bytes before an array

On those machines that do not support byte-addressable memory, those compilers that attempt to place the first element of an array on an easily accessed boundary will allocate storage for an array on whatever boundary the machine supports. If the size of each array element is less than the minimum size memory object the CPU supports, the compiler implementer has two options:

* Allocate the smallest accessible memory object for each element of the array
* Pack multiple array elements into a single memory cell

The first option has the advantage of being fast, but it wastes memory because each array element carries along some extra storage that it doesn't need. The second option is compact, but it requires extra instructions to pack and unpack data when accessing array elements, which means that accessing elements is slower. Compilers on such machines often provide an option that lets you specify whether you want the data packed or unpacked so you can make the choice between space and speed.

If you're working on a byte-addressable machine (like the 80x86) then you probably don't have to worry about this issue. However, if you're using a high-level language and your code might wind up running on a different machine at some point in the future, you should choose an array organization that is efficient on all machines.

### Accessing Elements of an Array

If you allocate all the storage for an array in contiguous memory locations, and the first index of the array is zero, then accessing an element of a one-dimensional array is simple. You can compute the address of any given element of an array using the following formula:

Element\_Address = Base\_Address + index \* Element\_Size

The Element\_Size item is the number of bytes that each array element occupies. Thus, if the array contains elements of type byte, the Element\_Size field is one and the computation is very simple. If each element of the array is a word (or another two-byte type) then Element\_Size is two. And so on.

Consider the following Pascal array declaration:

var SixteenInts : array[ 0..15] of integer;

To access an element of the SixteenInts on a byte-addressable machine, assuming 4-byte integers, you'd use this calculation:

Element\_Address = AddressOf( SixteenInts) + index\*4

In assembly language (where you would actually have to do this calculation manually rather than having the compiler do the work for you), you'd use code like the following to access array element SixteenInts[index] :

mov( index, ebx ); mov( SixteenInts[ ebx\*4 ], eax );

### Multidimensional Arrays

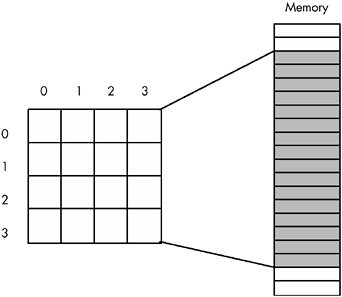
Most CPUs can easily handle one-dimensional arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and several machine instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to look at how to implement them in memory. The first problem is to figure out how to store a multidimensional object in a one-dimensional memory space.

Consider for a moment a Pascal array of the following form:

A:array[0..3,0..3] of char;

This array contains 16 bytes organized as four rows of four characters . Somehow you have to draw a correspondence between each of the 16 bytes in this array and each of the 16 contiguous bytes in main memory. Figure 7-4 shows one way to do this.

  
Figure  Mapping a 4x4 array to sequential memory locations

The actual mapping is not important as long as two things occur:

* No two entries in the array occupy the same memory location(s)
* Each element in the array always maps to the same memory location

Therefore, what you really need is a function with two input parameters - one for a row and one for a column value - that produces an offset into a contiguous block of 16 memory locations.

Any old function that satisfies these two constraints will work fine. However, what you really want is a mapping function that is efficient to compute at run time and that works for arrays with any number of dimensions and any bounds on those dimensions. While there are a large number of possible functions that fit this bill, there are two functions that most high-level languages use: *row-major ordering*and *column-major ordering.*

**Topic № 10**

Composite Data Types and Memory Objects (continued)

If you have an *m* × *n* array, it will have m × n elements and will require

m × n × Element\_Size bytes of storage. To allocate storage for an array

you must reserve this amount of memory. With one-dimensional arrays,

the syntax that the different high-level languages employ is very similar.

However, their syntax starts to diverge when you consider multidimensional

arrays.

In C, C++, and Java, you use the following syntax to declare a multidimensional

array:

*data\_type array\_name* [dim1][dim2] . . . [dimn];

Here is a concrete example of a three-dimensional array declaration in

C/C++:

int threeDInts[ 4 ][ 2 ][ 8 ];

This example creates an array with 64 elements organized with a depth of

four by two rows by eight columns. Assuming each int object requires 4 bytes,

this array consumes 256 bytes of storage.

Pascal’s syntax actually supports two equivalent ways of declaring multidimensional

arrays. The following example demonstrates both of these

forms:

var

threeDInts : array[0..3] of array[0..1] of array[0..7] of integer;

threeDInts2 : array[0..3, 0..1, 0..7] of integer;

Semantically, there are only two major differences that exist among different

languages. The first difference is whether the array declaration specifies the

overall size of each array dimension or whether it specifies the upper and

lower bounds. The second difference is whether the starting index is zero,

one, or a user-specified value.

**Accessing Elements of a Multidimensional Array**

Accessing an element of a multidimensional array in a high-level language

is so easy that a typical programmer will do so without considering the associated

cost. In this section, we’ll look at some of the assembly language

sequences you’ll need to access elements of a multidimension array to give

you a clearer picture of these costs.

Consider again, the C/C++ declaration of the ThreeDInts array from the

previous section:

int ThreeDInts[ 4 ][ 2 ][ 8 ];

In C/C++, if you wanted to set element [i][j][k] of this array to the value of

n, you’d probably use a statement similar to the following:

ThreeDInts[i][j][k] = n;

This statement, however, hides a great deal of complexity. Recall the formula

needed to access an element of a three-dimensional array:

Element\_Address =

Base\_Address +

((rowindex \* col\_size + colindex) \* depth\_size + depthindex) \*

Element\_Size

The ThreeDInts example does not avoid this calculation, it only hides it from

you. The machine code that the C/C++ compiler generates is similar to the

following:

intmul( 2, i, ebx ); // EBX = 2 \* i

add( j, ebx ); // EBX = 2 \* i + j

intmul( 8, ebx ); // EBX = (2 \* i + j) \* 8

add( k, ebx ); // EBX = (2 \* i + j) \* 8 + k

mov( n, eax );

mov( eax, ThreeDInts[ebx\*4] ); // ThreeDInts[i][j][k] = n

Actually, ThreeDInts is special. The sizes of all the array dimensions are nice

powers of two. This means that the CPU can use shifts instead of multiplication

instructions to multiply EBX by two and by four in this example.

Because shifts are often faster than multiplication, a decent C/C++ compiler

will generate the following code:

mov( i, ebx );

shl( 1, ebx ); // EBX = 2 \* i

add( j, ebx ); // EBX = 2 \* i + j

shl( 3, ebx ); // EBX = (2 \* i + j) \* 8

add( k, ebx ); // EBX = (2 \* i + j) \* 8 + k

mov( n, eax );

mov( eax, ThreeDInts[ebx\*4] ); // ThreeDInts[i][j][k] = n

Note that a compiler can only use this faster code if an array dimension is a

power of two. This is the reason many programmers attempt to declare arrays

whose dimension sizes are some power of two. Of course, if you must declare

extra elements in the array to achieve this goal, you may wind up wasting

space (especially with higher-dimensional arrays) to achieve a small increase

in speed.

For example, if you need a 10×10 array and you’re using row-major

ordering, you could create a 10×16 array to allow the use of a shift (by four)

instruction rather than a multiply (by 10) instruction. When using column major

ordering, you’d probably want to declare a 16×10 array to achieve the

same effect, since row-major calculation doesn’t use the size of the first

dimension when calculating an offset into an array, and column-major

calculation doesn't use the size of the second dimension when calculating an

offset. In either case, however, the array would wind up having 160 elements

instead of 100 elements. Only you can decide if this extra space is worth the

small increase in speed you’ll gain.

**Records/Structures**

Another major composite data structure is the Pascal *record* or C/C++

*structure*. The Pascal terminology is probably better, as it avoids confusion

with the term *data structure*. Therefore, we’ll adopt the term *record* here.

An array is homogeneous, meaning that its elements are all of the same

type. A record, on the other hand, is heterogeneous and its elements can

have differing types. The purpose of a record is to let you encapsulate

logically related values into a single object.

Arrays let you select a particular element via an integer index. With

records, you must select an element, known as a *field*, by the field’s name.

Each of the field names within the record must be unique. That is, the same

field name may not appear two or more times in the same record. However,

all field names are local to their record, and you may reuse those names

elsewhere in the program.

***Records in Pascal/Delphi***

Here’s a typical record declaration for a Student data type in Pascal/Delphi:

type

Student =

record

Name: string [64];

Major: smallint; // 2-byte integer in Delphi

SSN: string[11];

Mid1: smallint;

Midt: smallint;

Final: smallint;

Homework: smallint;

Projects: smallint;

end;

Many Pascal compilers allocate all of the fields in contiguous memory

locations. This means that Pascal will reserve the first 65 bytes for the name,3

the next 2 bytes hold the major code, the next 12 bytes the Social Security

number, and so on.

***Records in C/C++***

Here’s the same declaration in C/C++:

typedef

struct

{

char Name[65]; // Room for a 64-character zero-terminated string.

short Major; // Typically a 2-byte integer in C/C++

char SSN[12]; // Room for an 11-character zero-terminated string.

short Mid1;

short Mid2;

short Final;

short Homework;

short Projects

} Student;

A discriminant **union** (or just *union*) is very similar to a record. Like records,

unions have fields and you access those fields using dot notation. In fact, in

many languages, about the only syntactical difference between records and

unions is the use of the keyword union rather than record. Semantically,

however, there is a big difference between a record and a union. In a record,

each field has its own offset from the base address of the record, and the

fields do not overlap. In a union, however, all fields have the same offset,

zero, and all the fields of the union overlap. As a result, the size of a record

is the sum of the sizes of all the fields (plus, possibly, some padding bytes),

whereas a union’s size is the size of its largest field (plus, possibly, some

padding bytes at the end).

Because the fields of a union overlap, you might think that a union

has little use in a real-world program. After all, if the fields all overlap,

then changing the value of one field changes the values of all the other

fields as well. This generally means that the use of a union’s field is *mutually*

*exclusive* — that is, you can only use one field at any given time. This observation

is generally correct, but although this means that unions aren’t as

generally applicable as records, they still have many uses.

***Unions in C/C++***

Here’s an example of a union declaration in C/C++:

typedef union

{

unsigned int i;

float r;

unsigned char c[4];

} unionType;

Assuming the C/C++ compiler in use allocates four bytes for unsigned

integers, the size of a unionType object will be four bytes (because all three

fields are 4-byte objects).

***Memory Storage of Unions***

The big difference between a union and a record is the fact that records

allocate storage for each field at different offsets, whereas unions overlay

each of the fields at the same offset in memory.

**Topic № 11**

**Representation of Multimedia Data**

**Digitization of Sound**

Let us first analyse what a sound actually is:

* Sound is a continuous wave that travels through the air
* The wave is made up of pressure differences. Sound is detected by measuring the pressure level at a location.
* Sound waves have normal wave properties (reflection, refraction, diffraction, etc.).

A variety of sound sources:

**Source**

-- Generates Sound

* Air Pressure changes
* ***Electrical*** -- Loud Speaker
* ***Acoustic*** -- Direct Pressure Variations

The destination receives (sensed the sound wave pressure changes) and has to deal with accordingly:

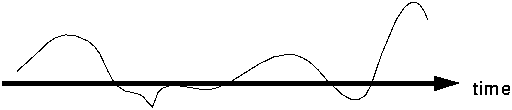
**Destination**

-- Receives Sound

* ***Electrical*** -- Microphone produces electric signal
* ***Ears*** -- Responds to pressure **hear** sound

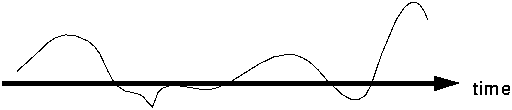
Sound is required input into a computer: it needs to sampled or digitised:

* Microphones, video cameras produce *analog signals* (continuous-valued voltages) as illustrated in Fig



**Continuous Analog Waveform**

* To get audio or video into a computer, we have to *digitize* it (convert it into a stream of numbers) **Need to convert Analog-to-Digital** -- Specialised Hardware
* So, we have to understand *discrete sampling* (both time and voltage)
* *Sampling* - divide the horizontal axis (the time dimension) into discrete pieces. Uniform sampling is ubiquitous.
* *Quantization* - divide the vertical axis (signal strength) into pieces. Sometimes, a non-linear function is applied.
  + 8 bit quantization divides the vertical axis into 256 levels. 16 bit gives you 65536 levels.



**Continuous Analog Waveform**

**Digitizing Audio**

That is the basic idea of digitizing a sound unfortunately things are (practically speaking) not so simple.

* Questions for producing digital audio (Analog-to-Digital Conversion):

**1.**

How often do you need to sample the signal?

**2.**

How good is the signal?

**3.**

How is audio data formatted?

**Computer Manipulation of Sound**

Once Digitised processing the digital sound is essentially straightforward although it depends on the processing you wish to do (***e.g.*** volume is easier to code than accuarte reverb)

Essentially they all operate on the 1-D array of digitised samples, typical examples include:

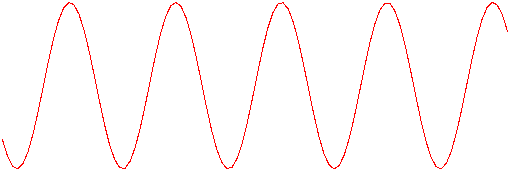
* Volume
* Cross-Fading
* Looping
* Echo/Reverb/Delay
* Filtering
* Signal Analysis

Soundedit Demos

* Volume
* Cross-Fading
* Looping
* Echo/Reverb/Delay
* Filtering
* **Sample Rates and Bit Size**
* How do we store each sample value (***Quantisation***)?
* **8 Bit Value**
* (0-255)
* **16 Bit Value**
* (Integer) (0-65535)
* How many Samples to take?
* **11.025 KHz**
* -- Speech (Telephone 8KHz)
* **22.05 KHz**
* -- Low Grade Audio  
  (WWW Audio, AM Radio)
* **44.1 KHz**
* -- CD Quality

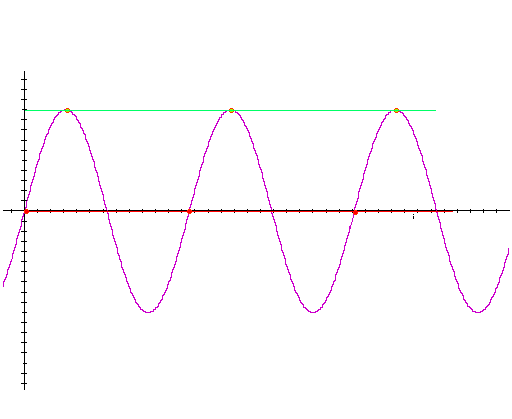
**Nyquist's Sampling Theorem**

* Suppose we are sampling a sine wave (Fig [6.3](https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node149.html#sine). How often do we need to sample it to figure out its frequency?



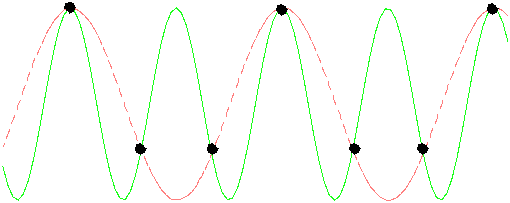
**A Sine Wave**

* If we sample at 1 time per cycle, we can think it's a constant (Fig [6.4](https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node149.html#1hz))



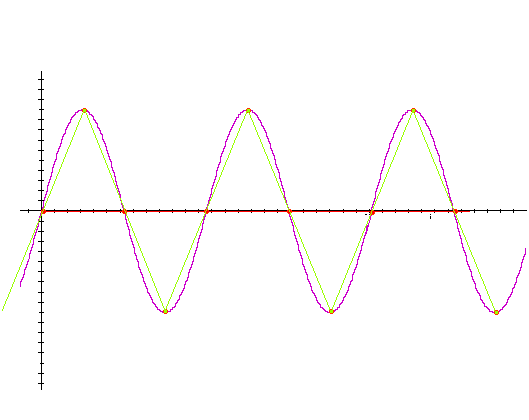
**Sampling at 1 time per cycle**

* If we sample at 1.5 times per cycle, we can think it's a lower frequency sine wave (Fig. [6.6](https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node149.html#15hz))



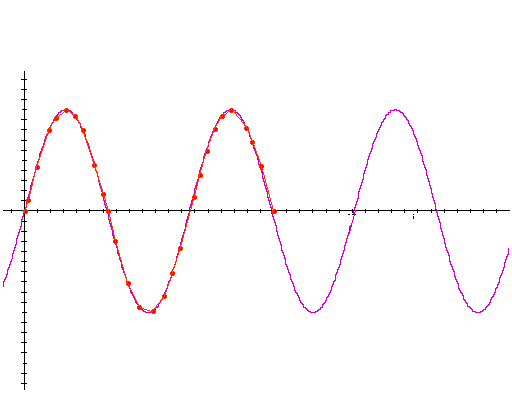
**Sampling at 1.5 times per cycle**

* Now if we sample at twice the sample frequency, i.e Nyquist Rate, we start to make some progress. An alternative way of viewing thr waveform (re)genereation is to think of straight lines joining up the peaks of the samples. In this case (at these sample points) we see we get a sawtooth wave that begins to start crudely approximating a sine wave



**Sampling at 2 times per cycle**

* **Nyquist rate** -- For lossless digitization, the sampling rate should be ***at least twice*** the maximum frequency responses. Indeed many times more the better.



**Sampling at many times per cycle**

**Typical Audio Formats**

* Popular audio file formats include .au (Unix workstations), .aiff (MAC, SGI), .wav (PC, DEC workstations)
* A simple and widely used audio compression method is Adaptive Delta Pulse Code Modulation (ADPCM). Based on past samples, it predicts the next sample and encodes the difference between the actual value and the predicted value.
* **Graphic/Image Data Structures**
* A digital image consists of many picture elements, termed **pixels**. The number of pixels that compose a monitor image determine the quality of the image (**resolution**). Higher resolution always yields better quality.
* A ***bit-map*** representation stores the graphic/image data in the same manner that the computer monitor contents are stored in video memory.

**Sample Monochrome Bit-Map Image**

* Each pixel is stored as a single bit (0 or 1)
* A 640 x 480 monochrome image requires 37.5 KB of storage.
* *Dithering* is often used for displaying monochrome images

### Gray-scale Images

**Example of a Gray-scale Bit-map Image**

* Each pixel is usually stored as a byte (value between 0 to 255)
* A 640 x 480 greyscale image requires over 300 KB of storage.

**Example of 8-Bit Colour Image**

* One byte for each pixel
* Supports 256 out of the millions s possible, acceptable colour quality
* Requires Colour Look-Up Tables (LUTs)
* A 640 x 480 8-bit colour image requires 307.2 KB of storage (the same as 8-bit greyscale)

**Example of 24-Bit Colour Image**

* Each pixel is represented by three bytes (e.g., RGB)
* Supports 256 x 256 x 256 possible combined colours (16,777,216)
* A 640 x 480 24-bit colour image would require 921.6 KB of storage
* Most 24-bit images are 32-bit images, the extra byte of data for each pixel is used to store an *alpha* value representing special effect information

### GIF (GIF87a, GIF89a)

* Graphics Interchange Format (GIF) devised by the UNISYS Corp. and Compuserve, initially for transmitting graphical images over phone lines via modems
* Uses the Lempel-Ziv Welch algorithm (a form of Huffman Coding), modified slightly for image scan line packets (line grouping of pixels)
* Limited to only 8-bit (256) colour images, suitable for images with few distinctive colours (e.g., graphics drawing)
* Supports *interlacing*

### JPEG

* A standard for photographic image compression created by the Joint Photographics Experts Group
* Takes advantage of limitations in the human vision system to achieve high rates of compression
* Lossy compression which allows user to set the desired level of quality/compression
* Detailed discussions in next chapter on compression.

### Microsoft Windows: BMP

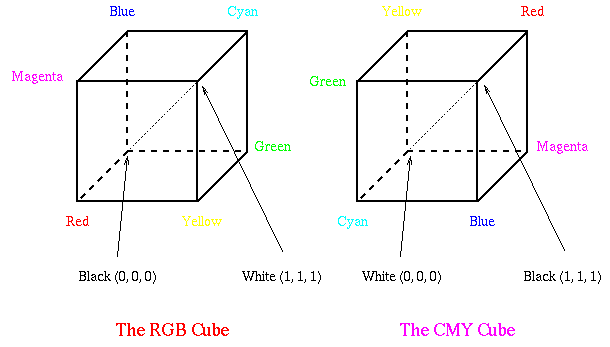
* A system standard graphics file format for Microsoft Windows
* Used in PC Paintbrush and other programs
* It is capable of storing 24-bit bitmap images

### The Human Retina

* The eye is basically just like a camera
* Each neuron is either a *rod* or a *cone*. Rods are not sensitive to colour.

**The CMY Colour Model**

* Cyan, Magenta, and Yellow (CMY) are complementary colours of RGB (Fig. [6.26](https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node189.html#RGBCUBE)). They can be used as *Subtractive Primaries*.
* CMY model is mostly used in printing devices where the colour pigments on the paper absorb certain colours (e.g., no red light reflected from cyan ink).



**The RGB and CMY Cubes**

#### Conversion between RGB and CMY:

- e.g., convert **White** from (1, 1, 1) in RGB to (0, 0, 0) in CMY.

$\left[ \begin{array}
{c}C\ M\ Y\end{array} \right] =
 \left[ \begin{array}
{c...
 ...\ 1\end{array} \right] -
 \left[ \begin{array}
{c}R\ G\ B\end{array} \right] $

$\left[ \begin{array}
{c}R\ G\ B\end{array} \right] =
 \left[ \begin{array}
{c...
 ...\ 1\end{array} \right] -
 \left[ \begin{array}
{c}C\ M\ Y\end{array} \right] $

* Sometimes, an alternative CMYK model (K stands for *Black*) is used in colour printing (e.g., to produce darker black than simply mixing CMY). where
* Colour images are encoded as triplets of values.
* Three common systems of encoding in video are RGB, YIQ, and YCrCb.
* Besides the hardware-oriented colour models (i.e., RGB, CMY, YIQ, YUV), HSB (Hue, Saturation, and Brightness, e.g., used in Photoshop) and HLS (Hue, Lightness, and Saturation) are also commonly used.
* YIQ uses properties of the human eye to prioritize information. Y is the black and white (luminance) image, I and Q are the colour (chrominance) images. YUV uses similar idea.
* CCIR 601 is a standard for digital video that specifies image size, and decimates the chrominance images (for 4:2:2 video).

**Types of Colour Video Signals**

* **Component video** - each primary is sent as a separate video signal.
  + The primaries can either be RGB or a luminance-chrominance transformation of them (e.g., YIQ, YUV).
  + Best colour reproduction
  + Requires more bandwidth and good synchronization of the three components
* **Composite video** - colour (chrominance) and luminance signals are mixed into a single carrier wave. Some interference between the two signals is inevitable.
* **S-Video** (Separated video, e.g., in S-VHS) - a compromise between component analog video and the composite video. It uses two lines, one for luminance and another for composite chrominance signal.

**Topic № 12**

**Boolean Logic and Digital Design**

Boolean logic is the basis of computation in modern computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of Boolean equations.

## Boolean Algebra

Boolean algebra is a deductive mathematical system. A *binary operator*' °' accepts a pair of Boolean inputs and produces a single Boolean value. For example, the Boolean AND operator accepts two Boolean inputs and produces a single Boolean output (the logical AND of the two inputs).

### The Boolean Operators

For our purposes, we will base Boolean algebra on the following set of operators and values:

* The two possible values in the Boolean system are zero and one. Often we will call these values *false*and *true*, respectively.
* The symbol '\*' represents the logical AND operation. For example, *A\*B*is the result of logically ANDing the Boolean values *A*and *B*. When using single letter variable names , this text will drop the '\*' symbol; therefore, *AB*also represents the logical AND of the variables *A*and *B,*which we will also call the product of *A*and *B*.
* The symbol '+' represents the logical OR operation. For example, *A + B*is the result of logically ORing the Boolean values *A*and *B*. We will also call this the sum of *A*and *B*.
* Logical complement, logical negation, and NOT, are all names for the same unary operator. This chapter will use the ( *'*) symbol to denote logical negation. For example, *A'*denotes the logical NOT of *A*.

### Boolean Postulates

Every algebraic system follows a certain set of initial assumptions, or *postulates*. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems are no different, and usually employ the following postulates:

* *Closure*. A Boolean system is *closed*with respect to a particular binary operator if, for every pair of Boolean values, it only produces a Boolean result.
* *Commutativity*. A binary operator ' °' is said to be commutative if *A °B=B °A*for all possible Boolean values *A*and *B*.
* *Associativity*. A binary operator ' °' is said to be associative if *(A °B) °C= A ° (B ° C)*for all Boolean values *A, B,*and *C*.
* *Distribution*. Two binary operators ' °' and '%' are distributive if *A °(B%C) = (A ° B) % (A ° C)*for all Boolean values *A, B,*and *C*.
* *Identity*. A Boolean value I is said to be the *identity element*with respect to some binary operator ' °' if *A ° I = A*for all Boolean values *A*.
* *Inverse*. A Boolean value I is said to be the *inverse element*with respect to some binary operator ' °' if *A ° I = B*and *B*‰  *A*(i.e., *B*is the opposite value of *A*in a Boolean system) for all Boolean values *A*and *B.*

When applied to the Boolean operators, the preceding postulates produce the following set of *Boolean postulates*:

* P1: Boolean algebra is closed under the AND, OR, and NOT operations.
* P2: The identity element of AND (\*) is one, and the identity element of OR (+) is zero. There is no identity element with respect to logical NOT ( *'*).
* P3: The \* and + operators are commutative.
* P4: \* and + are distributive with respect to one another. That is, *A \*(B+C) = (A \* B) + (A \* C)*and *A + (B \* C) = (A + B) \* (A + C)*.
* P5: \* and + are both associative. That is, *(A \* B) \* C = A \* (B \* C)*and *(A+B) + C = A + (B + C)*.
* P6: For every value *A*there exists a value *A'*such that *A \* A' =*0 and *A+A' =*1. This value is the logical complement (or NOT) of *A*.

You can prove all other theorems in Boolean algebra using this set of Boolean postulates. This chapter will not go into the formal proofs of the following theorems, but familiarity with some important theorems in Boolean algebra will be useful. Here are some of the important theorems:

|  |  |
| --- | --- |
| Th1: | *A*+ *A*= *A* |
| Th2: | *A*\* *A*= *A* |
| Th3: | *A*+ 0 = *A* |
| Th4: | *A*\* 1 = *A* |
| Th5: | *A*\* 0 = 0 |
| Th6: | *A*+ 1 = 1 |
| Th7: | ( *A*+ *B*) *'*= *A'*\* *B'* |
| Th8: | ( *A*\* *B*) *'*= *A'*+ *B'* |
| Th9: | *A*+ *A*\* *B*= *A* |
| Th10: | *A*\* ( *A*+ *B*) = *A* |
| Th11: | *A*+ *A'B*= *A*+ *B* |
| Th12: | *A'*\* ( *A*+ *B'*) = *A'B'* |
| Th13: | *AB*+ *AB'*= *A* |
| Th14: | ( *A'*+ *B'*) \* ( *A'*+ *B*) = *A'* |
| Th15: | *A*+ *A'*= 1 |
| Th16: | *A*\* *A'*= 0 |
| Note | *Theorems seven and eight are called*DeMorgan's Theorems *after the mathematician who discovered them.* |

An important principle in the Boolean algebra system is that of *duality*. Each pair, theorems 1 and 2, theorems 3 and 4, and so on, forms a *dual*. Any valid expression you can create using the postulates and theorems of Boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the \* and + operators and swap the 0 and 1 values in an expression, the resulting expression will obey all the rules of Boolean algebra. *This does not mean the dual expression computes the same values*; it only means that both expressions are legal in the Boolean algebra system.

### 8.1.3 Boolean Operator Precedence

If several different Boolean operators appear within a single Boolean expression, the result of the expression depends on the *precedence*of the operators. The following Boolean operators are ordered from highest precedence to lowest :

* parentheses
* logical NOT
* logical AND
* logical OR

The logical AND and OR operators are *left associative.*This means that if two operators with the same precedence appear between three operands, you must evaluate the expressions from left to right. The logical NOT operation is right associative, although it would produce the same result using either left or right associativity because it is a unary operator having only a single operand.

A Boolean *expression*is a sequence of zeros, ones, and *literals*separated by Boolean operators. A Boolean literal is a primed (negated) or unprimed variable name , and all variable names will be a single alphabetic character. A Boolean function is a specific Boolean expression; we will generally give Boolean functions the name *F*with a possible subscript. For example, consider the following Boolean function:

F  = AB + C

This function computes the logical AND of *A*and *B*and then logically ORs this result with *C*. If *A = 1*, *B = 0*, and *C = 1*, then *F*returns the value one (1\* 0 + 1 = 1).

You can also represent a Boolean function with a *truth table*. The truth tables for the logical AND and OR functions are shown in Tables.

| Table : AND truth table | | |
| --- | --- | --- |
| **AND** |  | **1** |
|  |  |  |
| 1 |  | 1 |

| Table : OR truth table | | |
| --- | --- | --- |
| **OR** |  | **1** |
|  |  | 1 |
| 1 | 1 | 1 |

For binary operators and two input variables , this form of a truth table is very natural and convenient . However, for functions involving more than two variables, these truth-table forms don't work well.

Table shows another way to represent truth tables. This form has several advantages - it is easier to fill in the table, it supports three or more variables, and it provides a compact representation for two or more functions. The example in Table demonstrates how to create a truth table for three different functions of three input variables.

| Table  Truth Table Format for a Function of Three Variables | | | | | |
| --- | --- | --- | --- | --- | --- |
| **C** | **B** | **A** | **F = ABC** | **F = AB + C** | **F = A+BC** |
|  |  |  |  |  |  |
|  |  | 1 |  |  | 1 |
|  | 1 |  |  |  |  |
|  | 1 | 1 |  | 1 | 1 |
| 1 |  |  |  | 1 |  |
| 1 |  | 1 |  | 1 | 1 |
| 1 | 1 |  |  | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

Although you can create an infinite variety of Boolean functions, they are not all unique. For example, *F = A*and *F = AA*are two different functions. By theorem two, however, it is easy to show that these two functions produce exactly the same result no matter what input value you supply for *A*. As it turns out, if you fix the number of input variables you're going to allow, there are a finite number of unique Boolean functions possible. For example, there are only 16 unique Boolean functions with two input variables and there are only 256 possible Boolean functions with three input variables. Given *n*input variables, there are https://flylib.com/books/1/330/1/html/2/images/p2n.jpg unique Boolean functions (two raised to two raised to the *nth*power). With two input variables there are https://flylib.com/books/1/330/1/html/2/images/fig212_01.jpg or 16 different functions. With three input variables there are https://flylib.com/books/1/330/1/html/2/images/fig212_02.jpg or 256 possible functions. Four input variables have https://flylib.com/books/1/330/1/html/2/images/fig212_03.jpg or 2 16, or 65,536 unique Boolean functions.

When working with only 16 Boolean functions (two input variables), we can name each unique function. Table lists common names for these functions.

| Table  Common Names for Boolean Functions of Two Variables | | |
| --- | --- | --- |
| **Function Number [1]** | **Function Name** | **Description** |
|  | Zero (clear) | Always returns zero regardless of A and B input values |
| 1 | Logical NOR | (NOT (A OR B)) = (A + B) ² |
| 2 | Inhibition (AB ² ) | Inhibition = AB ² (A AND not B). Also equivalent to A > B or B < A |
| 3 | NOT B | Ignores A and returns B ² |
| 4 | Inhibition (BA ² ) | Inhibition = BA ² (B AND not A). Also equivalent to B > A or A < B |
| 5 | NOT A | Returns A ² and ignores B |
| 6 | Exclusive-or (XOR) | A • B. Also equivalent to A ‰  B |
| 7 | Logical NAND | (NOT (A AND B)) = (A \* B) ² |
| 8 | Logical AND | A \* B = (A AND B) |
| 9 | Equivalence (exclusive-NOR) | (A = B). Also known as exclusive-NOR (not exclusive-OR) |
| 10 | A | Copy A. Returns the value of A and ignores B's value |
| 11 | Implication, B implies A | A + B ² . (If B then A). Also equivalent to B ‰ A |
| 12 | B | Copy B. Returns the value of B and ignores A's value |
| 13 | Implication, A implies B | B + A ² . (If A then B). Also equivalent to A ‰ B |
| 14 | Logical OR | A + B. Returns A OR B |
| 15 | One (set) | Always returns one regardless of A and B input values |

Although the connection between computer systems and Boolean logic in programming languages like C or Pascal may seem tenuous, it is actually much stronger than it first appears. There is a one-to-one relationship between the set of all Boolean functions and the set of all electronic circuits. Electrical engineers , who design CPUs and other computer- related circuits, have to be intimately familiar with this stuff.

**Topic № 13**

**Boolean Logic and Digital Design (continued)**

### Correspondence Between Electronic Circuits and Boolean Functions

For any Boolean function, you can design an equivalent electronic circuit and vice versa. Because Boolean functions only use the AND, OR, and NOT Boolean operators (these are the only operators that appear within canonical forms), we can construct any electronic circuit using only these three operations. The Boolean AND, OR, and NOT functions correspond to the AND, OR, and inverter (NOT) electronic circuit gates . These symbols are standard electronic symbols appearing in *schematic diagrams*. (interested readers wanted to learn more about electronic schematic diagrams should check out any book on electronic design).

click to expand  
Figure : AND, OR, and inverter (NOT) gates

The lines to the left of each item in Figure , with the *A*and *B*labels, correspond to a logic function input; the line leaving each diagram corresponds to the function's output.

However, you actually need only a single gate type to implement *any*electronic circuit. This gate is the NAND (not AND) gate, shown in Figure . The NAND gate tests its two inputs ( *A*and *B*) and presents a false on the output pin if both inputs are true, it places true on the output pin if both inputs are not true.

click to expand  
Figure : The NAND gate

To prove that we can construct any Boolean function using only NAND gates, we must show how to build an inverter (NOT), an AND gate, and an OR gate. Building an inverter is easy; just connect the two inputs together

click to expand  
Figure : Inverter built from a NAND gate

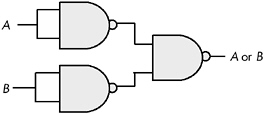
Once we can build an inverter, building an AND gate is easy - just invert the output of a NAND gate. After all, NOT (NOT ( *A*AND *B*)) is equivalent to *A*AND *B*. Of course, this takes two NAND gates to construct a single AND gate, but no one said that circuits constructed only with NAND gates would be optimal, only that they would be possible.

click to expand  
Figure : Constructing an AND gate from two NAND gates

The remaining gate we need to synthesize is the logical-OR gate. We can easily construct an OR gate from NAND gates by applying DeMorgan's Theorems.

(A or B)' = A' and B' DeMorgan's Theorem. A or B = (A' and B')' Invert both sides of the equation. A or B = A' nand B' Definition of NAND operation.

By applying these transformations, you get the circuit in Figure .

  
Figure : Constructing an OR gate from NAND gates

You might be wondering why we would even bother with this. After all, why not just use logical AND, OR, and inverter gates directly? There are two reasons. First, NAND gates are generally less expensive to build than other gates. Second, it is also much easier to build up complex integrated circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

### Combinatorial Circuits

A computer's CPU is built from combinatorial circuits. A *combinatorial circuit*is a system containing basic Boolean operations (AND, OR, NOT), some inputs, and a set of outputs. A combinatorial circuit often implements several different Boolean functions, with each output corresponding to an individual logic function. It is very important that you remember that *each output represents a different Boolean function*.

#### Addition Circuits

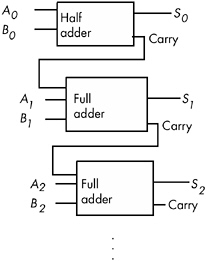
You can implement addition using Boolean functions. Suppose you have two 1-bit numbers , *A*and *B*. You can produce the 1-bit sum and the 1-bit carry of this addition using these two Boolean functions:

S = AB' + A'B Sum of A and B. C = AB Carry from addition of A and B.

These two Boolean functions implement a *half adder*. Electrical engineers call it a half adder because it adds two bits together but cannot add in a carry from a previous operation. A *full adder*adds three 1-bit inputs (two bits plus a carry from a previous addition ) and produces two outputs: the sum and the carry. These are the two logic equations for a full adder:

S = A'B'C  in  + A'BC  in  ' + AB'C  in  ' + ABC  in  C  out  = AB + AC  in  + BC  in

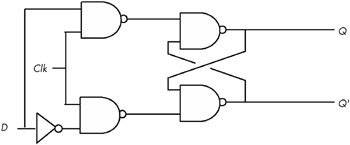
Although these equations only produce a single bit result (plus a carry), it is easy to construct an *n*-bit sum by combining adder circuits (see Figure ).



Building an n-bit adder using half and full adders.

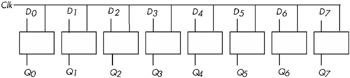
#### The D Flip-Flop

The only problem with the S/R flip-flop is that to be able to remember either a zero or a one value, you must have two different inputs. A memory cell would be more valuable to us if we could specify the data value to remember with one input value and supply a second *clock input*value to *latch*the data input value. [3]This type of flip-flop, the D flip-flop ( *D*stands for *data*) uses the circuit in Figure .

  
Figure: Implementing a D flip-flop with NAND gates

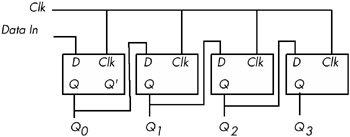
Assuming you fix the *Q*and *Q'*outputs to either 0/1 or 1/0, sending a *clock pulse*that goes from zero to one and back to zero will copy the *D*input to the *Q*output (and set *Q'*to the inverse of *Q*). To see how this works, just note that the right half of the circuit diagram in Figure is an S/R flip-flop. If the data input is one while the clock line is high, this places a zero on the *S*input of the S/R flip-flop (and a one on the *R*input). Conversely, if the data input is zero while the clock line is high, this places a zero on the *R*input (and a one on the *S*input) of the S/R flip-flop, thus clearing the S/R flip-flop's output. Whenever the clock input is low, both the *S*and *R*input are high, and the outputs of the S/R flip-flop do not change.

Although remembering a single bit is often important, in most computer systems you will want to remember a group of bits. You can do this by combining several D flip-flops in parallel. Concatenating flip-flops to store an *n*-bit value forms a *register.*The electronic schematic in Figure shows how to build an 8-bit register from a set of D flip-flops.

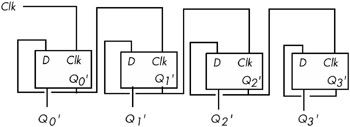
  
Figure: An 8-bit register implemented with eight D flip-flops

Note that the eight D flip-flops in Figure use a common clock line. This diagram does not show the *Q'*outputs on the flip-flops because they are rarely required in a register.

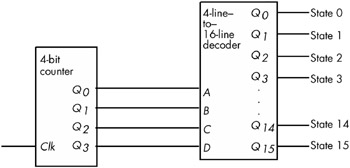
D flip-flops are useful for building many sequential circuits beyond simple registers. For example, you can build a *shift register*that shifts the bits one position to the left on each clock pulse. A 4-bit shift register appears in Figure .

  
Figure : A 4-bit shift register built from D flip-flops

You can even build a *counter*that counts the number of times the clock toggles from one to zero and back to one using flip-flops. The circuit in Figure implements a four bit counter using D flip-flops.

  
Figure : A 4-bit counter built from D flip-flops

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits. For example, you can build a simple state machine known as a sequencer by combining a counter and a decoder, as shown in Figure .

  
Figure : A simple 16-state sequencer

For each cycle of the clock in Figure , this sequencer activates one of its output lines. Those lines, in turn , may control other circuits. By 'firing' those other circuits on each of the 16 output lines of the decoder, we can control the order in which the circuits accomplish their tasks . This is essential in a CPU, as we often need to control the sequence of various operations. For example, it wouldn't be a good thing if the add(eax,ebx); instruction stored the result into EBX before fetching the source operand from EAX (or EBX). A simple sequencer such as this one can tell the CPU when to fetch the first operand, when to fetch the second operand, when to add them together, and when to store the result away.

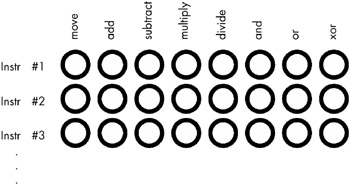
**Topic № 14**

**CPU Architecture**

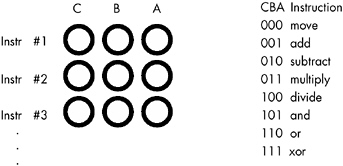
Basic CPU Design

A CPU is capable of executing a set of commands (or machine instructions), each of which accomplishes some small task. To execute a particular instruction, a CPU requires a certain amount of electronic circuitry specific to that instruction. Therefore, as you increase the number of instructions the CPU can support, you also increase the complexity of the CPU and you increase the amount of circuitry (logic gates) needed to support those instructions. To keep the number of logic gates on the CPU reasonably small (thus lowering the CPU's cost), CPU designers must restrict the number and complexity of the instructions that the CPU is capable of executing. This small set of instructions is the CPU's *instruction set*.

Programs in early computer systems were often 'hardwired' into the circuitry. That is, the computer's wiring determined exactly what algorithm the computer would execute. One had to rewire the computer in order to use the computer to solve a different problem. This was a difficult task, something that only electrical engineers were able to do. The next advance in computer design was the programmable computer system, one that allowed a computer operator to easily 'rewire' the computer system using sockets and plug wires (a *patch board*system). A computer program consisted of rows of sockets, with each row representing one operation during the execution of the program. The programmer could determine which of several instructions would be executed by plugging a wire into the particular socket for the desired instruction.

  
Figure: Patch board programming

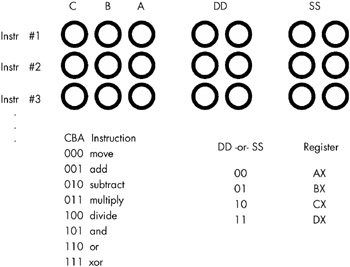
Of course, a major problem with this scheme was that the number of possible instructions was severely limited by the number of sockets one could physically place on each row. CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required for specifying *n*different instructions from *n*sockets to log 2( *n*) sockets. They did this by assigning a unique numeric code to each instruction and then representing each code as a binary number (for example, Figure shows how to represent eight instructions using only three bits).

  
Figure : Encoding instructions

The example in Figure requires eight logic functions to decode the *A, B,*and *C*bits on the patch board, but the extra circuitry (a single three-line-to-eight-line decoder) is worth the cost because it reduces the total number of sockets from eight to three for each instruction.

Of course, many CPU instructions do not stand alone. For example, a move instruction is a command that moves data from one location in the computer to another, such as from one register to another. The move instruction requires two operands: a source operand and a destination operand. The CPU's designer usually encodes the source and destination operands as part of the machine instruction, with certain sockets corresponding to the source and certain sockets corresponding to the destination.

Figure : shows one possible combination of sockets that would handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, and so on. This scheme allows the encoding of 128 different instructions with just seven sockets per instruction.

  
Figure : Encoding instructions with source and destination fields

One of the primary advances in computer design was the invention of the *stored program computer*. A big problem with patch-board programming was that the number of machine instructions in a program was limited by the number of rows of sockets available on the machine. Early computer designers recognized a relationship between the sockets on the patch board and bits in memory. They figured they could store the numeric equivalent of a machine instruction in main memory, fetch the instruction's numeric equivalent from memory when the CPU wanted to execute the instruction, and then load that binary number into a special register to decode the instruction.

The trick was to add additional circuitry, called the control unit (CU), to the CPU. The control unit uses a special register, the instruction pointer, that holds the address of an instruction's numeric code (also known as an *operation code*or *opcode*). The control unit fetches this instruction's opcode from memory and places it in the instruction decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution. This process repeats for each instruction the program executes.

The goal of the CPU's designer is to assign an appropriate number of bits to the opcode's instruction field and to its operand fields. Choosing more bits for the instruction field lets the opcode encode more instructions, just as choosing more bits for the operand fields lets the opcode specify a larger number of operands (often memory locations or registers). However, some instructions have only one operand, while others don't have any operands at all. Rather than waste the bits associated with these operand fields for instructions that don't have the maximum number of operands, the CPU designers often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family is a good example of this, with machine instructions ranging from 1 to almost 15 bytes long.

## Executing Instructions, Step by Step

To be able to write great code, you need to understand how a CPU executes individual machine instructions. To that end, let's consider four represen-tative 80x86 instructions: mov, add, loop , and jnz (jump if not zero). By understanding these four instructions, you can get a good feel for how a CPU executes all the instructions in the instruction set.

The mov instruction copies the data from the source operand to the destination operand. The add instruction adds the value of its source operand to its destination operand. The loop and jnz instructions are *conditional-jump*instructions - they test some condition and then jump to some other instruction in memory if the condition is true, or continue with the next instruction if the condition is false. The jnz instruction tests a Boolean variable within the CPU known as the *zero flag*and either transfers control to the target instruction if the zero flag contains zero, or continues with the next instruction if the zero flag contains one. The program specifies the address of the target instruction (the instruction to jump to) by specifying the distance, in bytes, from the jnz instruction to the target instruction in memory.

The loop instruction decrements the value of the ECX register and transfers control to a target instruction if ECX does not contain zero (after the decrement). This is a good example of a *Complex Instruction Set Computer*(CISC) instruction because it does multiple operations:

1. It subtracts one from ECX.
2. It does a conditional jump if ECX does not contain zero.

That is, loop is roughly equivalent to the following instruction sequence:

sub( 1, ecx ); // On the 80x86, the sub instruction sets the zero flag jnz SomeLabel; // the result of the subtraction is zero.

To execute the mov, add, jnz , and loop instructions, the CPU has to execute a number of different steps. Although each 80x86 CPU is different and doesn't necessarily execute the exact same steps, these CPUs do execute a similar sequence of operations. Each operation requires a finite amount of time to execute, and the time required to execute the entire instruction generally amounts to one clock cycle per operation or stage (as we usually refer to each of these steps) that the CPU executes. Obviously, the more steps needed for an instruction, the slower it will run. Complex instructions generally run slower than simple instructions, because complex instructions usually have many execution stages.

### 9.3.1 The mov Instruction

Although each CPU is different and may run different steps when executing instructions, the 80x86 mov( *srcReg,destReg*); instruction could use the following execution steps:

1. Fetch the instruction's opcode from memory.
2. Update the EIP (extended instruction pointer) register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the data from the source register (srcReg) .
5. Store the fetched value into the destination register (destReg) .

The mov( *srcReg,destMem*); instruction could use the following execution steps:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
5. Update EIP to point at the first byte beyond the operand that follows the opcode.
6. If the mov instruction uses a complex addressing mode (for example, the indexed addressing mode), compute the effective address of the destination memory location.
7. Fetch the data from srcReg .
8. Store the fetched value into the destination memory location.

Note that a mov( *srcMem,destReg*); instruction is very similar, simply swapping the register access for the memory access in these steps.

The mov( *constant,destReg*); instruction could use the following execution steps:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the constant associated with the source operand from the memory location immediately following the opcode.
5. Update EIP to point at the first byte beyond the constant that follows the opcode.
6. Store the constant value into the destination register.

Assuming each step requires one clock cycle for execution, this sequence will require six clock cycles to execute.

The mov( *constant,destMem*); instruction could use the following execution steps:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
5. Update EIP to point at the first byte beyond the operand that follows the opcode.
6. Fetch the constant operand's value from the memory location immediately following the displacement associated with the memory operand.
7. Update EIP to point at the first byte beyond the constant.
8. If the mov instruction uses a complex addressing mode (for example, the indexed addressing mode), compute the effective address of the destination memory location.
9. Store the constant value into the destination memory location.

### The add Instruction

The add instruction is a little more complex. Here's a typical set of operations that the add( *srcReg,destReg*); instruction must complete:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the value of the source register and send it to the arithmetic logical unit (ALU), which handles arithmetic on the CPU.
5. Fetch the value of the destination register operand and send it to the ALU.
6. Instruct the ALU to add the values.
7. Store the result back into the destination register operand.
8. Update the flags register with the result of the addition operation.

|  |  |
| --- | --- |
| Note | *The flags register, also known as the condition-codes register or program-status word, is an array of Boolean variables in the CPU that tracks whether the previous instruction produced an overflow, a zero result, a negative result, or other such condition.* |

If the source operand is a memory location instead of a register, and the add instruction takes the form add( *srcMem,destReg*); then the operation is slightly more complicated:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
5. Update EIP to point at the first byte beyond the operand that follows the opcode.
6. If the add instruction uses a complex addressing mode (for example, the indexed addressing mode), compute the effective address of the source memory location.
7. Fetch the source operand's data from memory and send it to the ALU.
8. Fetch the value of the destination register operand and send it to the ALU.
9. Instruct the ALU to add the values.
10. Store the result back into the destination register operand.
11. Update the flags register with the result of the addition operation.

If the source operand is a constant and the destination operand is a register, the add instruction takes the form add( *constant,destReg*); and here is how the CPU might deal with it:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the constant operand that immediately follows the opcode in memory and send it to the ALU.
5. Update EIP to point at the first byte beyond the constant that follows the opcode.
6. Fetch the value of the destination register operand and send it to the ALU.
7. Instruct the ALU to add the values.
8. Store the result back into the destination register operand.
9. Update the flags register with the result of the addition operation.

This instruction sequence requires nine cycles to complete.

If the source operand is a constant, and the destination operand is a memory location, then the add instruction takes the form add( *constant, destMem*); and the operation is slightly more complicated:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement associated with the memory operand from memory immediately following the opcode.
5. Update EIP to point at the first byte beyond the operand that follows the opcode.
6. If the add instruction uses a complex addressing mode (for example, the indexed addressing mode), compute the effective address of the destination memory location.
7. Fetch the constant operand that immediately follows the memory operand's displacement value and send it to the ALU.
8. Fetch the destination operand's data from memory and send it to the ALU.
9. Update EIP to point at the first byte beyond the constant that follows the memory operand.
10. Instruct the ALU to add the values.
11. Store the result back into the destination memory operand.
12. Update the flags register with the result of the addition operation.

This instruction sequence requires 11 or 12 cycles to complete, depending on whether the effective address computation is necessary.

### The jnz Instruction

Because the 80x86 jnz instruction does not allow different types of operands, there is only one sequence of steps needed for this instruction. The jnz label; instruction might use the following sequence of steps:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the displacement value following the instruction.
3. Decode the opcode to see what instruction it specifies.
4. Fetch the displacement value (the jump distance) and send it to the ALU.
5. Update the EIP register to hold the address of the instruction following the displacement operand.
6. Test the zero flag to see if it is clear (that is, if it contains zero).
7. If the zero flag was clear, copy the value in EIP to the ALU.
8. If the zero flag was clear, instruct the ALU to add the displacement and EIP values.
9. If the zero flag was clear, copy the result of the addition back to the EIP.

Notice how the jnz instruction requires fewer steps, and thus runs in fewer clock cycles, if the jump is not taken. This is very typical for conditional-jump instructions.

### The loop Instruction

Because the 80x86 loop instruction does not allow different types of operands, there is only one sequence of steps needed for this instruction. The 80x86 loop instruction might use an execution sequence like the following:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the displacement operand following the opcode.
3. Decode the opcode to see what instruction it specifies.
4. Fetch the value of the ECX register and send it to the ALU.
5. Instruct the ALU to decrement this value.
6. Send the result back to the ECX register. Set a special internal flag if this result is nonzero.
7. Fetch the displacement value (the jump distance) following the opcode in memory and send it to the ALU.
8. Update the EIP register with the address of the instruction following the displacement operand.
9. Test the special internal flag to see if ECX was nonzero.
10. If the flag was set (that is, it contains one), copy the value in EIP to the ALU.
11. If the flag was set, instruct the ALU to add the displacement and EIP values.
12. If the flag was set, copy the result of the addition back to the EIP register.

As with the jnz instruction, you'll note that the loop instruction executes more rapidly if the branch is not taken and the CPU continues execution with the instruction that immediately follows the loop instruction.

Parallelism - The Key to Faster Processing

If we can reduce the amount of time it takes for a CPU to execute the individual instructions appearing in that CPU's instruction set, it should be fairly clear that an application containing a sequence of those instructions will also run faster (compared with executing that sequence on a CPU whose individual instructions have not been sped up). Though the steps associated with a particular instruction's execution are usually beyond the control of a software engineer, understanding those steps and why the CPU designer chose an particular implementation for an instruction can help you pick more appropriate instruction sequences that execute faster.

An early goal of the *Reduced Instruction Set Computer*(RISC) processors was to execute one instruction per clock cycle, on the average. However, even if a RISC instruction is simplified, the actual execution of the instruction still requires multiple steps. So how could they achieve the goal? The answer is parallelism.

Consider the following steps for a mov( *srcReg,destReg*); instruction:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the data from srcReg .
5. Store the fetched value into the destination register (destReg) .

There are five stages in the execution of this instruction, with certain dependencies existing between most of the stages. For example, the CPU must fetch the instruction's opcode from memory before it updates the EIP register instruction with the address of the byte beyond the opcode. Likewise, the CPU won't know that it needs to fetch the value of the source register until it decodes the instruction's opcode. Finally, the CPU must fetch the value of the source register before it can store the fetched value in the destination register.

All but one of the stages in the execution of this mov instruction are *serial*. That is, the CPU must execute one stage before proceeding to the next . The one exception is step 2, updating the EIP register. Although this stage must follow the first stage, none of the following stages in the instruction depend upon this step. Therefore, this could be the third, forth, or fifth step in the calculation and it wouldn't affect the outcome of the instruction. Further, we could execute this step concurrently with any of the other steps, and it wouldn't affect the operation of the mov instruction. By doing two of the stages in parallel, we can reduce the execution time of this instruction by one clock cycle.

### The Prefetch Queue

The key to improving the speed of a processor is to perform operations in parallel. If we were able to do two operations on each clock cycle, the CPU would execute instructions twice as fast when running at the same clock speed. However, simply deciding to execute two operations per clock cycle doesn't make accomplishing it easy.

As you have seen, the steps of the add instruction that involve adding two values and then storing their sum cannot be done concurrently, because you cannot store the sum until after you've computed it. Furthermore, there are some resources that the CPU cannot share between steps in an instruction.

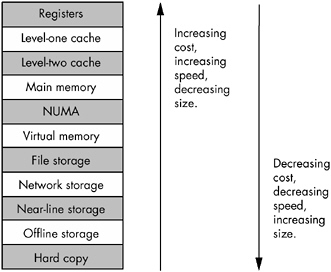
There is only one data bus, and the CPU cannot fetch an instruction's opcode while it is trying to store some data to memory. In addition, many of the steps that make up the execution of an instruction share *functional units*in the CPU. Functional units are groups of logic that perform a common operation, such as the *arithmetic logical unit*(ALU) and the *control unit*(CU). A functional unit is only capable of one operation at a time. You cannot do two operations concurrently that use the same functional unit. To design a CPU that executes several steps in parallel, one must arrange those steps to reduce potential conflicts, or add additional logic so the two (or more) operations can occur simultaneously by executing in different functional units.

**Topic № 15**

**Memory Architecture and Organization**

The Memory Hierarchy

Most modern programs benefit by having a large amount of very fast memory. Unfortunately, as a memory device gets larger, it tends to be slower. For example, cache memories are very fast, but they are also small and expensive. Main memory is inexpensive and large, but is slow, requiring wait states. The memory hierarchy provides a way to compare the cost and performance of memory. Figure diagrams one variant of the memory hierarchy.

  
Figure : The memory hierarchy

At the top level of the memory hierarchy are the CPU's general-purpose *registers*. The registers provide the fastest access to data possible on the CPU. The register file is also the smallest memory object in the hierarchy (for example, the 80x86 has just eight general-purpose registers). Because it is impossible to add more registers to a CPU, registers are also the most expensive memory locations. Even if we count the FPU, MMX/AltaVec, SSE/SIMD, and other CPU registers in this portion of the memory hierarchy, this does not change the fact that CPUs have a very limited number of registers, and the cost per byte of register memory is quite high.

Working our way down, the *level-one cache*system is the next highest performance subsystem in the memory hierarchy. As with registers, the CPU manufacturer usually provides the level-one (L1) cache on the chip, and you cannot expand it. The size is usually small, typically between 4 KB and 32 KB, though this is much larger than the register memory available on the CPU chip. Although the L1 cache size is fixed on the CPU, the cost per cache byte is much lower than the cost per register byte because the cache contains more storage than is available in all the combined registers, and the system designer's cost of both memory types is the price of the CPU.

*Level-two cache*is present on some CPUs, but not all. For example, most Pentium II, III, and IV CPUs have a level-two (L2) cache as part of the CPU package, but some of Intel's Celeron chips do not. The L2 cache is generally much larger than the L1 cache (for example, 256 KB to 1 MB as compared with 4 KB to 32 KB). On CPUs with a built-in L2 cache, the cache is not expandable. It is still lower in cost than the L1 cache because we amortize the cost of the CPU across all the bytes in the two caches, and the L2 cache is larger.

The *main-memory*subsystem comes below the L2 cache system in the memory hierarchy. [1]Main memory is the general-purpose, relatively low-cost memory found in most computer systems. Typically, this memory is DRAM or some similarly inexpensive memory. However, there are many differences in main memory technology that result in differences in speed. The main memory types include standard DRAM, synchronous DRAM (SDRAM), double data rate DRAM (DDRAM), and Rambus DRAM (RDRAM). Generally, though, you won't find a mixture of these technologies in the same computer system.

Below main memory is the *NUMA*memory subsystem. NUMA, which stands for Non-Uniform Memory Access, is a bit of a misnomer. The term NUMA implies that different types of memory have different access times, and so it is descriptive of the entire memory hierarchy. In Figure 11-1, however, the term NUMA is used to describe blocks of memory that are electronically similar to main memory but, for one reason or another, operate significantly slower than main memory. A good example of NUMAmemory is the memory on a video display card. Another example is flash memory, which has significantly slower access and transfer times than standard semiconductor RAM. Other peripheral devices that provide a block of memory to be shared between the CPU and the peripheral usually have slow access times, as well.

Most modern computer systems implement a *virtual memory*scheme that simulates main memory using a mass storage disk drive. A virtual memory subsystem is responsible for transparently copying data between the disk and main memory as needed by programs. While disks are significantly slower than main memory, the cost per bit is also three orders of magnitude lower for disks. Therefore, it is far less expensive to keep data on magnetic storage than in main memory.

*File storage*also uses disk media to store program data. However, whereas the virtual memory subsystem is responsible for handling data transfer between disk and main memory as programs require, it is the program's responsibility to store and retrieve file-storage data. In many instances, it is a bit slower to use file-storage memory than it is to use virtual memory, hence the lower position of file-storage memory in the memory hierarchy. [2]

Next comes *network storage*. At this level in the memory hierarchy, programs keep data on a different memory system that connects to the computer system via a network. Network storage can be virtual memory, file-storage memory, or a memory system known as *distributed shared memory*(DSM), where processes running on different computer systems share data stored in a common block of memory and communicate changes to that block across the network.

Virtual memory, file storage, and network storage are examples of so-called *online memory subsystems*. Memory access within these memory subsystems is slower than accessing the main-memory subsystem. However, when a program requests data from one of these three memory subsystems, the memory device will respond to the request as quickly as its hardware allows. This is not true for the remaining levels in the memory hierarchy.

The *near-line*and *offline storage*subsystems may not be ready to respond to a program's request for data immediately. An offline storage system keeps its data in electronic form (usually magnetic or optical), but on storage media that are not necessarily connected to the computer system that needs the data. Examples of offline storage include magnetic tapes, disk cartridges, optical disks, and floppy diskettes. Tapes and removable media are among the most inexpensive electronic data storage formats available. Hence, these media are great for storing large amounts of data for long periods. When a program needs data from an offline medium, the program must stop and wait for someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?).

Near-line storage uses the same types of media as offline storage, but rather than requiring an external source to mount the media before its data is available for access, the near-line storage system holds the media in a special robotic jukebox device that can automatically mount the desired media when a program requests it.

Hard-copy storage is simply a printout, in one form or another, of data. If a program requests some data, and that data is present only in hard-copy form, someone will have to manually enter the data into the computer. Paper, or other hard-copy media, is probably the least expensive form of memory, at least for certain data types.

[1]Actually, some systems now offer an external level-three cache. External level-three caches are present on some systems where the L1 and L2 caches are part of the CPU package and the system implementor wants to add more cache to the system.

[2]Note, however, that in some degenerate cases virtual memory can be much slower than file access.

How the Memory Hierarchy Operates

The whole point of having the memory hierarchy is to allow reasonably fast access to a large amount of memory. If only a little memory were necessary, we'd use fast static RAM (the circuitry that cache memory uses) for everything. If speed wasn't an issue, we'd use virtual memory for everything. The whole point of having a memory hierarchy is to enable us to take advantage of the principles of *spatial locality of reference*and *temporality of reference*to move often-referenced data into fast memory and leave less- often-used data in slower memory. Unfortunately, during the course of a program's execution, the sets of oft-used and seldom-used data change. We cannot simply distribute our data throughout the various levels of the memory hierarchy when the program starts and then leave the data alone as the program executes. Instead, the different memory subsystems need to be able to adjust for changes in spatial locality or temporality of reference during the program's execution by dynamically moving data between subsystems.

Moving data between the registers and memory is strictly a program function. The program loads data into registers and stores register data into memory using machine instructions like mov . It is strictly the programmer's or compiler's responsibility to keep heavily referenced data in the registers as long as possible, the CPU will not automatically place data in general-purpose registers in order to achieve higher performance.

Programs are largely unaware of the memory hierarchy between the register level and main memory. In fact, programs only explicitly control access to registers, main memory, and those memory-hierarchy subsystems at the file-storage level and below. In particular, cache access and virtual memory operations are generally transparent to the program. That is, access to these levels of the memory hierarchy usually occurs without any intervention on a program's part. Programs simply access main memory, and the hardware and operating system take care of the rest.

Of course, if every memory access that a program makes is to main memory, then the program will run slowly because modern DRAM mainmemory subsystems are much slower than the CPU. The job of the cache memory subsystems and of the CPU's cache controller is to move data between main memory and the L1 and L2 caches so that the CPU can quickly access oft-requested data. Likewise, it is the virtual memory subsystem's responsibility to move oft- requested data from hard disk to main memory (if even faster access is needed, the caching subsystem will then move the data from main memory to cache).

With few exceptions, most memory subsystem accesses take place transparently between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the L1 cache subsystem takes over. If the requested data is in the cache, then the L1 cache subsystem returns the data to the CPU, and that concludes the memory access. If the requested data is not present in the L1 cache, then the L1 cache subsystem passes the request on down to the L2 cache subsystem. If the L2 cache subsystem has the data, it returns this data to the L1 cache, which then returns the data to the CPU. Note that requests for the same data in the near future will be fulfilled by the L1 cache rather than the L2 cache because the L1 cache now has a copy of the data.

If neither the L1 nor the L2 cache subsystems have a copy of the data, then the request goes to main memory. If the data is found in main memory, then the main-memory subsystem passes this data to the L2 cache, which then passes it to the L1 cache, which then passes it to the CPU. Once again, the data is now in the L1 cache, so any requests for this data in the near future will be fulfilled by the L1 cache.

If the data is not present in main memory, but is present in virtual memory on some storage device, the operating system takes over, reads the data from disk or some other device (such as a network storage server), and passes the data to the main-memory subsystem. Main memory then passes the data through the caches to the CPU in the manner that we've seen.

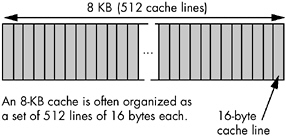
Because of spatial locality and temporality, the largest percentage of memory accesses take place in the L1 cache subsystem. The next largest percentage of accesses takes place in the L2 cache subsystem. The most infrequent accesses take place in virtual memory.

## Cache Architecture

Up to this point, we have treated the cache as a magical place that automatically stores data when we need it, perhaps fetching new data as the CPU requires it. But how exactly does the cache do this? And what happens when the cache is full and the CPU is requesting additional data not in the cache? In this section, we'll look at the internal cache organization and try to answer these questions, along with a few others.

Because programs only access a small amount of data at a given time, a cache that is the same size as the typical amount of data that programs access can provide very high-speed data access. Unfortunately, the data that programs want rarely sits in contiguous memory locations. Usually the data is spread out all over the address space. Therefore, cache design has to accommodate the fact that the cache must map data objects at widely varying addresses in memory.

As noted in the previous section, cache memory is not organized in a single group of bytes. Instead, cache memory is usually organized in blocks of *cache lines,*with each line containing some number of bytes (typically a small power of two like 16, 32, or 64), as shown in Figure .

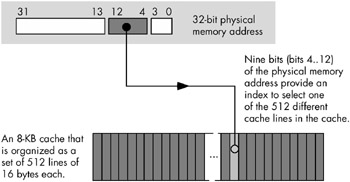
  
Figure : Possible organization of an 8-KB cache

Because of this 512 —16-byte cache organization found in Figure , we can attach a different noncontiguous address to each of the cache lines. Cache line 0 might correspond to addresses $10000..$1000F and cache line 1 might correspond to addresses $21400..$2140F. Generally, if a cache line is *n*bytes long, that cache line will hold *n*bytes from main memory that fall on an *n*-byte boundary. In the example in Figure , the cache lines are 16 bytes long, so a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory (in other words, the LO four bits of the address of the first byte in the cache line are always zero).

When the cache controller reads a cache line from a lower level in the memory hierarchy, where does the data go in the cache? The answer is determined by the caching scheme in use. There are three different cache schemes: *direct-mapped cache*, *fully associative cache*, and *n-way set associative cache*.

### Direct-Mapped Cache

In a *direct-mapped cache*(also known as the *one-way set associative cache*), a block of main memory is always loaded into the exact same cache line. Generally, a small number of bits in the data's memory address determines which cache line will hold the data. For example, Figure shows how the cache controller could select the appropriate cache line for an 8-KB cache with 512 16-byte cache lines and a 32-bit main-memory address. Because there are 512 cache lines, it requires 9 bits to select one of the cache lines (2 9= 512). This example uses bits 4 through 12 to determine which cache line to use ( assuming we number the cache lines from 0 to 511), while bits 0 through 3 of the original memory address determine the particular byte within the 16-byte cache line.

  
Figure 11-3: Selecting a cache line in a direct-mapped cache

The direct-mapped cache scheme is very easy to implement. Extracting nine (or some other number of) bits from the memory address and using the result as an index into the array of cache lines is trivial and fast. However, direct-mapped caches suffer from a few problems.

Perhaps the biggest problem with a direct-mapped cache is that it may not make effective use of all the cache memory. For example, the cache scheme in Figure 11-3 maps address zero to cache line 0. It also maps addresses $2000 (8 KB), $4000 (16 KB), $6000 (24 KB), and $8000 (32 KB) to cache line 0. In fact, it maps every address that is an even multiple of eight kilobytes to cache line 0. This means that if a program is constantly accessing data at addresses that are even multiples of 8 KB and not accessing any other locations, the system will only use cache line 0, leaving all the other cache lines unused. Each time the CPU requests data at an address that is mapped to cache line 0, but whose corresponding data is not present in cache line 0 (an address that is not an even multiple of 8 KB), the CPU will have to go down to a lower level in the memory hierarchy to access the data. In this extreme case, the cache is effectively limited to the size of one cache line.

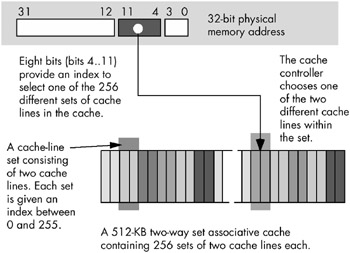
### Fully Associative Cache

The most flexible cache system is the fully associative cache. In a fully associative cache subsystem, the caching controller can place a block of bytes in any one of the cache lines present in the cache memory. While this is a very flexible system, the flexibility required is not without cost. The extra circuitry to achieve full associativity is expensive and, worse , can slow down the memory subsystem. Most L1 and L2 caches are not fully associative for this reason.

### n-Way Set Associative Cache

If a fully associative cache organization is too complex, too slow, and too expensive to implement, but a direct-mapped cache organization isn't as good as we'd like, one might ask if there is a compromise that doesn't have the drawbacks of a direct-mapped approach or the complexity of a fully associative cache. The answer is yes; we can create an *n*-way set associative cache that is a compromise between these two extremes. In an *n*-way set associative cache, the cache is broken up into *sets*of cache lines. The CPU determines the particular set to use based on some subset of the memory address bits, just as in the direct-mapping scheme. Within each cache line set, there are *n*cache lines. The caching controller uses a fully associative mapping algorithm to determine which one of the *n*cache lines within the set to use.

For example, an 8-KB two-way set associative cache subsystem with 16-byte cache lines organizes the cache into 256 cache-line sets with two cache lines each. ('Two-way' means that each set contains two cache lines.) Eight bits from the memory address determine which one of these 256 different sets holds the cache line that will contain the data. Once the cache-line set is determined, the cache controller then maps the block of bytes to one of the two cache lines within the set (see Figure ).

  
Figure : A two-way set associative cache

The advantage of a two-way set associative cache over a direct-mapped cache is that two different memory addresses located on 8-KB boundaries (addresses having the same value in bits 4 through 11) can both appear simultaneously in the cache. However, a conflict will occur if you attempt to access a third memory location at an address that is an even multiple of 8 KB.

A two-way set associative cache is much better than a direct-mapped cache and it is considerably less complex than a fully associative cache. However, if you're still getting too many conflicts, you might consider using a four-way set associative cache, which puts four associative cache lines in each cache-line set. In an 8-KB cache like the one in Figure , a four-way set associative cache scheme would have 128 cache-line sets with four cache lines each. This would allow the cache to maintain up to four different blocks of data without a conflict, each of which would map to the same cache line in a direct-mapped cache.

The more cache lines we have in each cache-line set, the closer we come to creating a fully associative cache, with all the attendant problems of complexity and speed. Most cache designs are direct-mapped, two-way set associative, or four-way set associative. The various members of the 80x86 family make use of all three.

### Matching the Caching Scheme to the Type of Data Access

Although this chapter has made the direct-mapped cache look bad, it is, in fact, very effective for many types of data. In particular, the direct-mapped cache is very good for data that you access in a sequential rather than random fashion. Because the CPU typically executes instructions in a sequential fashion, instruction bytes can be stored very effectively in a direct-mapped cache. However, because programs tend to access data more randomly than code, a two-way or four-way set associative cache usually makes a better choice for data accesses than a direct-mapped cache.

Because data and machine instruction bytes usually have different access patterns, many CPU designers use separate caches for each. For example, a CPU designer could choose to implement an 8-KB instruction cache and an 8-KB data cache rather than a single 16-KB unified cache. The advantage of dividing the cache size in this way is that the CPU designer could use a caching scheme more appropriate to the particular values that will be stored in each cache. The drawback is that the two caches are now each half the size of a unified cache, which may cause more cache misses than would occur with a unified cache. The choice of an appropriate cache organization is a difficult one and can only be made after analyzing many running programs on the target processor. How to choose an appropriate cache format is beyond the scope of this book, but be aware that it's not a choice you can make just by reading a textbook .

### Cache Line Replacement Policies

Thus far, we've answered the question, 'Where do we put a block of data in the cache?' An equally important question we've ignored until now is, 'What happens if a cache line isn't available when we want to put a block of data in that cache line?'

For a direct-mapped cache architecture, the answer is trivial. The cache controller simply replaces whatever data was formerly in the cache line with the new data. Any subsequent reference to the old data will result in a cache miss , and the cache controller will then have to bring that old data back into the cache by replacing whatever data is in the appropriate cache line at that time.

For a two-way set associative cache, the replacement algorithm is a bit more complex. Whenever the CPU references a memory location, the cache controller uses some subset of the address' bits to determine the cache-line set that should be used to store the data. Using some fancy circuitry, the caching controller determines whether the data is already present in one of the two cache lines in the destination set. If the data is not present, the CPU has to bring the data in from memory. Because the main memory data can go into either cache line, the controller has to pick one of the two lines to use. If either or both of the cache lines are currently unused, the controller simply picks an unused line. However, if both cache lines are currently in use, then the cache controller must pick one of the cache lines and replace its data with the new data.

How does the controller choose which of the two cache lines to replace? Ideally, we'd like to keep the cache line whose data will be referenced first and replace the other cache line. Unfortunately, neither the cache controller nor the CPU is omniscient - they cannot predict which of the lines is the best one to replace.

To understand how the cache controller makes this decision, remember the principle of temporality: if a memory location has been referenced recently, it is likely to be referenced again in the very near future. This implies the following corollary: *if a memory location has not been accessed in a while, it is likely to be a long time before the CPU accesses it again*. Therefore, a good replacement policy that many caching controllers use is the *least recently used*(LRU) algorithm. An LRU policy is easy to implement in a two-way set associative cache system. All you need is to reserve a single bit for each set of two cache lines. Whenever the CPU accesses one of the two cache lines this bit is set to zero, and whenever the CPU accesses the other cache line, this bit is set to one. Then, when a replacement is necessary, this bit will indicate which cache line to replace, as it tracks the last cache line the program has accessed (and, because there are only two cache lines in the set, the inverse of this bit also tracks the cache line that was least recently used).

For four-way (and greater) set associative caches, maintaining the LRU information is a bit more difficult, which is one of the reasons the circuitry for such caches is more complex. Because of the complications that LRU can introduce on these associative caches, other replacement policies are sometimes used. Two of these other policies are *first-in, first-out*(FIFO) and *random*. These are easier to implement than LRU, but they have their own problems, which are beyond the scope of this book, but which a text on computer architecture or operating systems will discuss.

### Writing Data to Memory

What happens when the CPU writes data to memory? The simple answer, and the one that results in the quickest operation, is that the CPU writes the data to the cache. However, what happens when the cache line containing this data is subsequently replaced by data that is read from memory? If the modified contents of the cache line are not written to main memory prior to this replacement, then the data that was written to the cache line will be lost. The next time the CPU attempts to access that data it will reload the cache line with the old data that was never updated after the write operation.

Clearly any data written to the cache must ultimately be written to main memory as well. There are two common write policies that caches use: *write-back*and *write-through*. Interestingly enough, it is sometimes possible to set the write policy in software, as the write policy isn't always hardwired into the cache controller. However, don't get your hopes up. Generally the CPU only allows the BIOS or operating system to set the cache write policy, so unless you're the one writing the operating system, you won't be able to control the write policy.

The write-through policy states that any time data is written to the cache, the cache immediately turns around and writes a copy of that cache line to main memory. An important point to notice is that the CPU does not have to halt while the cache controller writes the data from cache to main memory. So unless the CPU needs to access main memory shortly after the write occurs, this writing takes place in parallel with the execution of the program. Furthermore, because the write-through policy updates main memory with the new value as rapidly as possible, it is a better policy to use when two different CPUs are communicating through shared memory.

Still, writing a cache line to memory takes some time, and it is likely that the CPU (or some CPU in a multiprocessor system) will want to access main memory during the write operation, so the write-through policy may not be a high-performance solution to the problem. Worse, suppose the CPU reads from and writes to the memory location several times in succession. With a write-through policy in place, the CPU will saturate the bus with cache-line writes, and this will have a very negative impact on the program's performance.

The second common cache write policy is the write-back policy. In this mode, writes to the cache are not immediately written to main memory; instead, the cache controller updates main memory at a later time. This scheme tends to be higher performance because several writes to the same cache line within a short time period do not generate multiple writes to main memory.

Of course, at some point the cache controller must write the data in cache to memory. To determine which cache lines must be written back to main memory, the cache controller usually maintains a *dirty bit*within each cache line. The cache system sets this bit whenever it writes data to the cache. At some later time, the cache controller checks this dirty bit to determine if it must write the cache line to memory. For example, whenever the cache controller replaces a cache line with other data from memory, it must first check the dirty bit, and if that bit is set, the controller must write that cache line to memory before going through with the cache-line replacement. Note that this increases the latency time when replacing a cache line. If the cache controller were able to write dirty cache lines to main memory while no other bus access was occurring, the system could reduce this latency during cache line replacement. Some systems actually provide this functionality, and others do not for economic reasons.

### Cache Use and Software

A cache subsystem is not a panacea for slow memory access. In order for a cache system to be effective, software must exhibit locality of reference (either spatial or temporal). Fortunately, real-world programs tend to exhibit locality of reference, so most programs will benefit from the presence of a cache in the memory subsystem. But while programs do exhibit locality of reference, this is often accidental; programmers rarely consider the memory-access patterns of their software when designing and coding. Unfortunately, application programmers who work under the assumption that the cache will magically improve the performance of their applications are missing one important point - a cache can actually *hurt*the performance of an application.

Suppose that an application accesses data at several different addresses that the caching controller would map to the exact same cache line. With each access, the caching controller must read in a new cache line (possibly flushing the old cache line back to memory if it is dirty). As a result, each memory access incurs the latency cost of bringing in a cache line from main memory. This degenerate case, known as *thrashing*, can slow down the program by one to two orders of magnitude, depending on the speed of main memory and the size of a cache line. Great code is written with the behavior of the cache in mind. A great programmer attempts to place oft-used variables in adjacent memory cells so those variables tend to fall into the same cache lines. A great programmer carefully chooses data structures (and access patterns to those data structures) to avoid thrashing. We'll take another look at thrashing a little later in this chapter.

Another benefit of the cache subsystem on modern 80x86 CPUs is that it automatically handles many misaligned data references. As you may recall, there is a penalty for accessing words or double-word objects at an address that is not an even multiple of that object's size. As it turns out, by providing some fancy logic, Intel's designers have eliminated this penalty as long as the data object is located completely within a cache line. However, if the object crosses a cache line, there will be a performance penalty for the memory access.

Virtual Memory, Protection, and Paging

In a modern operating system such as Mac OS, Linux, or Windows, it is very common to have several different programs running concurrently in memory. This presents several problems.

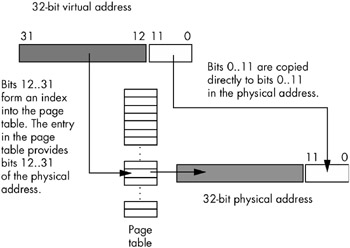
* How do you keep the programs from interfering with each other's memory?
* If one program expects to load a value into memory at address $1000, and a second program also expects to load a value into memory at address $1000, how can you load both values and execute both programs at the same time?
* What happens if the computer has 64 MB of memory, and we decide to load and execute three different applications, two of which require 32 MB and one that requires 16 MB (not to mention the memory that the operating system requires for its own purposes)?

The answers to all these questions lie in the virtual memory subsystem that modern processors support.

Virtual memory on CPUs such as the 80x86 gives each process its own 32-bit address space. [3]This means that address $1000 in one program is physically different from address $1000 in a separate program. The CPU achieves this sleight of hand by mapping the *virtual addresses*used by programs to different *physical addresses*in actual memory. The virtual address and the physical address don't have to be the same, and usually they aren't. For example, program 1's virtual address $1000 might actually correspond to physical address $215000, while program 2's virtual address $1000 might correspond to physical memory address $300000. How can the CPU do this? Easy, by using *paging*.

The concept behind paging is quite simple. First, you break up memory into blocks of bytes called *pages*. A page in main memory is comparable to a cache line in a cache subsystem, although pages are usually much larger than cache lines. For example, the 80x86 CPUs use a page size of 4,096 bytes.

After breaking up memory into pages, you use a lookup table to map the HO bits of a virtual address to the HO bits of the physical address in memory, and you use the LO bits of the virtual address as an index into that page. For example, with a 4,096-byte page, you'd use the LO 12 bits of the virtual address as the offset (0..4095) within the page, and the upper 20 bits as an index into a lookup table that returns the actual upper 20 bits of the physical address (see Figure ).

  
Figure : Translating a virtual address to a physical address

Of course, a 20-bit index into the page table would require over one million entries in the page table. If each of the over one million entries is a 32-bit value, then the page table would be 4 MB long. This would be larger than most of the programs that would run in memory! However, by using what is known as a multilevel page table, it is very easy to create a page table for most small programs that is only 8 KB long. The details are unimportant here. Just rest assured that you don't need a 4-MB page table unless your program consumes the entire 4 GB address space.

If you study Figure for a few moments, you'll probably discover one problem with using a page table - it requires two separate memory accesses in order to retrieve the data stored at a single physical address in memory: one to fetch a value from the page table, and one to read from or write to the desired memory location. To prevent cluttering the data or instruction cache with page-table entries, which increases the number of cache misses for data and instruction requests , the page table uses its own cache, known as the *translation lookaside buffer*(TLB). This cache typically has 32 entries on a Pentium family processor - enough to handle 128 KB of memory, or 32 pages, without a miss . Because a program typically works with less data than this at any given time, most page-table accesses come from the cache rather than main memory.

As noted, each entry in the page table contains 32 bits, even though the system really only needs 20 bits to remap each virtual address to a physical address. Intel, on the 80x86, uses some of the remaining 12 bits to provide some memory-protection information:

* One bit marks whether a page is read/write or read-only.
* One bit determines whether you can execute code on that page.
* A number of bits determine whether the application can access that page or if only the operating system can do so.
* A number of bits determine if the CPU has written to the page, but hasn't yet written to the physical memory address corresponding to the page entry (that is, whether the page is 'dirty' or not, and whether the CPU has accessed the page recently).
* One bit determines whether the page is actually present in physical memory or if it's stored on secondary storage somewhere.

Note that your applications do not have access to the page table (reading and writing the page table is the operating system's responsibility), and therefore they cannot modify these bits. However, operating systems like Windows may provide some functions you can call if you want to change certain bits in the page table (for example, Windows will allow you to set a page to read-only if you want to do so).

Beyond remapping memory so multiple programs can coexist in main memory, paging also provides a mechanism whereby the operating system can move infrequently used pages to secondary storage. Just as locality of reference applies to cache lines, it applies to pages in main memory as well. At any given time, a program will only access a small percentage of the pages in main memory that contain data and instruction bytes and this set of pages is known as the *working set*. Although this working set of pages varies slowly over time, for small periods of time the working set remains constant. Therefore, there is little need for the remainder of the program to consume valuable main memory storage that some other process could be using. If the operating system can save the currently unused pages to disk, the main memory they consume would be available for other programs that need it.

Of course, the problem with moving data out of main memory is that eventually the program might actually need that data. If you attempt to access a page of memory, and the page-table bit tells the memory management unit (MMU) that the page is not present in main memory, the CPU interrupts the program and passes control to the operating system. The operating system then analyzes the memory-access request and reads the corresponding page of data from the disk drive and copies it to some available page in main memory. This process is nearly identical to the process used by a fully associative cache subsystem, except that accessing the disk is much slower than accessing main memory. In fact, you can think of main memory as a fully associative write-back cache with 4,096-byte cache lines, which caches the data that is stored on the disk drive. Placement and replacement policies and other issues are very similar for caches and main memory.

However, that's as far as we'll go in exploring how the virtual memory subsystem works. If you're interested in further information, any decent textbook on operating system design will explain how the virtual memory subsystem swaps pages between main memory and the disk. Our main goal here is to realize that this process takes place in operating systems like Mac OS, Linux, and Windows, and that accessing the disk is very slow.

One important issue resulting from the fact that each program has a separate page table, and the fact that programs themselves don't have access to the page tables, is that programs cannot interfere with the operation of other programs. That is, a program cannot change its page tables in order to access data found in another process's *address space*. If your program crashes by overwriting itself, it cannot crash other programs at the same time. This is a big benefit of a paging memory system.

If two programs want to cooperate and share data, they can do so by placing such data in a memory area that is shared by the two processes. All they have to do is tell the operating system that they want to share some pages of memory. The operating system returns a pointer to each process that points at a segment of memory whose physical address is the same for both processes. Under Windows, you can achieve this by using *memory-mapped files*; see the operating system documentation for more details. Mac OS and Linux also support memory-mapped files as well as some special shared-memory operations; again, see the OS documentation for more details.

Although this discussion applies specifically to the 80x86 CPU, multilevel paging systems are common on other CPUs as well. Page sizes tend to vary from about 1 KB to 64 KB, depending on the CPU. For CPUs that support an address space larger than 4 GB, some CPUs use an *inverted page table*or a *three-level page table*. Although the details are beyond the scope of this chapter, rest assured that the basic principle remains the same - the CPU moves data between main memory and the disk in order to keep oft-accessed data in main memory as much of the time as possible. These other page-table schemes are good at reducing the size of the page table when an application uses only a fraction of the available memory space.

[3]Strictly speaking, you actually get a 36-bit address space on Pentium Pro and later processors, but Windows and Linux limit you to 32 bits, so we'll use that limitation here.

NUMA and Peripheral Devices

Although most of the RAM in a system is based on high-speed DRAM interfaced directly with the processor's bus, not all memory is connected to the CPU in this manner. Sometimes a large block of RAM is part of a peripheral device, and you communicate with that device by writing data to its RAM. Video display cards are probably the most common example of such a peripheral, but some network interface cards, USB controllers, and other peripherals also work this way. Unfortunately, the access time to the RAM on these peripheral devices is often much slower than the access time to main memory. In this section, we'll use the video card as an example, although NUMA performance applies to other devices and memory technologies as well.

A typical video card interfaces with a CPU via an AGP or PCI bus inside the computer system. The PCI bus typically runs at 33 MHz and is capable of transferring four bytes per bus cycle. Therefore, in burst mode, a video controller card is capable of transferring 132 MB per second, though few would ever come close to achieving this for technical reasons. Now compare this with main-memory access time. Main memory usually connects directly to the CPU's bus, and modern CPUs have an 800-MHz 64-bit-wide bus. If memory were fast enough, the CPU's bus could theoretically transfer 6.4 GB per second between memory and the CPU. This is about 48 times faster than the speed of transferring data across the PCI bus. Game programmers long ago discovered that it's much faster to manipulate a copy of the screen data in main memory and only copy that data to the video display memory whenever a vertical retrace occurs (about 60 times per second). This mechanism is much faster than writing directly to the video card memory every time you want to make a change.

Caches and the virtual memory subsystem operate in a transparent fashion, but NUMA memory does not, so programs that write to NUMA devices must minimize the number of accesses whenever possible (for example, by using an off-screen bitmap to hold temporary results). If you're actually storing and retrieving data on a NUMA device, like a flash memory card, you must explicitly cache the data yourself.