PRINCIPLES OF Computer Organization and Assembly Language

Using the Java™ Virtual Machine



PATRICK JUOLA

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 1
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

Principles of Computer Organization and Assembly Language

Using the Java™ Virtual Machine

PATRICK JUOLA

Duquesne University



Upper Saddle River, New Jersey 07458

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 2
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

Library of Congress Cataloging-in-Publication Data

Juola, Patrick
Principles of computer organization and Assembly language : using the Java virtual machine / Patrick Juola.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-148683-7
1. Computer organization. 2. Assembler language (Computer program language) 3. Java virtual machine. I. Title.
QA76.9.C643J96 2006
004.2′2–dc22

2006034154

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Associate Editor: *Carole Snyder*
Editorial Assistant: *Christianna Lee*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *Camille Trentacoste*
Production Editor: *Karen Ettinger*
Director of Creative Services: *Paul Belfanti*
Creative Director: *Juan Lopez*
Cover Art Director: *Jayne Conte*
Cover Designer: *Kiwi Design*
Cover Photo: *Getty Images, Inc.*
Managing Editor, AV Management and Production: *Patricia Burns*
Art Editor: *Gregory Dulles*
Manufacturing Manager, ESM: *Alexis Heydt-Long*
Manufacturing Buyer: *Lisa McDowell*
Executive Marketing Manager: *Robin O'Brien*
Marketing Assistant: *Mack Patterson*



© 2007 Pearson Education, Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, New Jersey 07458

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall® is a trademark of Pearson Education, Inc.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America
All other trademarks or product names are the property of their respective owners.

TRADEMARK INFORMATION
Java is a registered trademark of Sun Microsystems, Inc.
Pentium is a trademark of Intel Corporation.
Visual C++ is a registered trademark of Microsoft Corporation.
PowerPC is a registered trademark of IBM Corporation.

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-148683-7

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

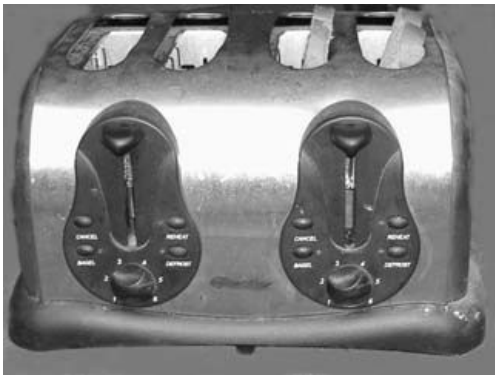
1

Computation and Representation

1.1 Computation



A computer



Also a computer

1.1.1 Electronic Devices

How many people really know what a computer is? If you asked, most people would point to a set of boxes on someone's desk (or perhaps in someone's briefcase)—probably a set of dull-looking rectangular boxes encased in gray or beige plastic, and surrounded by a tangle of wires and perhaps something resembling a TV. If pressed for detail, they would point at one particular box as "the computer." But, of course, there are also computers hidden in all sorts of everyday electronic gadgets to make sure that your car's fuel efficiency stays high enough, to interpret the signals from a DVD player, and possibly even to make sure your morning toast is the right shade of brown. To most people, though, a computer is still the box you buy at an electronics shop, with bits and bytes and gigahertz that are often compared, but rarely understood.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 4
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

In functional terms, a computer is simply a high-speed calculator capable of performing thousands, millions, or even billions of simple arithmetic operations per second from a stored program. Every thousandth of a second or so, the computer in your car reads a few key performance indicators from various sensors in the engine and adjusts the machine slightly to ensure proper functioning. The key to being of any use is at least partially in the sensors. The computer itself processes only electronic signals. The sensors are responsible for determining what's really going on under the hood and converting that into a set of electronic signals that describe, or represent, the current state of the engine. Similarly, the adjustments that the computer makes are stored as electronic signals and converted back into physical changes in the engine's working.

How can electronic signals "represent" information? And how exactly does a computer process these signals to achieve fine control without any physical moving parts or representation? Questions of representation such as these are, ultimately, the key to understanding both how computers work and how they can be deployed in the physical world.

1.1.2 Algorithmic Machines

The single most important concept in the operation of a computer is the idea of an **algorithm**: an unambiguous, step-by-step process for solving a problem or achieving a desired end. The ultimate definition of a computer does not rely on its physical properties, or even on its electrical properties (such as its transistors), but on its ability to represent and carry out algorithms from a stored program. Within the computer are millions of tiny circuits, each of which performs a specific well-defined task (such as adding two integers together or causing an individual wire or set of wires to become energized) when called upon. Most people who use or program computers are not aware of the detailed workings of these circuits.

In particular, there are several basic types of operations that a typical computer can perform. As computers are, fundamentally, merely calculating machines, almost all of the functions they can perform are related to numbers (and concepts representable by numbers). A computer can usually perform basic mathematical operations such as addition and division. It can also perform basic comparisons—is one number equal to another number? Is the first number less than the second? It can store millions or billions of pieces of information and retrieve them individually. Finally, it can adjust its actions based on the information retrieved and the comparisons performed. If the retrieved value is greater than the previous value, then (for example) the engine is running too hot, and a signal should be sent to adjust its performance.

1.1.3 Functional Components

System-Level Description

Almost any college bulletin board has a few ads that read something like "GREAT MACHINE! 3.0-GHz Intel Celeron D, 512 mg, 80-GB hard drive, 15-inch monitor, must sell to make car payment!" Like most ads, there's a fair bit of information in there that requires extensive unpacking to understand fully. For example, what part of a 15-inch monitor is actually 15 inches? (The length of the diagonal of the visible screen, oddly enough.) In order to understand the detailed workings of a computer, we must first understand the major components and their relations to each other (figure 1.1).

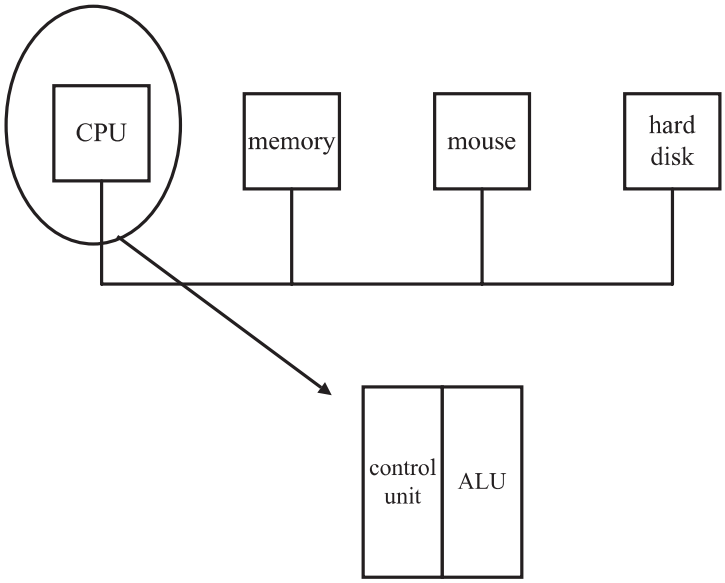


Figure 1.1 Major hardware components of a computer

Central Processing Unit

The heart of any computer is the **Central Processing Unit**, or CPU. This is usually a single piece of high-density circuitry built on single **integrated circuit** (IC) silicon chip (figure 1.2). Physically, it usually looks like a small piece of silicon, mounted on a plastic slab a few centimeters square, surrounded by metal pins. The plastic slab itself is mounted on the **motherboard**, an electronic circuit board consisting of a piece of plastic and metal tens of centimeters

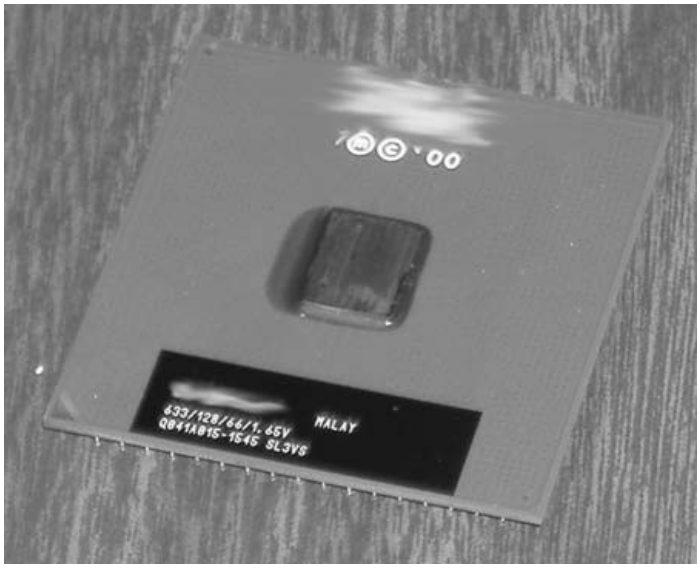


Figure 1.2 Photograph of a CPU chip

on a side, containing the CPU and a few other components that need to be placed near the CPU for speed and convenience. Electronically, the CPU is the ultimate controller of the computer, as well as the place where all calculations are performed. And, of course, it's the part of the computer that everyone talks and writes about—a 3.60-GHz Pentium 4 computer, like the Hewlett-Packard HP xw4200, is simply a computer whose CPU is a Pentium 4 chip and that runs at a speed of 3.60 gigahertz (GHz), or 3,600,000,000 **machine cycles** per second. Most of the basic operations a computer can perform take one machine cycle each, so another way of describing this is to say that a 3.60-GHz computer can perform just over 3.5 billion basic operations per second. At the time of writing, 3.60 GHz is a fast machine, but this changes very quickly with technological developments. For example, in 2000, a 1.0-GHz Pentium was the state of the art, and, in keeping with a long-standing rule of thumb (Moore's law) that computing power doubles every 18 months, one can predict the wide availability of 8-GHz CPUs by 2008.

SIDEBAR

Moore's Law

Gordon Moore, the cofounder of Intel, observed in 1965 that the number of transistors that could be put on a chip was doubling every year. In the 1970s, that pace slowed slightly, to a doubling every 18 months, but has been remarkably uniform since then, to the surprise of almost everyone, including Dr. Moore himself. Only in the past few years has the pace weakened.

The implications of smaller transistors (and increasing transistor density) are profound. First, the cost per square inch of a silicon chip itself has been relatively steady by comparison, so doubling the density will approximately halve the cost of a chip. Second, smaller transistors react faster, and components can be placed closer together, so that they can communicate with each other faster, vastly increasing the speed of the chip. Smaller transistors also consume less power, meaning longer battery life and lower cooling requirements, avoiding the need for climate-controlled rooms and bulky fans. Because more transistors can be placed on a chip, less soldering is needed to connect chips together, with an accordingly reduced chance of solder breakage and correspondingly greater overall reliability. Finally, the fact that the chips are smaller means that computers themselves can be made smaller, enough to make things like embedded controller chips and/or personal digital assistants (PDAs) practical. It is hard to overestimate the effect that Moore's law has had on the development of the modern computer. Moore's law by now is generally taken to mean, more simply, that the power of an available computer doubles every 18 months (for whatever reason, not just transistor density). A standard, even low-end, computer available off the shelf at the local store is faster, more reliable, and has more memory than the original Cray-1 supercomputer of 1973.

The problem with Moore's law is that it will not hold forever. Eventually, the laws of physics are likely to dictate that a transistor can't be any smaller than an atom (or something like that). More worrisome is what's sometimes called Moore's second law, which states that fabrication costs double every three years. As long as fabrication costs grow more slowly than computer power, the performance/cost ratio should remain reasonable. But the cost of investing in new chip technologies may make it difficult for manufacturers such as Intel to continue investing in new capital.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 7
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

CPUs can usually be described in families of technological progress; the Pentium 4, for example, is a further development of the Pentium, the Pentium II, and the Pentium III, all manufactured by the Intel corporation. Before that, the Pentium itself derived from a long line of numbered Intel chips, starting with the Intel 8088 and progressing through the 80286, 80386, and 80486. The so-called "x86 series" became the basis for the best-selling IBM personal computers (PCs and their clones) and is probably the most widely used CPU chip. Modern Apple computers use a different family of chips, the PowerPC G3 and G4, manufactured by a consortium of Apple, IBM, and Motorola (AIM). Older Apples and Sun workstations used chips from the Motorola-designed 68000 family.

The CPU itself can be divided into two or three main functional components. The **Control Unit** is responsible for moving data around within the machine For example, the Control Unit takes care of loading individual program instructions from memory, identifying individual instructions, and passing the instructions to other appropriate parts of the computer to be performed The **Arithmetic and Logical Unit** (ALU) performs all necessary arithmetic for the computer; it typically contains special-purpose hardware for addition, multiplication, division, and so forth. It also, as the name implies, performs all the logical operations, determining whether a given number is bigger or smaller than another number or checking whether two numbers are equal. Some computers, particularly older ones, have special-purpose hardware, sometimes on a separate chip from the CPU itself, to handle operations involving fractions and decimals. This special hardware is often called the **Floating Point Unit** or FPU (also called the Floating Point Processor or FPP). Other computers fold the FPU hardware onto the same CPU chip as the ALU and the Control Unit, but the FPU can still be thought of as a different module within the same set of circuitry.

Memory

Both the program to be executed and its data are stored in memory. Conceptually, memory can be regarded as a very long array or row of electromagnetic storage devices. These array locations are numbered, from 0 to a CPU-defined maximum, and can be addressed individually by the Control Unit to place data memory or to retrieve data from memory (figure 1.3). In addition, most modern machines allow high-speed devices such as disk drives to copy large blocks of data without needing the intervention of the Control Unit for each signal. Memory can be broadly divided into two types: **Read-Only Memory** (ROM), which is permanent, unalterable, and remains even after the power is switched off, and **Random Access Memory** (RAM), the contents of which can be changed by the CPU for temporary storage but usually disappears when the power does. Many machines have both kinds of memory; ROM holds standardized data and a basic version of the operating system that can be used to start the machine. More extensive programs are held in long-term storage such as disk drives and CDs, and loaded as needed into RAM for short-term storage and execution.

This simplified description deliberately hides some tricky aspects of memory that the hardware and operating system usually take care of for the user. (These issues also tend to be hardware-specific, so they will be dealt with in more detail in later chapters.) For example, different computers, even with identical CPUs, often have different amounts of memory. The amount of physical memory installed on a computer may be less than the maximum number of locations the CPU can address or, in odd cases, may even be more. Furthermore, memory located on the CPU chip

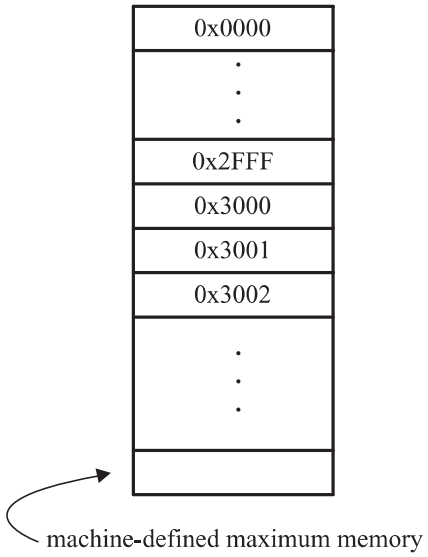


Figure 1.3 Block diagram of a linear array of memory cells

itself can typically be accessed much more quickly than memory located on separate chips, so a clever system can try to make sure that data is moved or copied as necessary to be available in the fastest memory when needed.

Input/Output (I/O) Peripherals

In addition to the CPU and memory, a computer usually contains other devices to read, display or store data, or more generally to interact with the outside world. These devices vary from commonplace keyboards and hard drives through more unusual devices like facsimile (FAX) boards, speakers, and musical keyboards to downright weird gadgets like chemical sensors, robotic arms, and security deadbolts. The general term for these gizmos is **peripherals**. For the most part, these devices have little direct effect on the architecture and organization of the computer itself; they are just sources and sinks for information. A keyboard, for instance, is simply a device to let the computer gather information from the user. From the point of view of the CPU designer, data is data, whether it came from the Internet, from the keyboard, or from a fancy chemical spectrum analyzer.

In many cases, a peripheral can be physically divided into two or more parts. For example, computers usually display their information to the user on some form of video monitor. The **monitor** itself is a separate device, connected via a cable to a video adapter board located inside the computer's casing. The CPU can draw pictures by sending command signals to the video board, which in turn will generate the picture and send appropriate visual signals over the video cable to the monitor itself. A similar process describes how the computer can load a file from many different kinds of hard drives via a SCSI (Small Computer System Interface) controller card, or interact via an Ethernet card with the millions of miles of wire that comprise the Internet. Conceptually, engineers draw a distinction between the device itself, the device cable (which is often just a wire), and the device controller, which is usually a board inside the computer—but to the programmer, they're usually all one device. Using this kind of logic, the entire Internet, with

its millions of miles of wire, is just "a device." With a suitably well-designed system, there's not much difference between downloading a file off the Internet or loading it from a hard drive.

Interconnections and Buses

In order for the data to move between the CPU, memory, and the peripherals, there must be connections. These connections, especially between separate boards, are usually groups of wires to allow multiple individual signals to be sent in a block. The original IBM-PC, for example, had eight wires to allow data to pass between the CPU and peripherals. A more modern computer's PCI (Peripheral Component Interconnect) bus has 64 data wires, allowing data to pass eight times as fast, even before increased computer speed is taken into account. These wires are usually grouped into what is called a **bus**, a single wire set connecting several different devices. Because it is shared (like an early party-line telephone), only one device can transmit data at a time, but the data is available to any connected device. Additional wires are used to determine which device should listen to the data and what exactly it should do when it gets it.

In general, the more devices attached to a single bus, the slower it runs. This is true for two main reasons. First, the more devices, the greater the possibility that two devices will have data to transmit at the same time, and thus that one device will have to wait its turn. Second, more devices usually means longer wires in the bus, which reduces the speed of the bus due to delays in propagation—the time it takes a signal to get from one end of the wire to the other. For this reason, many computers now have a multiple-bus design, in which, for example, the local bus connects the CPU with high-speed memory stored on the CPU's motherboard (often on the CPU chip itself). The system bus connects the memory board, the CPU motherboard, and an "expansion bus" interface board. The expansion bus, in turn, is a second bus that connects to other devices such as the network, the disk drives, the keyboard, and the mouse.

On particularly high-performance computers (such as the one in figure 1.4), there may be four or five separate buses, with one reserved for high-speed, data-intensive devices such as the network and video cards, while lower-speed devices such as the keyboard are relegated to a separate, slower bus.

Support Units

In addition to the devices mentioned already, a typical computer will have a number of components that are important to its physical condition. For example, inside the case (itself crucial for the physical protection of the delicate circuit boards) will be a power supply that converts the AC line voltage into an appropriately conditioned DC voltage for the circuit boards. There may also be a battery, particularly in laptops, to provide power when wall current is unavailable and to maintain memory settings. There is usually a fan to circulate air inside the case and to prevent components from overheating. There may also be other devices such as heat sensors (to control fan speed), security devices to prevent unauthorized use or removal, and often several wholly internal peripherals such as internal disk drives and CD readers.

1.2 Digital and Numeric Representations

1.2.1 Digital Representations and Bits

At a fundamental level, computer components, like so many other electronic components, have two stable states. Lights are on or off, switches are open or closed, and wires are either carrying current or not. In the case of computer hardware, individual components such as transistors and

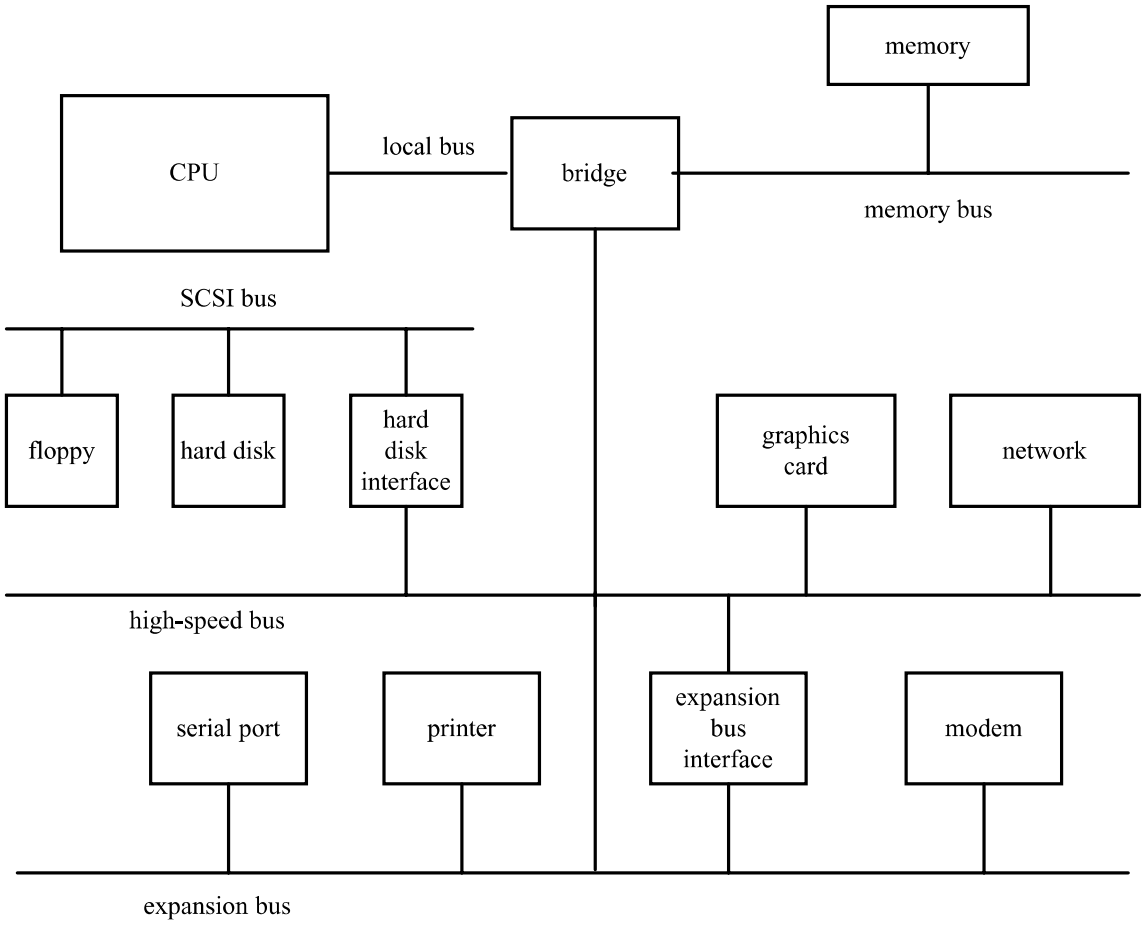


Figure 1.4 Block architecture of a high-speed, multibus computer

resistors are either at 0 volts relative to ground or at some other voltage (typically 5 volts above ground). These two states are usually held to represent the numbers 1 and 0, respectively. In the early stages of computer development, these values were hand-encoded by flipping mechanical switches. Today, high-speed transistors serve much the same purpose, but the representation of data in terms of these two values remains unchanged since the 1940s. Every such 1 or 0 is usually called a **bit**, an abbreviation for "binary digit." (Of course, the word "bit" is itself a normal English word, meaning "a very small amount"—which also decribes a "bit" of information.)

SIDEBAR

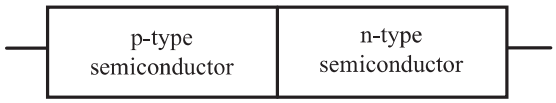
How Transistors Work

The single most important electrical component in the modern computer is the transistor, first invented by Bardeen, Brattain, and Shockley in 1947 at Bell Telephone Labs. (These men received the Nobel Prize for Physics in 1956 for this invention.) The fundamental idea involves some fairly high-powered (well, yes, Nobel-caliber) quantum physics, but it can be understood in terms of electron transport, as long as you don't need the actual equations. A transistor consists mostly of a type of material called a **semiconductor**, which occupies an uneasy middle ground between

good conductors (like copper) and bad conductors/good insulators (like glass). A key aspect of semiconductors is that their ability to transmit electricity can change dramatically with impurities (**dopants**) in the semiconductor.

For example, the element phosphorus, when added to pure silicon (a semiconductor), will donate electrons to the silicon. Since electrons have a negative charge, phosphorus is termed an **n-type** dopant, and phosphorus-doped silicon is sometimes called an **n-type semiconductor**. Aluminum, by contrast, is a **p-type** dopant and will actually remove—actually, lock up — electrons from the silicon matrix. The places where these electrons have been removed are sometimes called "holes" in the p-type semiconductor.

When you put a piece of n-type next to a piece of p-type semiconductor (the resulting widget is called a **diode**; figure 1.5), an interesting electrical effect occurs. An electrical current will not typically be able to pass through such a diode; the electrons carrying the current will encounter and "fall into" the holes. If you apply a **bias voltage** to this gadget, however, the extra electrons will fill the holes, allowing current to pass. This means that electricity can pass in only one direction through a diode, which makes it useful as an electrical **rectifier**.



Structure of diode



Symbol for diode

Figure 1.5 Diodes

A modern transistor is made like a semiconductor sandwich; a thin layer of p-type semiconductor between two slices of n-type semiconductor, or sometimes the reverse. (Yes, this is just two diodes back to back; figure 1.6). Under normal circumstances, current can't pass from the **emitter** to the **collector** as electrons fall into the holes. Applying a **bias voltage** to the **base** (the middle wire) will fill the holes so that electricity can pass. You can think of the base as a gate that can open or shut to allow electricity to flow or not. Alternatively, you can think of it as a valve in a hosepipe to control the amount of water it lets through. Turn it one way, and the electrical signal drops to a trickle. Turn it the other way, and it flows without hindrance.

The overall effect of the transistor is that a small change in voltage (at the base) will result in a very large change in the amount of current that flows from the emitter to the collector. This makes a transistor extremely useful for amplifying small signals. It also can function as a binary switch, the key advantage being that it has no moving parts and thus nothing to break. (It's also much, much faster to throw an electrical switch than a mechanical one.) With the invention of the IC, for which Jack Kilby also won the Nobel Prize, engineers gained the ability to create thousands, millions, or billions of tiny transistors by doping very small areas of a larger piece of silicon. To this day, this remains the primary way that computers are manufactured.

(*continued*)

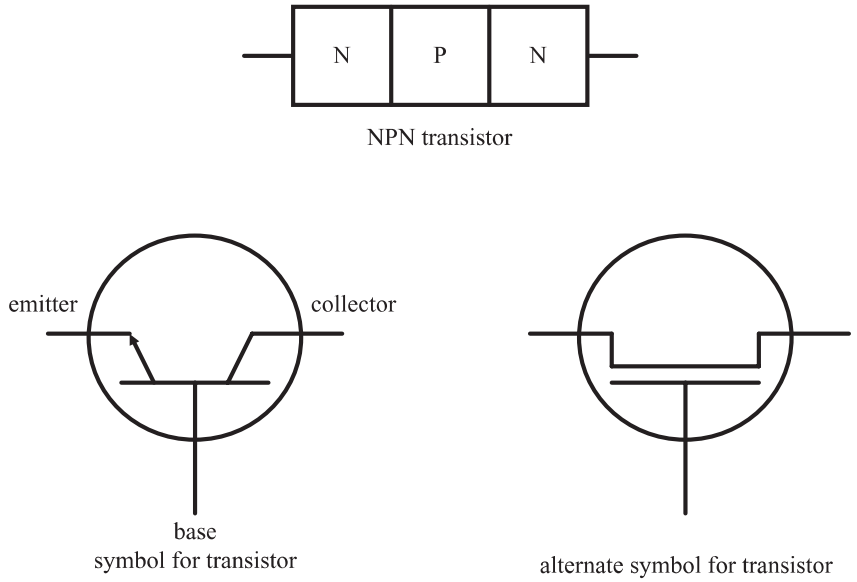


Figure 1.6 Transistors

1.2.2 Boolean Logic

Modern computers operate using a logic based on **bits**. A bit is the smallest unit that can be said to carry information, as in the children's game "Twenty Questions," where each yes or no question yields an answer that could be encoded with a single bit (for example, 1 represents a "yes" and 0 a "no"). If you are familiar with the old Milton Bradley board game, "Guess Who," the same principle applies. By asking yes/no questions, you gradually learn which person your opponent has chosen—and by recording the answers to your questions, you can recreate that learning at any point. A bit is therefore the smallest unit that can be operated upon—stored, transmited, or manipulated—using logic. The conventional way of performing logic on bit quantities is called **Boolean** logic, after the nineteenth-century mathematician George Boole. He identified three basic operations—AND, OR, and NOT—and defined their meaning in terms of simple changes on bits. For example, the expression X AND Y is true (a "yes," or a 1) if and only if, independently, X is a "yes" and Y is a "yes." The expression X OR Y, conversely, is a "yes" if either X is a "yes" or Y is a "yes." An equivalent way of stating this is that X OR Y is false (a "no," or a 0) if and only if X is a "no" and Y is a "no." The expression NOT X is the exact opposite of X: "yes" if X is a "no" and "no" if X is a "yes" (figure 1.7). Because a bit can be in only one of two states, there are no other possibilities to consider. These operations(AND, OR, and NOT) can be nested or combined as needed. For example, NOT (NOT X)) is the exact opposite of the exact opposite of X, which works out to be the same as X itself. These three operations parallel their English lexical equivalents fairly well: if I want a cup of coffee "with milk and sugar," logically what I am asking for is a cup of coffee where "with milk" is true AND "with sugar" is true. Similarly, a cup of coffee "without milk or sugar" is the same as a cup "with no milk and no sugar." (Think about it for a bit.)

In addition to these three basic operations, there are a number of other operations that can be defined from them. For example, NAND is an abbreviation for NOT-AND. The expression

AND	no	yes
no	no	no
yes	no	yes

OR	no	yes
no	no	yes
yes	yes	yes

NOT	
no	yes
yes	no

Figure 1.7 Truth tables for AND, OR, and NOT

X NAND Y refers to NOT (X AND Y). Similarly, X NOR Y refers to NOT (X OR Y). Another common expression is exclusive-OR, written XOR. The expression X OR Y is true if X is true, Y is true, or both. By contrast, X XOR Y is true if X is true or Y is true, but not both. This difference is not captured cleanly in English but is implicit in several different uses: for example, if I am asked if I want milk or sugar in my coffee, I'm allowed to say "yes, please," meaning that I want both. This is the normal (inclusive) OR. By contrast, if I am offered coffee or tea, it wouldn't make much sense for me to say "yes," meaning both. This is an exclusive XOR; I can have either coffee XOR tea, but not both at the same time.

From a strictly theoretical point of view, it doesn't matter much whether 1/"yes"/"true" is encoded as ground voltage or as 5 volts above ground as long as 0/"no"/"false" is encoded differently and the difference is consistently applied. From the point of view of a computer engineer or system designer, there may be particular reasons (such as power consumption) to choose one representation over another. The choice of representation can have profound implications for the design of the chips themselves. The Boolean operations described above are usually implemented in hardware at a very low level on the chip itself. For example, one can build a simple circuit with a pair of switches (or transistors) that will allow current to flow only if both switches are closed. Such a circuit is called an AND gate, because it implements the AND function on the 2 bits represented by the switch state. This tiny circuit and others like it (OR gates, NAND gates, and so forth), copied millions or billions of times across the computer chip, are the fundamental building blocks of a computer. (See Appendix A for more on how these blocks work.)

1.2.3 Bytes and Words

For convenience, 8 bits are usually grouped into a single block, conventionally called a **byte**. There are two main advantages to doing this. First, writing and reading a long sequence of 0s and 1s is, for humans, tedious and error prone. Second, most interesting computations require more data than a single bit. If multiple wires are available, as in standard buses, then electrical signals can be moved around in groups, resulting in faster computation.

The next-largest named block of bits is a **word**. The definition and size of a word are not absolute, but vary from computer to computer. A word is the size of the most convenient block of data for the computer to deal with. (Usually, though not always, it's the size of the bus leading to main memory—but see the Intel 8088, discussed later, for a counterexample.) For example, the Zilog Z-80 microprocessor (the chip underlying the Radio Shack TRS-80, popular in the mid-1970s) had a word size of 8 bits, or 1 byte. The CPU, memory storage, and buses had all been optimized to handle 8 bits at a time (for example, there were eight data wires in the system bus). In the event that the computer had to process 16 bits of data, it would be handled in two separate halves, while if the computer had only 4 bits of data to process, the CPU would work as though it had 8 bits of data, then throw away the extra 4 (useless) bits. The original IBM-PC, based on the

Intel 8088 chip, had a word size of 16 bits. More modern computers such as the Intel Pentium 4 or the PowerPC G4 have word sizes of 32 bits, and computers with word sizes of 64 bits or even larger, such as the Intel Itanium and AMD Opteron series, are available. Especially for high-end scientific computation or graphics, such as in home video game consoles, a large word size can be key to delivering data fast enough to allow smoothly animated, high-detail graphics.

Formally speaking, the word size of a machine is defined as the size (in bits) of the machine's **registers**. A register is the memory location inside the CPU where the actual computations, such as addition, subtraction, and comparisons, take place (figure 1.8). The number, type, and organization of registers vary widely from chip to chip and may even change significantly within chips of the same family. The Intel 8088, for example, had four 16-bit general-purpose registers, while the Intel 80386, designed seven years later, used 32-bit registers instead. Efficient use of registers is key to writing fast, well-optimized programs. Unfortunately, because of the differences between different computers, this can be one of the more difficult aspects of writing such programs for various computers.

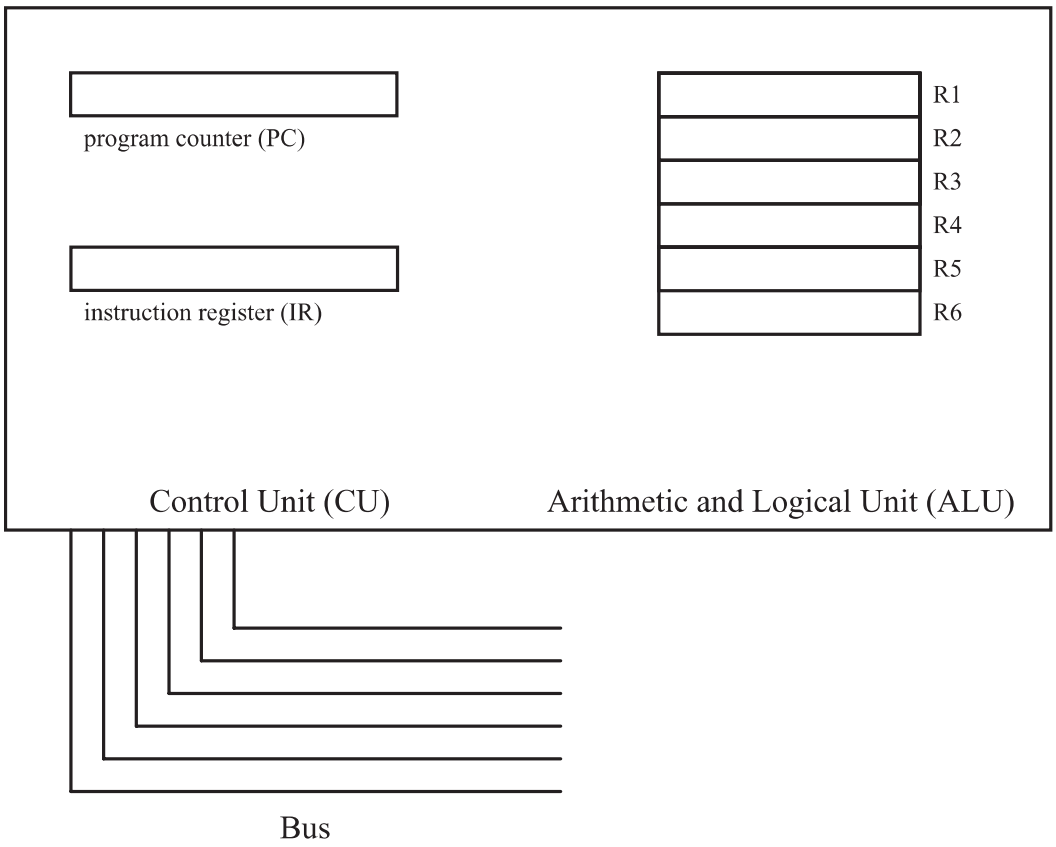


Figure 1.8 Typical internal structure of a CPU

1.2.4 Representations

Bit Patterns are Arbitrary

Consider, for a moment, one of the registers in an old-fashioned 8-bit microcomputer chip. How many different patterns can it hold? Other ways of asking the same question are to consider how

many different ways you can arrange a sequence of 10 pennies in terms of heads and tails or how many strings can be made up of only 8 letters, each of which is a 0 or a 1.

There are two possibilities for the first bit/coin/letter/digit, two for the second, and so on until we reach the eighth and final digit. There are thus $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ possibilities. This works out to 2^8 or 256 different storable patterns. Similar reasoning shows that there are 2^{32} or just over 4 billion storable patterns in a 32-bit register. (All right, for the pedants, $2^{32} = 4,294,967,296$. A handy rule of thumb for dealing with large binary powers is that 2^{10}, really 1024, is close to 1000. Remember that to multiply numbers, you add the exponents: $a^b \cdot a^c = a^{b+c}$. Thus, 2^{32} is $2^{2+10+10+10}$, or $2^2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10}$, or about $4 \cdot 1000 \cdot 1000 \cdot 1000$.)

Yes, but what do these patterns mean? The practical answer is: whatever you as the programmer want them to mean. As you'll see in the next subsection, it's fairly easy to read the bit pattern 00101101 as the number 77. It's also possible to read it as a record of the answers to eight different yes/no questions. ("Are you married?"—No. "Are you older than 25?"—No. "Are you male?"—Yes. And so forth.) It could also represent a key being pressed on the keyboard (in this case, the pattern represents the ASCII value for a capital M). The interpretation of bit patterns is arbitrary, and computers can typically use the same patterns in many ways. Part of the programmer's task is to make sure that the computer interprets these arbitrary and ambiguous patterns correctly at all times.

Natural Numbers

A common way to interpret bit patterns involves using **binary** arithmetic (in base 2). In conventional, or **decimal** (base 10) arithmetic, there are only ten different number symbols, 0 through 9. Larger numbers are expressed as individual digits times a power of the base value. The number four hundred eighty-one (481), for instance, is really $4 \cdot 10^2 + 8 \cdot 10^1 + 1$. Using this notation, we can express all natural numbers up to nine hundred ninety-nine in only three decimal digits. Using a similar notation, but adding up powers of 2, we can express any number in binary using only 0s and 1s.

Taking as an example the (decimal) number 85, simple arithmetic shows that it is equivalent to $64 + 16 + 4 + 1$, or (in more detail) to $1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1$. In binary, then, the number would be written as 1010101. In an 8-bit register, this could be stored as 01010101, while in a 32-bit register, it would be 00000000000000000000000001010101.

It's possible to do arithmetic on these binary numbers using the same strategies and algorithms that elementary students use for solving base 10 problems. In fact, it's even easier, as the addition and multiplication tables are much smaller and simpler! Because there are only two digits, 0 and 1, there are only four entries in the tables, as can be seen in figure 1.9. Just remember that, when adding in binary (base 2), every time the result is a 2, it generates a carry (just as every 10 generates a carry in base 10). So the result of adding $1 + 1$ in base 2 is not 2, but 0 carry the 1 or 10.

Inspection of the tables reveals the fundamental connection between binary arithmetic and Boolean algebra. The multiplication table is identical to the AND of the two factors. Addition, of course, can potentially generate two numbers: a one-digit sum and a possible carry. There is a carry if and only if the first number is a 1 and the second number is a 1; in other words, the carry is simply the AND of the two addends, while the sum (excluding the carry) is 1 if the first number is 1 or the second number is 1, but not both: the XOR of the two addends. By building an

+	0	1
0	0	1
1	1	10

·	0	1
0	0	0
1	0	1

Figure 1.9 Addition and multiplication tables for binary (base 2) arithmetic

appropriate collection of AND and XOR gates, the computer can add or multiply any numbers within the expressive power of the registers.

How large a number, then, can be stored in an 8-bit register? The smallest possible value is obviously 00000000, representing the number 0. The largest possible value, then, would be 111111111, the number 255. Any integer in this range can be represented easily as an 8-bit quantity. For 32-bit registers, the smallest value is still 0, but the largest value is just over 4.2 billion.

Although computers have no difficulty interpreting long binary numbers, humans often do. For example, is the 32-bit number 00010000000000000000100000000000 the same as the number (deep breath here) 00010000000000000001000000000000? (No, they are different. There are sixteen 0s between the 1s in the first number and only fifteen in the second.) For this reason, when it is necessary (rarely, one hopes) to deal with binary numbers, most programmers prefer to use **hexadecimal** (base 16) numbers instead. Since $16 = 2^4$, every block of 4 bits (sometimes called a **nybble**) can be represented as a single base 16 "digit." Ten of the 16 hexadecimal digits are familiar to us as the numbers 0 through 9, representing the patterns 0000 through 1001. Since our normal base 10 uses only 10 digits, computer scientists have co-opted the letters A through F to represent the remaining patterns (1010, 1011, 1100, 1101, 1110, 1111; see table 1.1 for the complete conversion list). Nowadays, this is about the only use that people have for hexadecimal numbers: to specify and abbreviate long lists of bitstring values such as might be used in cryptography or networking. The two numbers above are clearly different when converted to base 16:

0001	0000	0000	0000	0000	1000	0000	0000	
1	0	0	0	0	8	0	0	= 0x10000800
0001	0000	0000	0000	0001	0000	0000	0000	
1	0	0	0	1	0	0	0	= 0x10001000

Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Table 1.1 Hexadecimal ↔ binary digit conversions

By convention in many computer languages (including Java, C, and C++), hexadecimal numbers are written with an initial "0x" or "0X." We follow that convention here, so the number 1001 refers to the decimal value one thousand one. The value 0x1001 would refer to $16^3 + 1$, the decimal value four thousand ninety-seven. (Binary quantities will be clearly identified as such in the text. Also, on rare occasions, some patterns will be written as **octal**, or base 8, numbers. In C, C++, or Java, these numbers are written with a leading 0, so the number 01001 would be an octal value equivalent to 513. This is a rather unfortunate convention, since almost no one uses octal for any reason, but the tradition continues.) Note that 0 is still 0 (and 1 is still 1) in any base.

Base Conversions

Converting from a representation in one base to another can be a tedious but necessary task. Fortunately, the mathematics involved is fairly simple. Converting from any other base into base 10, for example, is simple if you understand the notation. The binary number 110110, for instance, is defined to represent $2^5 + 2^4 + 2^2 + 2^1$, $32 + 16 + 4 + 2$, or 54. Similarly, 0x481 is $4 \cdot 16^2 + 8 \cdot 16^1 + 1$, $1024 + 128 + 1$, or (decimal) 1153.

An easier way to perform the calculation involves alternating multiplication and addition. The binary number 110110 is, perhaps obviously, twice the binary value of 11011. (If this isn't obvious, notice that the base 10 number 5280 is 10 times the value of 528.) The binary number 11011 is, in turn, twice 1101 plus 1. Thus, one can simply alternate multiplying by the base value and adding the new digit. Using this system, binary 110110 becomes

$$((((((((((\mathbf{1} \cdot 2) + \mathbf{1}) \cdot 2) + \mathbf{0}) \cdot 2) + \mathbf{1}) \cdot 2) + \mathbf{1}) \cdot 2) + \mathbf{0})$$

which simple arithmetic will confirm is 54. Similarly 0x481 is

$$((((\mathbf{4} \cdot 16) + \mathbf{8}) \cdot 16) + \mathbf{1})$$

which can be shown to be 1153.

If alternating multiplication and addition will convert to base 10, then it stands to reason that alternating division and subtraction can be used to convert from base 10 to binary. Subtraction is actually implicit in the way we will be using division. When dividing integers by integers, it's rather rare that the answer is exact; normally there's a remainder that must be implicitly subtracted from the dividend. These remainders are exactly the base digits. Using 54 again as our example, the remainders generated when we repeatedly divide by 2 will generate the necessary binary digits.

$$54 \div 2 = 27r\mathbf{0}$$
$$27 \div 2 = 13r\mathbf{1}$$
$$13 \div 2 = 6r\mathbf{1}$$
$$6 \div 2 = 3r\mathbf{0}$$
$$3 \div 2 = 1r\mathbf{1}$$
$$1 \div 2 = 0r\mathbf{1}$$

The boldface numbers are, of course, the bits for the binary digits of 54 (binary 110110). The only tricky thing to remember is that, in the multiplication procedure, the digits are entered in the normal order (left to right), so in the division procedure, unsurprisingly, the digits come out in right-to-left order, backward. The same procedure works for base 16 (or indeed for base 8, base 4, or any other base):

$$1153 \div 16 = \mathbf{72}r\mathbf{1}$$
$$72 \div 16 = 4r\mathbf{8}$$
$$4 \div 16 = 0r\mathbf{4}$$

Finally, the most often used and perhaps the most important conversion is direct (and quick) conversion between base 2 and base 16, in either direction. Fortunately, this is also the easiest. Because 16 is the fourth power of 2, multiplying by 16 is really just multiplying by 2^4. Thus, every hexadecimal digit corresponds directly to a group of 4 binary digits. To convert from hexadecimal to binary, as discussed earlier, simply replace each digit with its 4-bit equivalent. To convert from binary to hexadecimal (hex), break the binary number into groups of 4 bits (starting at the right) and perform the replacement in the other direction. The complete conversion chart is (re)presented as table 1.2.

Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Table 1.2 Hexadecimal ↔ binary digit conversions (copy of table 1.1)

So, for example, the binary number 100101101100101 would be broken up into 4-bit nybbles, starting from the right, as 100 1011 0110 0101. (Please note that I cheated: the number I gave you has only 15 bits, so one group of 4 isn't complete. This group will always be on the far left and will be padded out with 0s, so the "real" value you will need to convert will be 0100 1011 0110 0101.) If you look these four values up in the table, you will find that they correspond to the values 4, B, 6, and 5. Therefore, the corresponding hexadecimal number is 0x4B65.

Going the other way, the hexadecimal number 0x18C3 would be converted to the four binary groups 0001 (1), 0100 (8), 1100 (C) and 0101 (3), which are put together to give the binary quantity 0001010011000101.

A similar technique would work for octal (base 8) with only the first two columns of table 1.1 and using only the last 3 (instead of 4) bits of the binary entries, as shown in table 1.3. Using these notations and techniques, you can represent any nonnegative integer in a sufficiently large register and interpret it in any base we've discussed. With a bit of creativity and flair, you can even adapt these techniques to oddball bases like base 3 or base 7.

Hex	Binary	Hex	Binary
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Table 1.3 Octal ↔ binary digit conversions

Signed Representations

In the real world, there is often a use for negative numbers. If the smallest possible value stored in a register is 0, how can a computer store negative values? The question, oddly enough, is not one of storage but of interpretation. Although the maximum number of storable patterns is fixed (for a given register size), the programmer can decide to interpret some patterns as meaning negative values. The usual method for doing this is to use an interpretation known as **two's complement notation**.

It's a common belief (you can even see it in the film *Ferris Beuller's Day Off*) that if you drive a car in reverse, the odometer will run backward and apparently take miles off the engine. I don't know if it works or not—it didn't work for Ferris—but Howstuffworks.com[1] says that it should have. Imagine for a moment that it did. Suppose I took a relatively new car (say, with 10 miles on it) and ran it in reverse for 11 miles. What would the odometer say?

Well, the odometer *wouldn't* say −1 miles. It would probably read 999,999 miles, having turned over at 000,000. But if I then drove another mile forward, the odometer would turn over (again) to 000,000. We can implicitly define the number −1 as the number that, when 1 is added to it, results in a 0.

This is how two's complement notation works; negative numbers are created and manipulated by counting backward (in binary) from a register full of 0s, as in figure 1.10. Numbers in which the first bit is a 0 are interpreted as positive numbers (or 0), while numbers in which the first bit is a 1 are interpreted as negative. For example, the number 13 would be written in (8-bit) binary as 00001101 (hexadecimal 0x0D), while the number −13 would be 11110011 (0xF3). These patterns are called **signed** numbers (integers, technically) as opposed to the previously defined **unsigned** numbers. In particular, the pattern 0xF3 is the **two's complement notation** representation of −13 (in an 8-bit register).

How do we get from 0xF3 to −13? Beyond the leading 1, there appears to be no similarity between the two representations. The connection is a rather subtle one based on the above definition of negative numbers as the inverses of positive numbers. In particular, 13 + −13 should equal 0. Using the binary representations above, we note that

```
    00001101                    13
+   11110011                   -13 in two's-complement notation
------------
   100000000                    0 plus an overflow/carry
```

[1] http://auto.howstuffworks.com/odometer1.htm

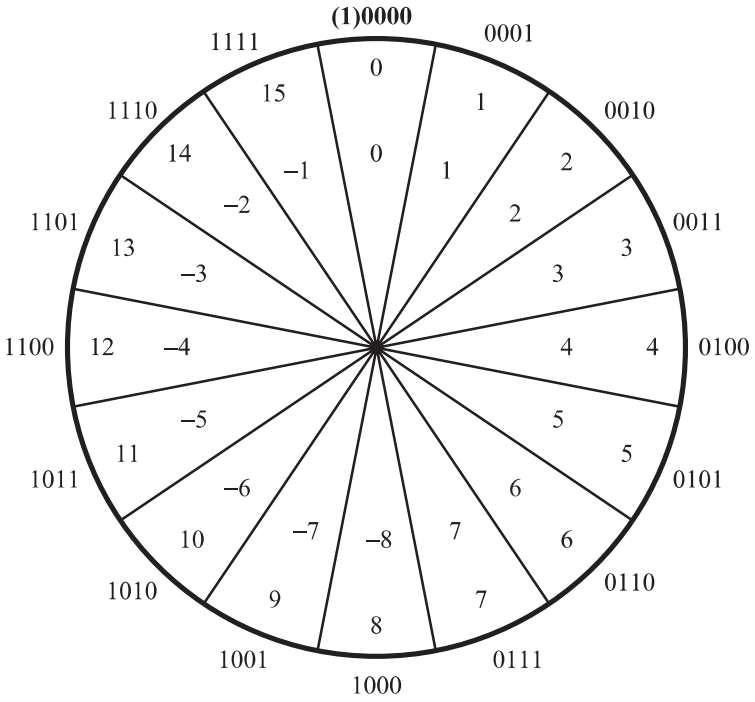


Figure 1.10 Structure of 4-bit signed integer representation

However, the 9-bit quantity 100000000 (0x100) cannot be stored in only an 8-bit register! Like a car odometer that rolls over when the mileage becomes too great, an 8-bit register will **overflow** and lose the information contained in the ninth bit. The resulting stored pattern is therefore 00000000 or 0x00, which is the binary (and hex equivalent of 0. Using this method, we can see that the range of values stored in an 8-bit register will vary from −128 (0x80) to +127 (0x7F). Approximately half of the values are positive and half are negative, which in practical terms is about what people typically want. Figure 1.10 shows that, in general, there is one more negative number than positive number, because the number opposite 0 on the figure is negative.

This demonstration relies critically on the use of an 8-bit register. In a 32-bit register, a much larger value is necessary to produce overflow and wrap around to 0. The 32-bit two's complement representation of −13 would not be 0xF3, but 0xFFFFFFF3. In fact, viewed as a 32-bit number, 0xF3 would normally be interpreted as 0x000000F3, which isn't even negative, since the first bit isn't a 1.

Calculating the two's complement representation of a number (for a given fixed register size) by hand is not difficult. Notice first that the representation of −1, for any register size, will always be a register containing all 1s. Adding 1 to this number will produce overflow and a register full of 0s. For any given bit pattern, if you reverse every individual bit (each 1 becomes a 0, each 0 becomes a 1, while preserving the original order—this operation is sometimes called the **bitwise** NOT, because it applies a NOT to every individual bit) and add the resulting number to the original, the result will always give you a register full of 1s. (Why?) This inverted pattern (sometimes called the "one's complement" or just the "complement"), added to the original pattern, will yield a sum

of −1. And, of course, adding one more will give you a −1 + 1, or 0. This inverted pattern plus 1, then, will give you the two's complement of the original number.

```
    00001101   (= 13)
    11110010   (inverted)
+          1
------------
    11110011   (= -13)
```

Note that repeating the process a second time will negate the negation.

```
    11110011   (= -13)
    00001100   (inverted)
+          1
------------
    00001101   (= 13)
```

This process will generalize to any numbers and any (positive) register size. Subtraction will happen in exactly the same way, since subtracting a number is exactly the same as adding its negative.

Floating Point Representation

In addition to representing signed integers, computers are often called upon to represent fractions or quantities with decimal points. To do this, they use a modification of standard scientific notation based on powers of 2 instead of powers of 10. These numbers are often called **floating point** numbers because they contain a "decimal" point that can float around, depending upon the representation.

Starting with the basics, it's readily apparent that any integer can be converted into a number with a decimal point just by adding a decimal point and a lot of 0s. For example, the integer 5 is also 5.000..., the number −22 is also −22.0000..., and so forth. This is also true for numbers in other bases (except that technically the "decimal" point refers to base 10; in other bases, it would be called a "radix" point). So the (binary) number 1010 is also 1010.0000..., while the (hexadecimal) number 0x357 is also 0x357.0000....

Any radix point number can also be written in scientific notation by shifting the point around and multiplying by the base a certain number of times. For example, Avogadro's number is usually approximated as $6.023 \cdot 10^{23}$. Even students who have forgotten its significance in chemistry should be able to interpret the notation—Avogadro's number is a 24-digit (23 + 1) number whose first four digits are 6, 0, 2, 3, or about 602,300,000,000,000,000,000,000. Scientific notation as used here has three parts: the **base** (in this case, 10), the **exponent** (23), and the **mantissa** (6.023). To interpret the number, one raises the base to the power of the exponent and multiplies by the mantissa. Perhaps obviously, there are many different mantissa/exponent sets that would produce the same number; Avogadro's number could also be written as $6023 \cdot 10^{20}$, $60.23 \cdot 10^{22}$, or even $0.6023 \cdot 10^{24}$.

Computers can use the same idea, but using binary representations. In particular, note the patterns in table 1.4. Multiplying the decimal number by 2 shifts the representation 1 bit to the

left, while dividing by 2 shifts it 1 bit to the right. Alternatively, a form of scientific notation applies where the same bit pattern can be shifted (and the exponent adjusted appropriately), as in table 1.4.

Decimal	Binary integer	Binary radix	Scientific notation
5	00000101	00000101.0000...	1.01 (binary) · 2^2
10	00001010	00001010.0000...	1.01 (binary) · 2^3
20	00010100	00010100.0000...	1.01 (binary) · 2^4
40	00101000	00101000.0000...	1.01 (binary) · 2^5

Table 1.4 Exponential notation with binary numbers

We extend this to represent noninteger floating point numbers in binary the usual way, as expressed in table 1.5. The number 2.5, for example, being exactly half of 5, could be represented as the binary quantity 10.1 or as the binary quantity 1.01 times 2^1. 1.25 would be (binary) 1.01 or 1.01 times 2^0. Using this sort of notation, any decimal floating point quantity has an equivalent binary representation.

Decimal	Binary	Binary real	Scientific notation
5	101	00101.000...	1.0100 (binary) · 2^2
2.5		00010.100...	1.0100 (binary) · 2^1
1.25		00001.010...	1.0100 (binary) · 2^0
0.6125		00000.101...	1.0100 (binary) · 2^{-1}

Table 1.5 Exponential notation with binary fractions

The Institute for Electrical and Electronic Engineers (IEEE) has issued a series of specification documents describing standardized ways to represent floating point numbers in binary. One such standard, IEEE 754-1985, describes a method of storing floating point numbers as 32-bit words as follows (and as illustrated in figure 1.11:

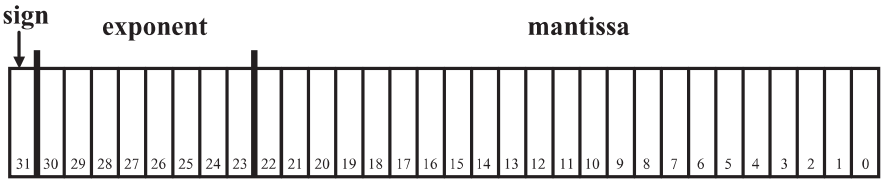


Figure 1.11 IEEE 32-bit floating point storage: sign, exponent, mantissa

Number the 32 bits of the word starting from bit 31 at the left down to bit 0 at the right. The first bit (bit 31) is the **sign** bit, which (as before) tells whether the number is positive or negative. Unlike two's complement notation, this sign bit is the only way in which two numbers of equal magnitude differ.

The next 8 bits (bits 30–23) are a "biased" **exponent**. It would make sense to use the bit pattern 0000000 to represent 2^0 and 0000001 to represent 2^1, but then one couldn't represent small values like 0.125 (2^{-3}). It would also make sense to use 8-bit two's complement notation,

but that's not what the IEEE chose. Instead, the IEEE specified the use of the unsigned numbers 0..255, but the number stored in the exponent bits (the representational exponent) is a "biased" exponent, actually 127 higher than the "real" exponent. In other words, a real exponent of 0 will be stored as a representational exponent of 127 (binary 01111111). A real exponent of 1 would be stored as 128 (binary 10000000), and a stored exponent of 00000000 would actually represent the tiny quantity 2^{-127}. The remaining 23 bits (bits 22–0) are the mantissa with the decimal point—technically called the **radix point**, since we're no longer dealing with base 10—placed conventionally immediately after the first binary digit. This format, using a fixed number of positions in front of the radix point, is sometimes called **normalized form**.

Thus, for normal numbers, the value stored in the register is the value

$$(-1)^{sign\ bit} \cdot mantissa \cdot 2^{realexponent + 127}$$

A representational exponent of 127, then, means that the mantissa is multiplied by 1 (a corresponding real exponent of 0, hence a multiplier of 2^0), while an exponent of 126 means that the fraction is multiplied by 2^{-1} or 0.5, and so forth.

Actually, there is a micro-lie in the above formula. Because the numbers are in binary, the first nonzero digit has to be a 1 (there aren't any other choices that aren't 0!). Since we know that the first digit is a 1, we can leave it out and use the space freed up to store another digit whose value we didn't know beforehand. So the real formula would be

$$(-1)^{sign\ bit} \cdot 1.mantissa \cdot\ 2^{realexponent + 127}$$

As a simple example, the decimal number 2.0 in binary is $1.0 \cdot 2^1$. Representing this as an IEEE floating point number, the sign bit would be 0 (a positive number), the mantissa would be all 0s (and an implicit leading 1), while the exponent would be 127 + 1, or 128, or binary 10000000. This would be stored in the 32-bit quantity

```
 0     10000000    00000000000000000000000
sign   exponent           mantissa
```

This same bit pattern could be written as the hexadecimal number 0x40000000.

The number −1.0, on the other hand, would have a sign bit of 1, an exponent of 127 + 0, or binary 01111111, and a mantissa of all 0s (plus an implicit leading 1). This would yield the following 32-bit quantity :

```
 1     01111111    00000000000000000000000
sign   exponent           mantissa
```

Another way of writing this bit pattern would be 0xBF800000.

There's a little problem with this. What do you do if the number is to be stored exactly (0.000...)? There is no "implicit leading 1" anywhere in a string of all 0s. The IEEE defined as a special case that a bit pattern of all 0s (sign, exponent, and mantissa) would represent the number 0.0. There are also a number of other special cases, including representations of both positive and negative infinity, and the so-called NaN (not a number, the number that results when you

try an illegal operation such as taking the logarithm of a negative number) and "denormalized numbers," which are used, rarely, to represent extremely tiny numbers with greater accuracy. The IEEE has also defined standard methods for storing numbers more accurately in 64-bit and larger registers. These additional cases are similar in spirit, if not in detail, to the 32-bit standard described above, but the details are rather dry and technical and not necessarily of interest to the average programmer.

One problem that comes up in any kind of radix-based representation is that of numbers that are not represented exactly. Even without worrying about irrational numbers (like π), some simple fractions can't be represented exactly. In base 10, for example, the fraction 1/7 has the approximate value 0.14285714285714... but never comes to an end. The fraction 1/3 is similarly 0.33333.... In base 2, the fraction 1/3 is 0.010101010101010101.... But there's no way to fit an infinite sequence into only 23 mantissa bits. So the solution is: we don't.

Instead, the number is represented as closely as possible. Converting to radix point, we see that 1/3 is about equal to $1.01010101010101010101010 \cdot 2^{-2}$, represented as

```
 0   01111101  01010101010101010101010
sign exponent         mantissa
```

This isn't a perfect representation, but it's close. But "close" may not be good enough in all contexts; small errors (called **roundoff error**) will inevitably creep into calculations involving floating point numbers. If I multiplied this number by the value 3.0 ($1.1 \cdot 2^1$), I'd be very unlikely to get the exact value 1.0. Programmers, especially those who handle big numerical problems like matrix inversion, spend a lot of time trying to minimize this sort of error. For now, all we can do is to be aware of it.

Performing operations on floating point numbers is tricky (and often slow). Intuitively, the algorithms are understandable and very similar to those you already know for manipulating numbers in scientific notation. Multiplication is relatively easy, since we know that 2^x times 2^y is just 2^{x+y}. Therefore, the product of two floating point numbers has as its sign bit the product or XOR of the two signs, as its mantissa the product of the two mantissae, and as its exponent the sum of the two exponents.[2] Thus we have the following:

3.0 = (0) (10000000) (10000000000000000000000) [0x40400000]

2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]

The sign bit of the result of $3.0 \cdot 2.5$ would be 0. The mantissa would be $1.100... \cdot 1.010...$, or 1.111... (do you see why?). Finally, the exponent would be $1 + 1$, or 2, represented as 1000001. Thus we see that the product would be

(0) (10000001) (11100000000000000000000) [0x40F00000]

as expected, which converts to 7.5.

[2] But remember to account for both the exponent bias and the unexpressed 1 bit at the head of the mantissa!

Addition is considerably more difficult, as addition can happen only when the exponents of the two quantites are the same. If the two exponent values are the same, then adding the mantissae is (almost) enough:

3.0 = (0) (10000000) (10000000000000000000000) [0x40400000]

2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]

The binary quantities 1.01 and 1.1 add up to 10.11, so the answer would be 10.11 times the common exponent: $10.11 \cdot 2^1$. Of course, this isn't legal, but it's easy enough to convert by shifting to a legal equivalent: $1.011 \cdot 2^2$. This yields

(0) (10000001) (01100000000000000000000) [0x40B00000]

which converts, as expected, to 5.5.

However, when the two exponents are not the same (for example, when adding 2.5 and 7.5), one of the addends must be converted to an equivalent form, but with a compatible exponent.

2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]

7.5 = (0) (10000001) (11100000000000000000000) [0x40F00000]

Let's convert 7.5: $1.111 \cdot 2^2$ is the same as $11.11 \cdot 2^1$. Adding 11.11 + 1.01 yields 101.00; $101.00 \cdot 2^1$ is the same as $1.01 \cdot 2^3$. The final answer is thus

(0) (10000010) (01000000000000000000000) [0x41200000]

which, again, is the expected value, in this case 10.0.

Character Representations

Nonnumeric data such as characters and strings are also treated as binary quantities and differ only in interpretation by the programmer/user. The most common standard for storing characters is the **ASCII** code, formally the American Standard Code for Information Interchange. ASCII code assigns a particular character value to every number between 0 and 127.

This is a slight oversimplification. Technically speaking, ASCII provides an interpretation for every 7-bit binary pattern between (and including) 0000000 and 1111111. Many of these patterns are interpreted as characters; for example, the pattern 100001 is an uppercase 'A.' Some binary strings, especially those between 0000000 and 0011111, are interpreted as "control characters," such as a carriage return, or a command to a peripheral such as "start of header" or "end of transmission." As almost all computers are byte-oriented, most store ASCII characters not as 7-bit patterns but as 8-bit patterns, with the leading bit being a 0. The letter A would be, for example, (binary) 01000001, or (hexadecimal) 0x41, or (decimal) 65. Using 8-bit storage allows computers to use the additional (high-bit) patterns for character set extensions; in Microsoft Windows, for example, almost every character set has different display values in the range 128–255. These display values may include graphics characters, suits (of cards), foreign characters with diacritical marks, and so forth.

The chief advantage of the ASCII encoding is that every character fits comfortably into a single byte. The chief disadvantage of ASCII is that, as the *American* standard code, it does not well reflect the variety of alphabets and letters in use worldwide. As the Internet continued to connect people of different nationalities and languages together, it became obvious that some method of encoding non-English (or at least, non-U.S.) characters was necessary. The result was **UTF-16** encoding, promulgated by the **Unicode** consortium.

UTF-16 uses 2 bytes (16 bits) to store each character. The first 128 patterns are almost identical to the ASCII code. With 16 bits available, however, there are over 65,000 (technically, 65,536) different patterns, each of which can be assigned a separate (printable) character. This huge set of characters allows uniform and portable treatment of documents written in a variety of alphabets, including (U.S.) English, unusual characters such as ligatures (e.g., æ) and currency symbols, variations on the Latin alphabet such as French and German, and unusual (from the point of view of American computer scientists) alphabets such as Greek, Hebrew, Cyrillic (used for Russian), Thai, Cherokee, and Tibetan. The Greek capital psi (Ψ), for example, is represented by 0x803A, or in binary by 1000000000111010. Even the Chinese/Japanese/Korean ideograph set (containing over 40,000 characters) can be represented (see figure 1.12 for an example).

Machine Operation Representations

In addition to storing data of many different types, computers need to store executable program code. As with all other patterns we have discussed, at its most basic level the computer can only interpret binary activation patterns. These patterns are usually called **machine language** instructions. One of the major roles of a register in the CPU is to fetch and hold an individual bit pattern where that pattern can be decoded into a machine instruction and then executed as that instruction.

Interpretation of machine language is difficult in general and varies greatly from computer to computer. For any given computer, the **instruction set** defines which operations the computer can execute. The Java Virtual Machine (JVM), for example, has a relatively small instruction set, with only 256 possible operations. Every byte, then, can be interpreted as a possible action to take. The value 89 (0x59) corresponds to the `dup` instruction, causing the machine to duplicate a particular piece of information stored in the CPU. The value 146 (0x92) corresponds to the `i2c` instruction, which converts a 32-bit quantity (usually an integer) to a 16-bit quantity (usually a Unicode character). These number-to-instruction correspondences are specific to the JVM and would not work on a Pentium 4 or a PowerPC, which have their own idiosyncratic instruction sets.

The task of generating machine code to do a particular task is often extremely demanding. Computers usually provide a large degree of programmed support for this task. Programmers usually write their programs in some form of human-readable language. This language is then converted into machine code by a program such as a compiler, in essence a program that converts human-readable descriptions of programs into machine code.

Interpretation

In light of what we have seen in the preceding sections, it is clear that any given bit pattern can almost certainly be interpreted in several different ways. A given 32-bit pattern might be a floating

point number, two UTF-16 characters, a few machine instructions, two 16-bit signed integers, an unsigned 32-bit integer, or many other possibilities (see figure 1.12). How does the computer distinguish between two otherwise identical bitstrings?

Interpreted as...	0011 1110	0010 0000	0111 0010	0011 0111
Hexadecimal numbers	3E	20	72	37
Signed 32-bit integer	1042313783			
Signed 16-bit integers	15904		29239	
Signed 32-bit float	0.156686			
8-bit characters	>	space	r	7
16-bit characters	㫾		爷	
JVM machinei nstructions	istore_3	lstore_2	frem	lstore

Figure 1.12 Bit pattern with multiple interpretations

The short and misleading answer is that it can't. A longer and more useful answer is that the distinction between the bitstrings is provided by the context of the program instructions. As will be discussed in the following chapter, most computers (including the primary machine discussed, the JVM) have several different kinds of instructions that do, broadly speaking, the same thing. The JVM, for example, has separate instructions to add 32-bit integers, 64-bit integers, 32-bit floating point numbers, and 64-bit floating point numbers. Implicit in these instructions is that the bit patterns to be added will be treated as though they were the appropriate type. If your program loads two integers into registers and then tells the computer to add two floating point numbers, the computer will naively and innocently treat the (integer) bit patterns as though they were floating point numbers, add them, and get a meaningless and error-ridden result. (Figure 1.12 illustrates this, since the instructions implicit in that pattern are type-illegal.) Similarly, if you tell the computer to execute a floating point number as though it were machine code, the computer will attempt to carry out, to the best of its ability, whatever silly instruction(s) that number corresponds to. If you're lucky, this will merely cause your program to crash. If you're not lucky, . . . well, that's one of the major ways that hackers can break into a computer: by overflowing a buffer and overwriting executable code with their own instructions.

It's almost always an error to try to use something as though it were a different data type. Unfortunately, this is an error that the computer can only partially compensate for. Ultimately, it is the responsibility of the programmer (and the compiler writer) to make sure that data is correctly stored and that bit patterns are correctly interpreted. One of the major advantages of the JVM is that it can catch some of these errors.

1.3 Virtual Machines

1.3.1 What is a "Virtual Machine"?

Because of differences between instruction sets, the chances are very good (warning: that's quite an understatement!) that a program written for one particular platform (CPU type and/or

operating system) will not run on a different one. This is why software vendors sell different versions of programs for Windows computers using Pentium 4 CPUs and Macintosh computers using PowerPC CPUs, and also why many programs have "required configurations," stating that a particular computer must have certain amounts of memory or certain specific video cards to work properly. In the extreme case, this would require that computer programmers write related programs (such as the Mac and Windows versions of the same game) independently from scratch, a process that would essentially double the time, effort, and cost of such programs. Fortunately, this is rarely necessary. Most programming is done in so-called **high-level languages** such as C, C++, or Java, and then the (human-readable) program source code is converted to executable machine code by another program such as a compiler. Only small parts of programs—for example, embedded systems, graphics requiring direct hardware access, or device drivers controlling unusual peripherals—need be written in a machine-specific language.

The designers of Java, recognizing the popularity of the Web and the need for program-enabled Web pages to run anywhere, have taken a different approach. Java itself is a high-level language. Java programs are typically compiled into **class files**, with each file corresponding to the machine language for a program or part of a program. Unlike normal executables compiled from C, Pascal, or C++, the class files do not necessarily correspond to the physical computer on which the program is written or running. Instead, the class file is written using the machine language and instruction set of the **Java Virtual Machine** (JVM), a machine that exists only as a software emulation, a computer program pretending to be a chip. This "machine" has structure and computing power typical of—in some cases, even greater than–a normal physical computer such as an Intel Pentium 4, but is freed from many of the problems and limitations of a physical chip.

The JVM is usually a program running on the host machine. Like any other executable program, it runs on a physical chip using the instruction set of the local machine. This program, though, has a special purpose: its primary function is to interpret and execute class files written in the machine language of the JVM. By running a specific program, then, the physical chip installed in the computer can pretend to be a JVM chip, thereby being able to run programs written using the JVM instruction set and machine code.

The idea of a "virtual machine" is not new. In 1964, IBM began work on what would be known as VM/CMS, an operating system for the System/360 that provided time-sharing service to a number of users simultaneously. In order to provide the full services of the computer to every user, the IBM engineers decided to build a software system and user interface that gave the user the impression that he or she was alone and had the entire system to himself or herself. Every person or program could have an entire virtual S/360 at his or her disposal as needed, without worrying about whether this program would cause another person's program to crash. This also allowed engineers to upgrade and improve the hardware significantly without forcing users to relearn or rewrite programs. More than twenty years later, VM/CMS was still in use on large IBM mainframes, running programs designed and written for a virtual S/360 on hardware almost a million times faster. Since then, virtual machines and emulators have become a standard part of many programming tools and languages (including Smalltalk, an early example of an object-oriented language).

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 29
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

SIDEBAR

The .Net Framework

Another example of a common virtual machine is the .NET Framework, developed by Microsoft and released in 2002. The .NET Framework underlies the current version of Visual Studio and provides a unified programming model for many network-based technologies such as ASP.NET, ADO.NET, SOAP, and .NET Enterprise Servers. Following Sun's example of the JVM, .NET incorporates a **Common Language Runtime** (CLR), a virtual machine to manage and execute code developed in any of several languages. The underlying machine uses a virtual instruction set called **Microsoft Intermediate Language** (MSIL) that is very similar in spirit to JVM bytecode. Even the detailed architecture of the CLR's execution engine is similar to that of the JVM; for example, they're both stack-based architectures with instruction set support for object-oriented, class-based environments. Like the JVM, the CLR was designed to have most of the advantages of a virtual machine: abstract portable code that can be widely distributed and run without compromising local machine security. Microsoft has also followed Sun's example in the development and distribution of a huge library of predefined classes to support programmers using the .NET Framework who don't want to reinvent wheels. Both systems were designed with Web services and mobile applications in mind. Unlike the JVM, MSIL was designed more or less from the start to support a wide variety of programming languages, including J#, C#, Managed C++, and Visual Basic. Microsoft has also established a standard assembler **llasm**. In practical terms, the CLR is not as common or as widely distributed a computing environment as the JVM, but the software market is extremely dynamic and its final verdict has not yet been returned.

One key issue that will probably have a significant impact is the degree of multiplatform support that Microsoft provides. In theory, MSIL is as platform-independent as the JVM, but substantial parts of the .NET Framework libraries are based on earlier Microsoft technologies and will not run successfully on UNIX systems or even on older Windows systems (such as Windows 98). Microsoft's track record of supporting non-Microsoft (or even older Microsoft) operating systems does not encourage third-party developers to develop cross-platform MSIL software. Java, by comparison, has developed a strong group of developers on all platforms who both rely on and support this portability. If Microsoft can follow through on its promise of multiplatform support and develop a sufficient customer base, .NET might be able to replace Java as the system of choice for developing and deploying portable Web-based applications.

1.3.2 Portability Concerns

A primary advantage of a virtual machine, and the JVM in particular, then, is that it will run anywhere that a suitable interpreter is available. Unlike a Mac program that requires a PowerPC G4 chip to run (G4 emulators exist, but they can be hard to find and expensive), the JVM is widely available for almost all computers and for many kinds of equipment such as PDAs. Every major Web browser such as Internet Explorer, Netscape, or Konqueror has a JVM (sometimes called a "Java runtime system" built in to allow Java programs to run properly).

Furthermore, the JVM will probably continue to run anywhere, as the program itself is relatively unaffected by changes in the underlying hardware. A Java program (or JVM class file) should behave in the same way, except for speed, on a JVM emulator written for an old Pentium computer as for a top-end Power G7—a machine so new that it doesn't exist yet. However, if/when Motorola makes one, someone will almost certainly write a JVM client for it.

1.3.3 Transcending Limitations

Another advantage that virtual machines (and the JVM in particular) can provide is the ability to transcend, ignore, or mask limitations imposed by the underlying hardware. The JVM has imaginary parts corresponding to the real computer components discussed above, but because they consist only of software, there's little or no cost to making them. As a result, they can be as large or numerous as the programmer needs. Every (physical) register on a chip takes up space, consumes power, and costs significant amounts of money; as a result, registers are often in somewhat short supply. On the JVM registers are essentially free, and programmers can have and use as many as they like. The system bus, connecting the CPU to memory, can be as large as a programmer wants or needs.

A historical example may make the significance of this point clear. The original IBM-PC was based on the Intel 8088 chip. The 8088 was, in turn, based on another (earlier) chip by Intel, the 8086, almost identical in design but with a 16-bit bus instead of an 8-bit bus. This implicitly limits the data transfer speed between the CPU and memory (of an 8088) by about 50% relative to an 8086, but IBM chose the 8088 to keep its manufacturing costs and sales prices down. Unfortunately, this decision limited PC performance for the next 15 years or so, as IBM, Intel, and Microsoft were required to maintain backward compatibility with every succeeding generation of chips in the so-called Intel 80x86 family. Only with the development of Windows was Microsoft finally able to take full advantage of cheaper manufacturing and wider buses. Similar problems still occur, with most major software and hardware manufacturers struggling to support several different chips and chipsets in their products, and a feature available on a high-end chipset may not be available at the low end. Appropriately written Java runtime systems can take advantage of the feature where available (on a JVM written specifically for the new architecture) and make it useful to all Java programs or JVM class files.

The JVM also has the advantage of having, fundamentally, a better and cleaner design than most physical computer chips. This is due in part to its being designed from scratch in 1995 instead of inheriting several generations of engineering compromises from earlier versions. But the simplicity that results from not having to worry about physical limitations such as chip size, power consumption, or cost meant that the designers were able to focus their attention on producing a mathematically tractable and elegant design that allows the addition of useful high-level properties. In particular, the JVM design allows for a high degree of security enhancements, something discussed briefly below and in greater detail in chapter 10.

1.3.4 Ease of Updates

Another advantage of a virtual machine is the ease of updating or changing it compared to the difficulty of upgrading hardware. A well-documented error with the release of the Pentium chip in 1994 showed that the FPU in the Pentium P54C didn't work properly. Unfortunately, fixing this flaw required consumers to send the old chips back to Intel and to receive/install a complete physical replacement. By contrast, a bug in a JVM implementation can be repaired in software by the writers, or even possibly by a third party with source-code access, and distributed via normal channels, including simply making it available on the Internet.

Similarly, a new and improved version of the JVM, perhaps with updated algorithms or a significant speed increase, can be distributed as easily as any other updated program, such as a new video card driver or a security upgrade to a standard program. Since the JVM is software

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 31
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

only, a particularly paranoid user can even keep several successive versions so that in case a new version has some subtle and undiscovered bug, she can revert to the old version and still run programs. (Of course, you only think this user is paranoid until you find out she's right. Then you realize she's just careful and responsible.)

1.3.5 Security Concerns

A final advantage of virtual machines is that, with the cooperation of the underlying hardware, they can be configured to run in a more secure environment. The Java language and the JVM were designed specifically with this sort of security enhancement in mind. For instance, most Java applets don't require access to the host computer's hard drive, and providing them with such access, especially with the ability to write on the hard drive, might leave the computer open to infection by a computer virus, theft of data, or simply deletion or corruption of crucial system files. The virtual machine, being software only, is in a position to vet attempted disk accesses and to enforce a more sophisticated security policy than the operating system itself may be willing or able to enforce.

The JVM goes even further, being designed not only for security but for a certain degree of verifiable security. Many security flaws in programs are created accidentally—for example, by a programmer attempting to read in a string without making sure that enough space has been allocated to hold it, or even attempting to perform an illegal operation (perhaps dividing a number by an ASCII string), with unpredictable and perhaps harmful results. JVM bytecode is designed to be verifiable and verified via a computer program that checks for this sort of accidental error. Not only does this reduce the possibility of a harmful security flaw, but it also improves the overall quality and reliability of the software. Software errors, after all, don't necessarily crash the system or allow unauthorized access. Many errors will, instead, quietly and happily—and almost undetectably—produce wrong answers. Catching these errors before the answers are produced, sometimes even before the program is run, substantially reduces this source of wrong answers and significantly increases the reliability of programs. The JVM security policy will be discussed extensively in chapter 10.

1.3.6 Disadvantages

So, if virtual machines are so wonderful, why aren't they more common? The primary answer, in one word, is "speed." It usually takes about 1000 times longer to do a particular operation in software instead of hardware, so performing hard-core computations (matrix multiplications, DNA sequencing, and so forth) in Java on a JVM may be significantly slower than performing the same computations in (chip-specific) machine language compiled from C++.

The practical gap, fortunately, does not appear to be anywhere near 1000 times, mainly because there have been some very smart JVM implementers. A properly written JVM will take advantage of the available hardware where practical and will do as many operations as possible using the available hardware. Thus, the program will (for example) use the native machine's circuitry for addition instead of emulating it entirely in software. Improvements in compiler technology have made Java run almost as fast as natively compiled code, often within a factor of 2, and sometimes almost imperceptibly slower. A February 1998 study by *JavaWorld* ("Performance tests show Java as fast as C++") found that, for a set of benchmark tasks, high-performance JVMs with high-efficiency compilers typically produced programs that ran, at worst, only slightly (about 5.6%) more slowly than their C++ equivalents, and for the most part ran identically to the limits of

measurement. Of course, comparing computers and languages for speed is a notoriously difficult process, much like the standard apples and oranges comparison, and other researchers have found other values.

In some regards, speed comparisons can be a nonissue; for most purposes other than hard-core number crunching, Java or any other reasonable language is fast enough for most people's purposes. Java, and by extension the JVM, provide a powerful set of computation primitives including extensive security measures that are not found in many other languages, such as C++. It's not clear that the ability to crash a computer quickly is a tremendous advantage over being able to run a program to completion but somewhat more slowly.

A more significant disadvantage is the interposition of the JVM interpreter between the programmer and the actual physical hardware. For many applications demanding assembly language programming (games, high-speed network interfacing, or attaching new peripherals), the key reason that assembly language is used is to allow direct control of the hardware (especially of peripherals) at a very low level, often bit by bit and wire by wire. A badly written JVM prevents this direct control. This is becoming less and less of an issue with the development of silicon implementations of the JVM such as the aJile Systems' aJ-100 microcontroller or the Zucotto Systems' Xpresso family of processors. Both of these companies are producing controller chips suitable for hand-held Internet devices that use a chip-based implementation of JVM bytecode as machine code. In other words, these chips do not require any software support to translate JVM bytecode into native instruction sets and therefore can run at full hardware speeds, with full control over the hardware at a bit-by-bit level. With the development of Java chips, the JVM has come full circle from a clever mathematical abstraction to allow portable and secure access to a wide variety of processors to a physical chip of interest in its own right.

1.4 Programming the JVM

1.4.1 Java: What the JVM Isn't

Java, it must be stressed, is not the same as the Java Virtual Machine, although they were designed together and often are used together. The Java programming language is a high-level programming language designed to support secure platform-independent applications in a distributed networking environment such as the Internet. Perhaps the most common use of Java is to create "applets," to be downloaded as part of a Web page and to interact with the browser without cluttering up the server's network connection. The Java Virtual Machine, on the other hand, is a shared virtual computer that provides the basis for Java applications to run.

Much of the design of Java strongly influenced the JVM. For example, the JVM is a virtual machine precisely because Java should be a platform-independent language, so it cannot make any assumption about what kind of computer the viewer uses (figure 1.13). The JVM was designed around the notion of a security verifier so that Java programs could run in a secure environment. Networking support is built into the JVM's standard libraries at a relatively low level to ensure that Java programs will have access to a standardized and useful set of networking operations. There is, however, no necessary connection between the two products. A Java compiler could in theory be written to compile to native code for the PowerPC (the hardware underlying older Macintosh computers) instead of to JVM bytecode, and a compiler for any other language could be written to compile to JVM code.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 33
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

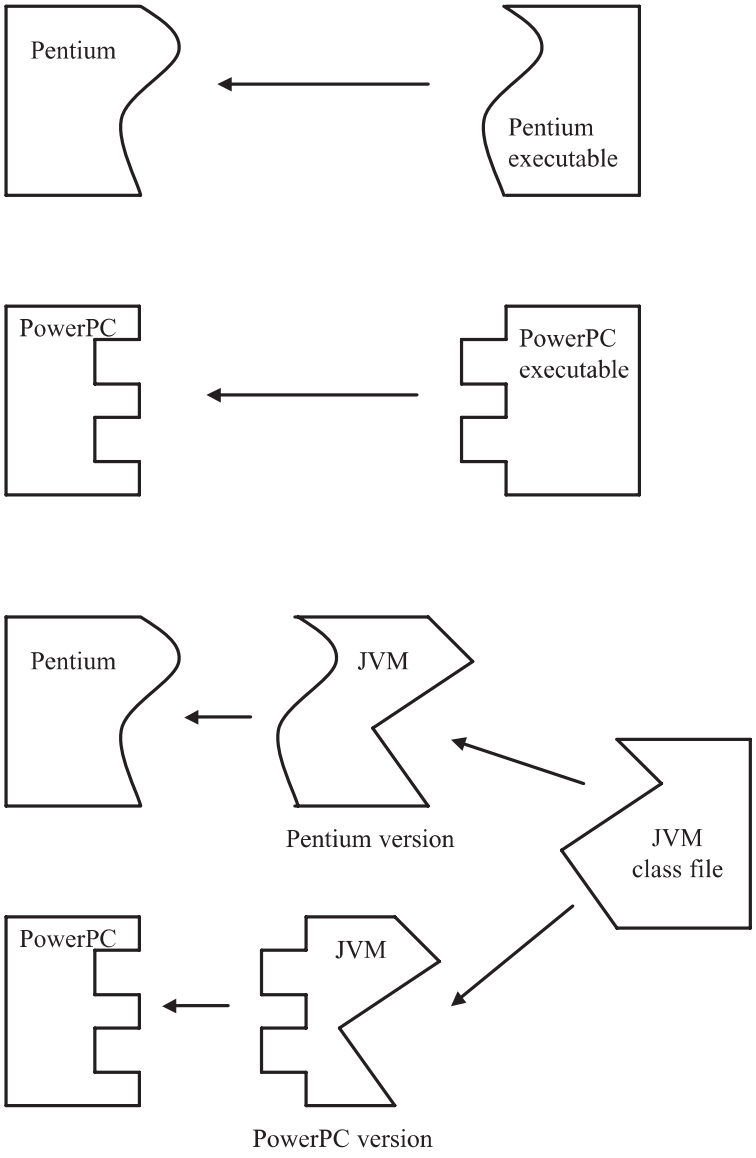


Figure 1.13 Hardware compatibility and virtual machines

In particular, consider the program written in figure 1.14. This is a simple example of a Java program that outputs a given fixed string to the default system output, usually the currently active window. This, or a very similar program, is often the standard first example in any elementary programming class, although sometimes one is shown instead a program that opens a new window and displays a message. When examined in more detail, however, the Java program itself is found to do nothing of the sort. The Java program, by itself, does nothing. In order to be executed, it must first be run through a translation program of some sort (the compiler) in order to produce an executable class file. Only this can be run to produce the desired output.

Java, or any programming language, is better viewed as a structure for specifying computations to be performed by the computer. A program in any language is a specification of a particular computation. In order for the computer to do its thing, the specification (program) must

```
public class JavaExample {
       public static void main(String [] args) {
              System.out.println("This is a sample program.");
       }
}
```

Figure 1.14 Sample program in Java

be translated into the machine code that the computer ultimately recognizes as a sequence of program steps.

1.4.2 Translations of the Sample Program

There are usually many ways of specifying a given computation. A very similar program can be written in many other languages. For example, figures 1.15, 1.16, and 1.17 show programs with identical behavior written in C, Pascal, and C++. These programs also show great similarity in overall structure. The core of the program is written in a single line that somehow accesses a

```
#include <stdio.h>

int main()
{
        (void) printf("This is a sample program.\n");
}
```

Figure 1.15 Sample program in C

```
Program PascalExample(output);

begin
        writeln ('This is a sample program.');
end.
```

Figure 1.16 Sample program in Pascal

```
using namespace std;
#include <iostream.h>
int main()
{
        cout << "This is a sample program." << endl;
}
```

Figure 1.17 Sample program in C++

fixed string (of ASCII characters) and calls a standard library function on it to pass it to a default output device. The differences are subtle and in the details; for example, the Pascal program has a name (PascalExample), while the C and C++ versions do not. In both C and C++, the program must explicitly represent the end-of-line carriage return, while Java and Pascal have a function that will automatically put a return at the end of the line. These differences, however, are relatively

small when compared with the tremendous amount of similarity, especially when one considers the differences between this and machine-level code.

In light of the preceding discussion of the architecture and organization of a typical computer, consider the following: Few, if any, CPUs have enough registers within the ALU or Control Unit to allow storing an entire string. (This is especially true since there is no necessary upper limit to the length of a string in the abstract. Instead of printing a mere sentence, the programs could have printed an entire textbook.) No CPU has a single instruction to print a string to the screen. Instead, the compiler must break the program down into sufficiently small steps that are within the computer's instruction set.

1. The string itself must be stored somewhere in main memory, and
2. the CPU must determine where that storage location is.
3. The CPU must also determine which output peripheral should print the message, and possibly
4. what type it is. Finally,
5. it must pass the appropriate instructions to the peripheral, telling it
6. where the string is stored,
7. that the string is a string (and not an integer or a floating point number), and
8. that the appropriate action to take is to print it (possibly while automatically appending a return).

A single line of code may translate into eight or more individual operations that the computer must perform. In fact, there's no limit to the number of operations that might be performed in a single line of code; a line in Java like

```
i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
```

requires an operation for every addition, in this case nine separate calculations. Since there's no limit to the theoretical complexity of a mathematical formula, there's no limit to the complexity of a single Java statement.

1.4.3 High- and Low-Level Languages

This encapsulation of many machine language instructions in a single line of code is typical of what are called **high-level** languages. Java, Pascal, and so forth are typical examples of one style of such languages. The task of a Java compiler is to take a complicated statement or expression and produce an appropriate series of machine instructions to perform the task.

By contrast, a **low-level** language is characterized by a very close relationship between operations in machine language and statements in the program code. Using this definition, machine language is, of course, a very low-level language, since there is always a 1:1 relationship between a machine language program and itself. **Assembly language** is a slightly more human-readable, but still low-level, language designed to promote total control over the machine and the machine code instructions but still be readable and understandable by the programmer. Assembly language is also characterized by a 1:1 relationship between assembly language instructions and machine code instructions.

In machine language, every element of the instruction (also called **opcode**, an abbreviation for "operation code") is, like everything else in the computer, a number. An earlier section

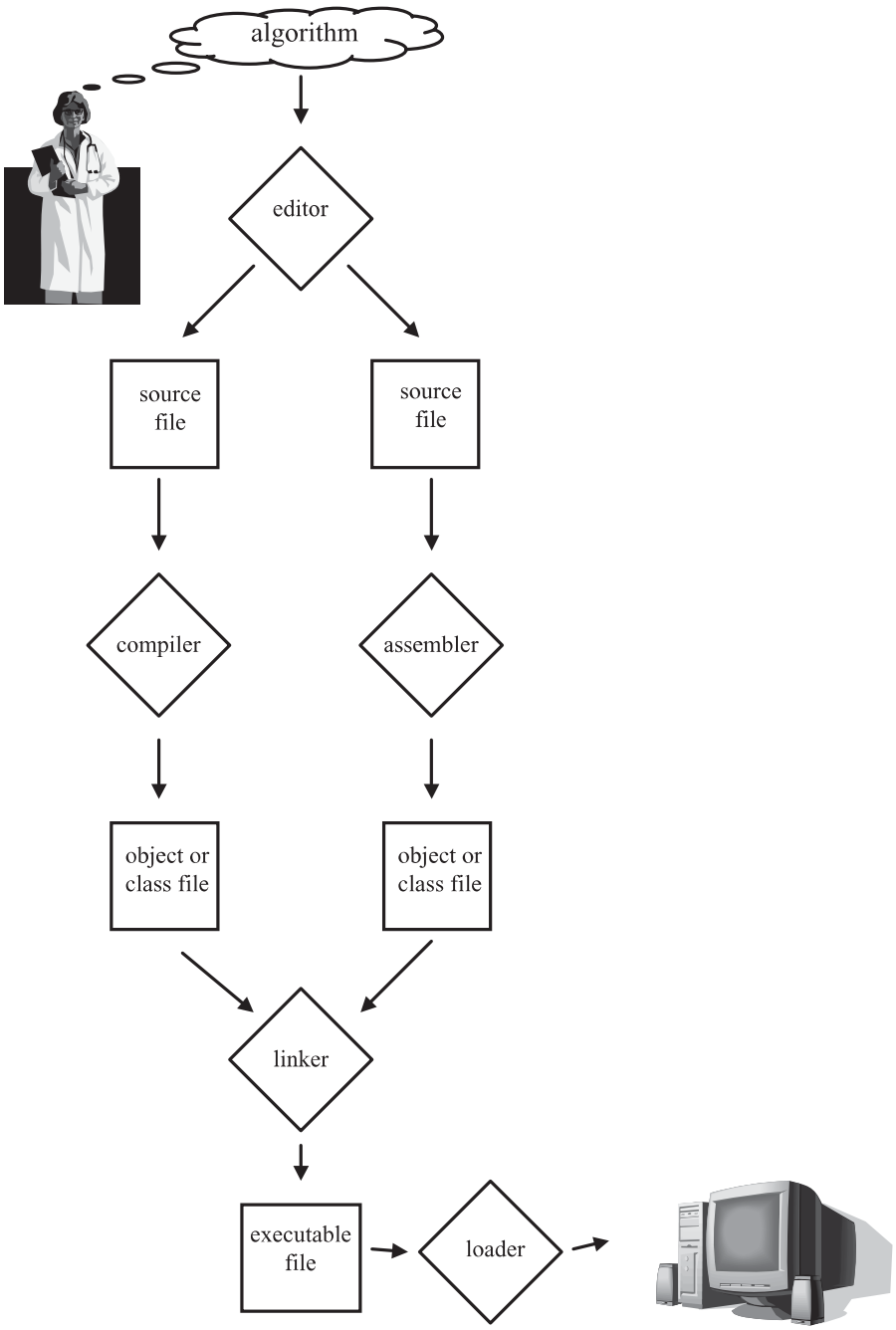


Figure 1.18 The programming process

mentioned in passing, for instance, that the opcode 89 (0x59) is meaningful to the JVM and causes it to duplicate a particular piece of information. In assembly language, every opcode is also given a **mnemonic** (pronounced "num-ON-ik," from the Greek word for memory) that explains or summarizes exactly what it does. The corresponding mnemonic for opcode 89 is `dup`, short for "duplicate." Similarly, the mnemonic `iadd`, short for "integer add," corresponds to the machine opcode used to perform an integer addition. The task of the translation program, in this

case called an **assembler**, is to translate each mnemonic into its appropriate binary opcode and to gather the appropriate pieces of data that the computer needs (figure 1.18).

1.4.4 The Sample Program as the JVM Sees It

The JVM machine code version of the sample program(s) above would thus be a long and mostly unreadable binary string. A version of the program that corresponds to the machine code is presented in figure 1.19. This program was written (by hand) in a (low-level) JVM assembly language called `jasmin`, but could also have been written by a compiler starting from one of the original sample programs. Notice that the program is much longer—almost 30 lines instead of only 3 or 4—and much more difficult to understand. This is because many of the things that you take

```
; defines the class file associated with this as jasminExample.class
.class public jasminExample
; defines jasminExample as a subclass of Object
.super java/lang/Object

; boilerplate needed for object creation
.method public <init>()V
     aload_0

     invokespecial java/lang/Object/<init>()V
     return
.end method

.method public static main([Ljava/lang/String;)V
     ; we need two stack elements, for System.out and the string
     .limit stack 2

     ; find System.out (an object of type PrintStream)
     ; and put it on the stack
     getstatic java/lang/System/out Ljava/io/PrintStream;

     ; find the string (characters) to be printed
     ; and put it on the stack
     ldc "This is a sample program."

     ; invoke the PrintStream/println method
     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

     ; ... and that's it!
     return
.end method
```

Figure 1.19 Sample program in JVM assembly language

Java code	`jasmin` code	JVM machine code
$x = 1 + 2 + 3 + 4 + 5;$	`bipush 1`	0x10
		0x01
	`bipush 2`	0x10
		0x02
	`iadd`	0x60
	`bipush 3`	0x10
		0x03
	`iadd`	0x60
	`bipush 4`	0x10
		0x04
	`iadd`	0x60
	`bipush 5`	0x10
		0x05
	`iadd`	0x60
	`istore_1`	0x3c

Figure 1.20 Sample expression in Java, `jasmin`, and JVM machine code (see the next chapter for details)

for granted in writing high-level programs must be specified exactly and explicitly. For example, in Java, every class is implicitly a subclass of type Object unless specified otherwise. The JVM requires, instead, that every class define its relationship to other classes explicitly. The comments (beginning with semicolons) in figure 1.19 present a more detailed, line-by-line description (for the human reader) of exactly how the (implicit) operations are defined and carried out.

Notice, though, that although the notions of class, subclass, and so forth are explicitly supported and used in Java, there is nothing especially Java-specific about the JVM low-language program presented in this section. In fact, just as it is the task of a Java compiler to convert the high-level Java code to something akin to JVM machine code, it is the task of a C++ or Pascal compiler to do the same with its program code for a specific platform. There is no reason that a Pascal compiler couldn't be written that produces JVM code as its final (compiled) output instead of PowerPC or Pentium machine code; the program would then run on a JVM (for instance, on a JVM emulator, inside a Web browser, or on a special-purpose chip like those mentioned earlier) instead of on the specific chip hardware (figure 1.20).

1.5 Chapter Review

- Computers are simply high-speed algorithmic devices; the details of their construction as electronic devices are less important than the details of how they work as information processing devices.
- The main part of a computer is the Central Processing Unit (CPU), which in turn contains the Control Unit, Arithmetic and Logical Unit (ALU), and Floating Point Unit (FPU). This is where all computations actually happen and programs actually run.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 39
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

- Memory and peripherals are connected to the CPU through a collection of electrical buses that carry information to and from the CPU.
- All values stored in a conventional computer are stored as binary digits or bits. These are grouped for convenience into larger units such as bytes and words. A particular bit pattern may have several interpretations as an integer, a floating point number, a character or sequence of characters, or even a set of instructions to the computer itself. Deciding what types of data a given bit patterns represents depends on the context and is largely the programmer's responsibility.
- Different CPU chips have different instruction sets representing different things that they can do and ways that they do things. Every CPU, then, needs a different executable program written in a different kind of machine language even to run the same program.
- A virtual machine is a program that runs atop a real CPU and interprets machine instructions as though it itself were a CPU chip. The Java Virtual Machine (JVM) is a common example of such a program, and can be found on almost every computer and in every Web browser worldwide.
- Virtual machines have many advantages over conventional silicon-based CPUs, such as portability, fewer hardware limitations, ease of updates, and security. Virtual machines may have a big disadvantage in speed.
- Java is an example of a high-level language that can combine many machine-code instructions into a single statement. C, Pascal, and C++ are similar high-level languages. These languages must be compiled to convert their code into a machine-executable format.
- Low-level languages like assembly language have a tight, typically 1:1 relationship between program statement and machine instructions.
- There is no necessary connection between Java and the JVM; most Java compilers compile to JVM executable code, but a C, Pascal, or C++ compiler could do this as well. Similarly, a Java compiler could compile to Pentium 4 native code, but the resulting program would not run on a Macintosh computer.

1.6 Exercises

1. What is an algorithm?
2. Is a recipe as given in a cookbook an example of an algorithm?
3. Name the part of the computer described by each of the following phrases:
   - The heart and ultimate controller of the computer, the place where all calculations are performed
   - The part of the computer responsible for moving data around within the machine
   - The part of the computer responsible for all computations, such as addition, subtraction, multiplication, and division
   - A set of wires used to interconnect different devices for data connectivity
   - A device for reading, displaying, or storing data
   - A place for short-term storage for data and executing programs
4. How many different patterns could be stored in a 16-bit register? What is the largest value that could be stored as a (two's complement) signed integer in such a register? What is the

smallest value? How about the largest and smallest values that could be stored as unsigned integers?

5. Convert the following 16-bit binary numbers into hexadecimal and signed decimal numbers (no, you don't get to use a calculator!):

   - 1001110011101110
   - 1111111111111111
   - 0000000011111111
   - 0100100010000100
   - 1111111100000000
   - 1100101011111110

6. Convert the following 32-bit IEEE floating point numbers from hexadecimal into standard decimal notation.

   - 0x40200000
   - 0x41020000
   - 0xC1060000
   - 0xBD800000
   - 0x3EAAAAAB
   - 0x3F000000
   - 0x42FA8000
   - 0x42896666
   - 0x47C35000
   - 0x4B189680

7. Convert the following decimal numbers into 32-bit IEEE floating point notation.

   - 2.0
   - 45.0
   - 61.01
   - −18.375
   - −6.68
   - 65536
   - 0.000001
   - 10000000.0

8. Are there any numbers that can be represented exactly as a 32-bit integer but not as a 32-bit IEEE floating point number? Why or why not?
9. Using a standard ASCII table (check the Internet or appendix E), what 4 hexadecimal bytes would represent the string "Fred"?
10. What ASCII character string would correspond to the hexadecimal number 0x45617379?
11. True or false: the more is there are in a binary number, the larger it is. Why or why not?
12. Why won't executables created for a Windows Pentium IV run on a PowerPC-based Macintosh (without special software support)?
13. What is the most important advantage of a virtual machine over a chip-based architecture?

14. What is the most important disadvantage?
15. What languages can be used to write programs for the Java Virtual Machine (JVM)?
16. How many low-level machine instructions correspond to the following statement?

$$x = a + (b^*c) + (d^*e);$$

1.7 Programming Exercises

1. Write a program (in any language approved by the instructor) to read in a 32-bit (binary) signed integer and to output its decimal equivalent.
2. Write a program (. . .) to read in a 32-bit (binary) floating point number and to output its decimal equivalent.
3. Write a program (. . .) to read in a decimal floating point number and output its IEEE 64-bit hexadecimal equivalent. (Note: this may require additional reading.)
4. Write a program (. . .) to read two 32-bit floating point numbers (in hexadecimal IEEE format) A and B and to output their product $A \cdot B$ in hexadecimal. Do not convert internally to decimal floating point numbers and multiply using the language's multiply operation.
5. Write a program (. . .) to read two 32-bit floating point numbers (in hexadecimal IEEE format) A and B and to output their sum $A + B$ in hexadecimal. Do not simply convert to decimal and add.
6. Write a program (. . .) to read a 32-bit floating point number A (in hexadecimal) and to output its reciprocal $\frac{1.0}{A}$ in both hexadecimal and decimal. Can you use this program in conjunction with program 4 to perform floating point division? How?

APPENDIX A

Digital Logic

A.1 Gates

Without the transistor, the modern computer would not be possible. As suggested by Moore's Law (transistor density doubles every eighteen months), the ability to fabricate and arrange transistors is the fundamental way that data is controlled, moved, and processed within the chips at the heart of a computer.

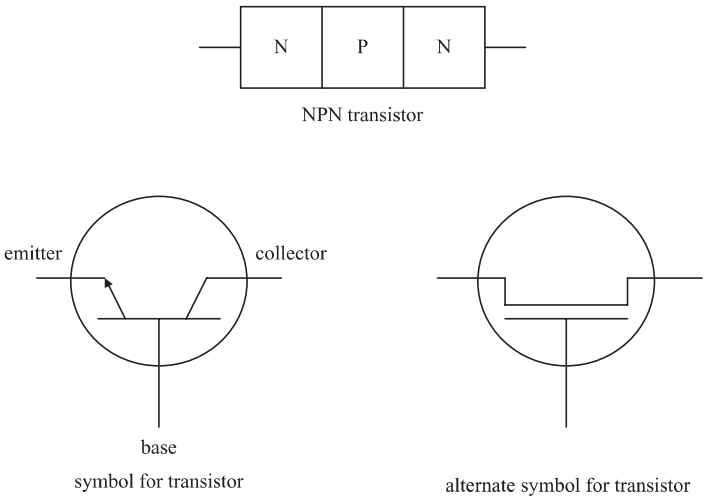


Figure A.1 Sample transistors and symbols

A transistor can be regarded as a kind of electronically controlled switch. A typical transistor and its electronic diagram are shown in figure A.1. Under normal circumstances, electricity flows from the emitter to the collector like water through a pipe or cars through a tunnel. However, this is possible only as long as appropriate application of an electrical signal to the base permits the flow of electricity to pass. Without this control signal, it's as though someone had turned off a faucet (or set up a traffic light). When this happens, electricity can't get through. By combining these switches in combination, engineers can create dependency structures; electricity will flow only if all transistors are energized, for example.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 43
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

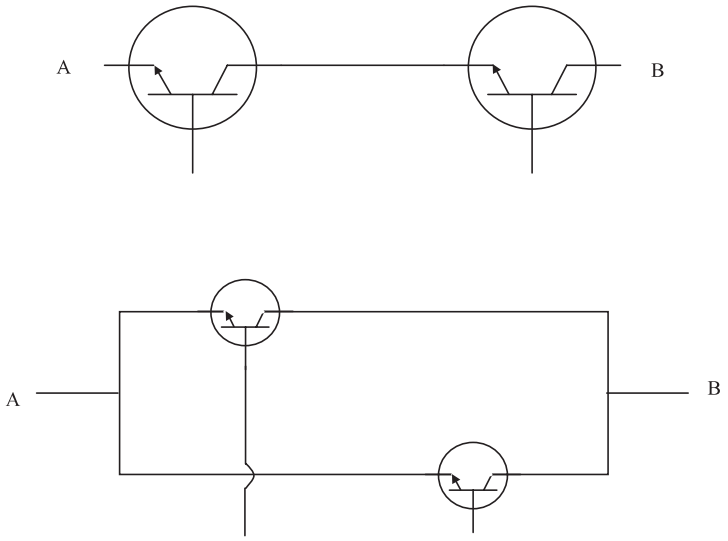


Figure A.2 Transistors in series (top) and parallel (bottom)

Figure A.2 shows two examples of dependency structure. In the **series** circuit, both transistors share a common path, and electricity must be able to flow through both circuits at the same time if it is to flow from point A to point B. In the **parallel** circuit, each transistor has its own current path, and any transistor can allow current to flow across the circuit.

The electrical circuit shown in figure A.3 is an example of two transistors connected in series. In order for electricity to flow from the power source (V_{cc}) to the output, both

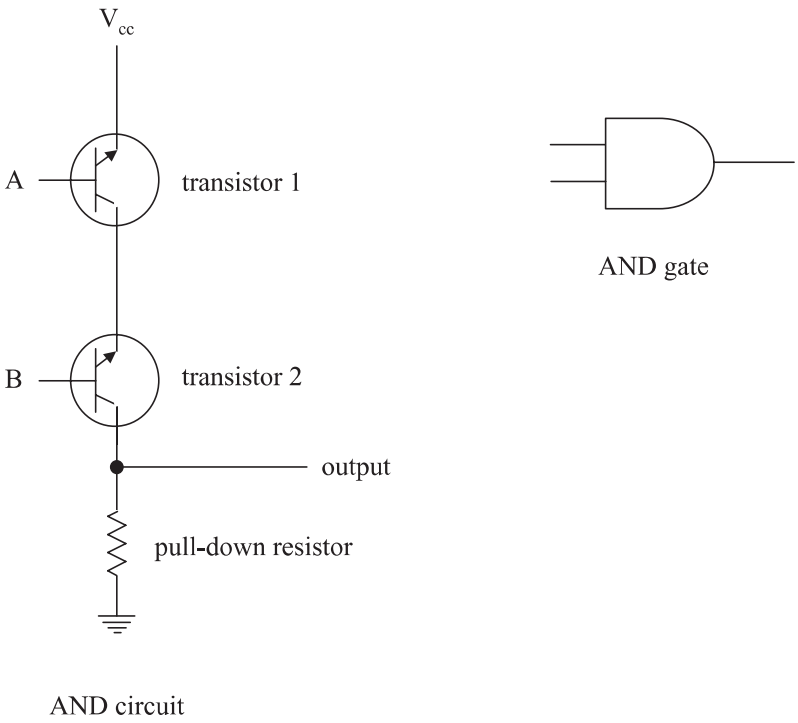


Figure A.3 Simplified AND gate and symbol

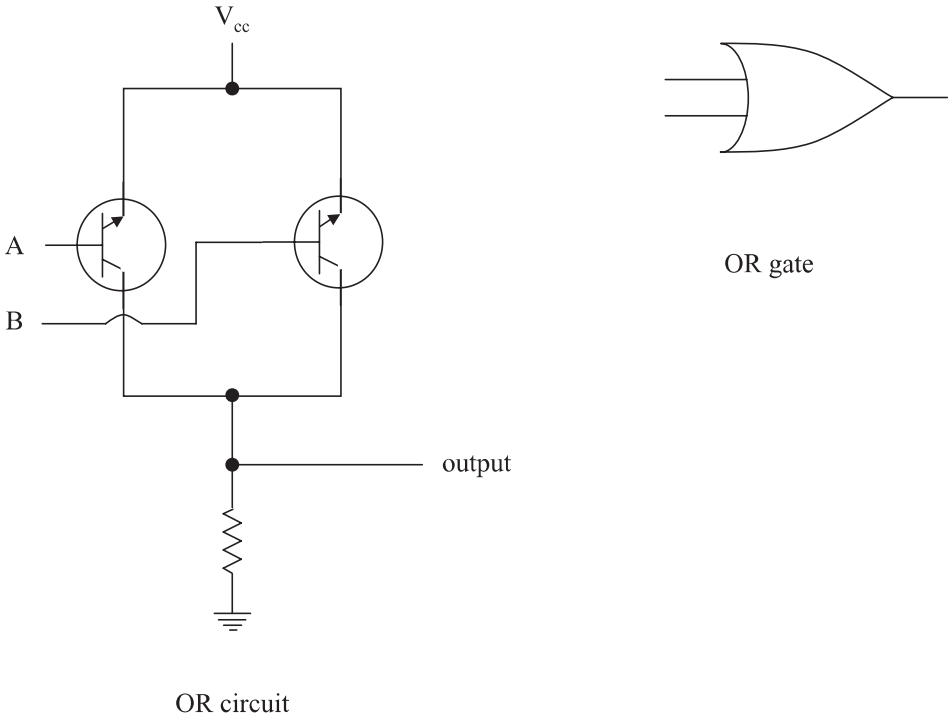


Figure A.4 Simplified OR gate and symbol

transistors need to be signaled to allow current to pass. If either transistor is an open switch, then electricity can't flow. This implies that electricity can flow (there is power at the output) only if transistor 1 is closed **AND** transistor 2 is closed. Similarly, in figure A.4, two transistors in parallel, will allow current to flow if either the first transistor **OR** the second transistor is closed (or both).

The basic building blocks of a computer consist of simple circuits, called **gates**, that implement these simple, logical signals. These gates typically contain between one and six transistors each and implement basic logical or arithmetic operations. Remember that the basic values used in logic are True and False. If "current is flowing" represents True, then the circuit in figure A.3 is an implementation of the logical function AND in a simple (and somewhat idealized—don't try to build this at home out of Radio Shack components!) gate. Other functions available include OR (figure A.4), NOT, NAND, and NOR. The symbols used for drawing various gates are given in figure A.5. Note that each gate (except NOT) has two inputs and a single output. Also noted that the NOT gate symbol has a circle (representing signal inversion) at the output line. The symbol for a NAND (Not-AND) gate is the same as the symbol for an AND gate, but with the little inversion circle at the output. Similarly, a NOR (Not-OR) gate is just an OR gate with the inversion circle.

A.2 Combinational Circuits

Complicated networks of gates can implement any desired logical response. For example, the XOR (eXclusive OR) function is True if and only if exactly one, but not both, of the inputs is

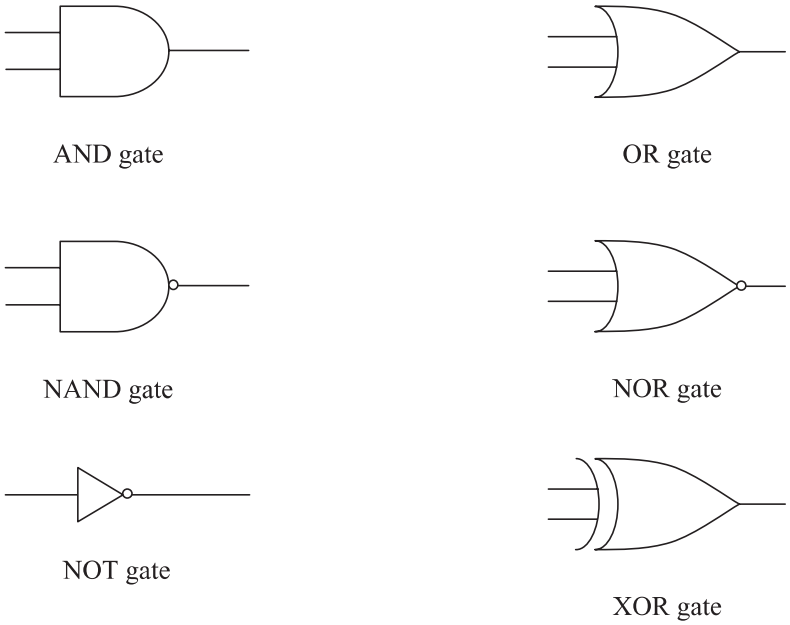


Figure A.5 Gate types and symbols

True. In logical notation, this can be written

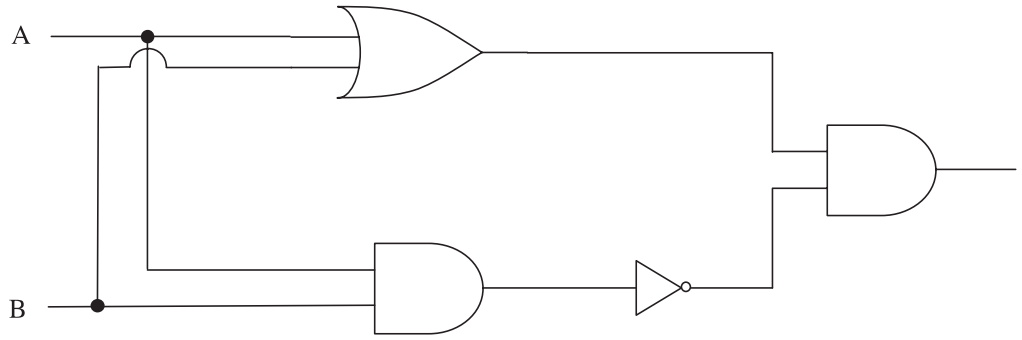
As: A XOR B = (A OR B) AND (NOT (A AND B))
As a truth table, this can be written as in table A.1
And finally, as a circuit, as figure A.6

The basic concept of combinatorial circuits is that the output is always a function of the current input signal(s) without regard to signal history. Such circuits can be very useful for implementing simple decisions or arithmetic operations. Although a full description of how to design such circuits is beyond the scope of this appendix, figure A.7 shows how a set of gates could implement simple binary addition (as part of the ALU of a computer).

Specifically, this circuit will accept two single-bit signals (A and B) and output both their sum and whether or not there is a carry. (This circuit is sometimes called a "half adder.") Examination of the binary addition table shows that the sum of two bits is simply their XOR, while a carry is generated if and only if both bits are 1s, or in other words, their AND. Similar analysis can yield a more complicated design capable of incorporating a carry bit from a previous addition stage (a "full adder"; see figure A.8) or adding several bits at one time (as in a register). Figure A.9 shows how a 4-bit addition can be performed by four

A	B	Result
True	True	False
True	False	True
False	True	True
False	False	False

Table A.1 Truth table defining eXclusive OR



A XOR B

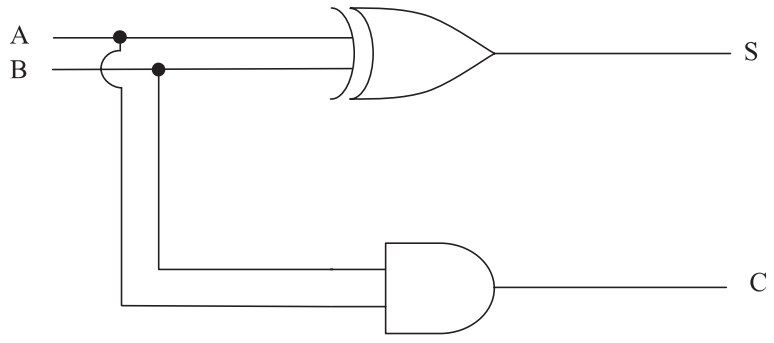
Figure A.6 Implementation of XOR in gates

replications of the full adder circuit; 32 replications, of course, could add together two registers on the Pentium. One can also build a circuit to perform binary multiplication and other operations. Multiply these simple gates by several billion, and you approach the power and complexity of a Pentium.

A.3 Sequential Circuits

In contrast to a combinatorial circuit, a sequential circuit retains some notion of memory and of the history of the circuit. The output of a sequential circuit depends not only on the present input, but also on the past inputs. Another way of expressing this is to say that such a circuit has an internal state. By using the internal state, one can store information for later use, essentially creating the sort of memory necessary for a register.

One of the simplest sequential circuits is the **S–R flip-flop**, illustrated in figure A.10. To understand this circuit, let's first pretend that S and R are both 0 (False), Q is 0, and $\bar{Q}$ is 1 (True). Since $\bar{Q}$ is True, R NOR $\bar{Q}$ is 1. Similarly, S NOR Q is 1. We thus see that this configuration is internally consistent and will remain stable as long as the individual components



A + B

Figure A.7 Single-bit half adder

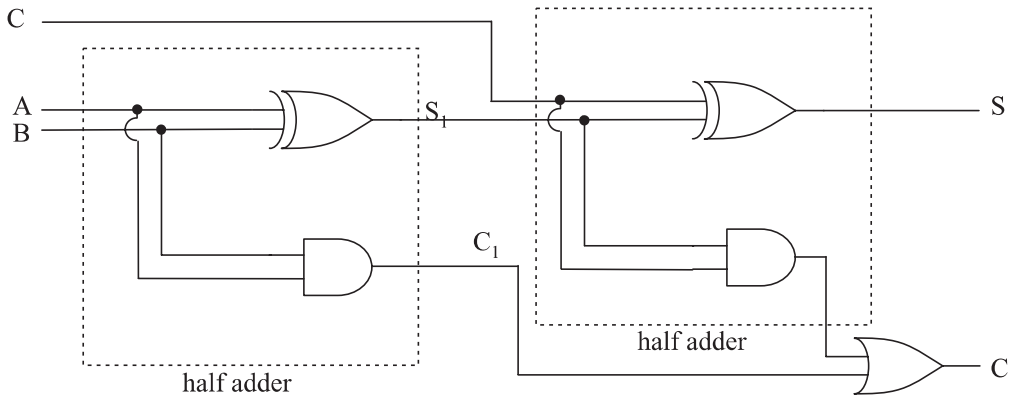


Figure A.8 Single-bit full adder (with carry)

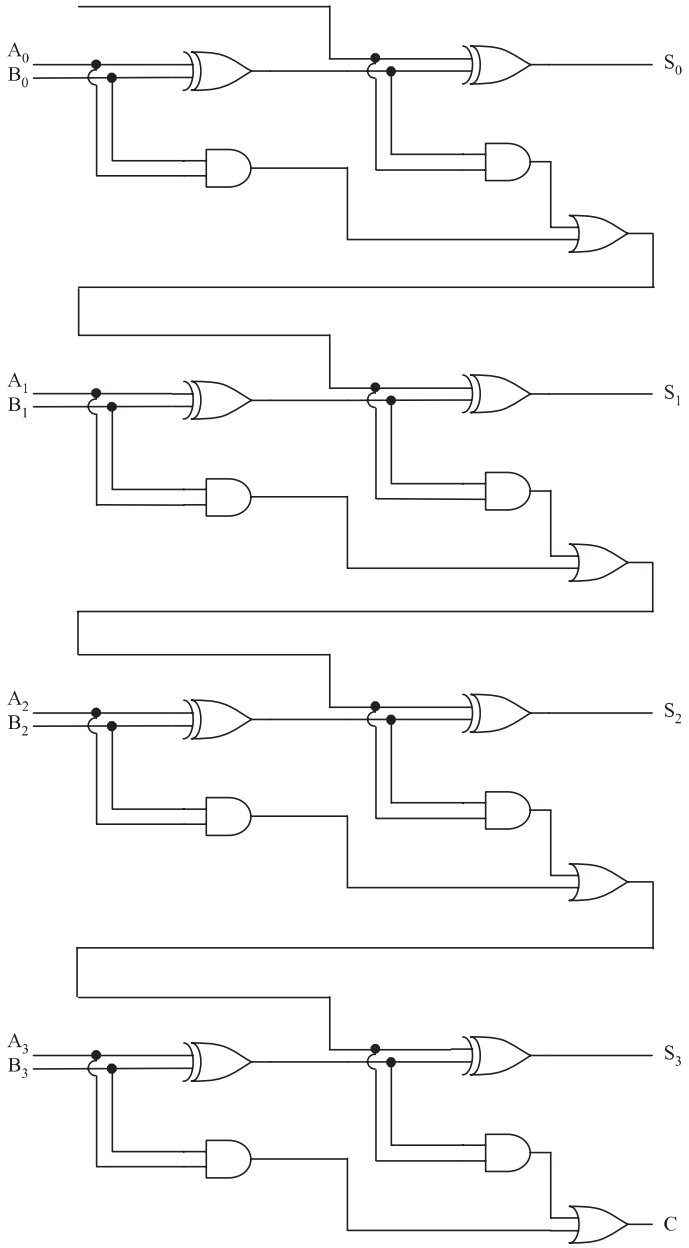


Figure A.9 Cascaded 4-bit adder, including internal carries

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 48
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.

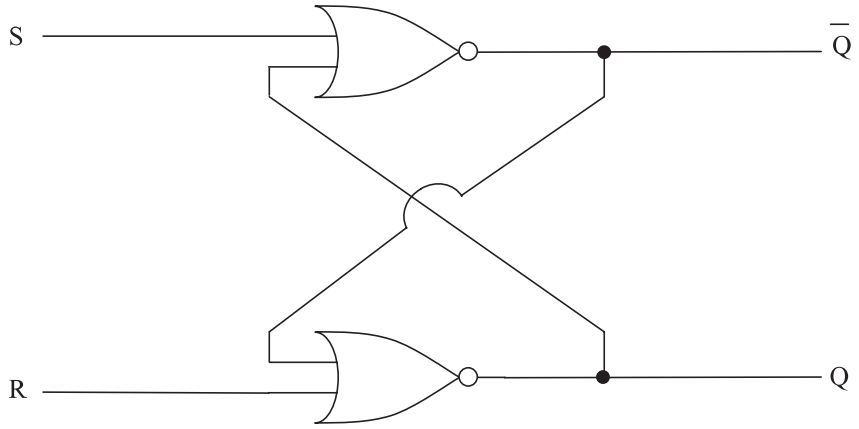


Figure A.10 S–R flip-flop, a circuit with memory

work (which usually means as long as they have power). A similar analysis will show that if S and R are both 0, while Q is 1, the result will be a self-consistent, stable state.

This simple flip-flop can be used as a 1-bit memory; the value of Q is the value stored in the memory. The signals S and R can be used to set or reset (set to 1 or 0) the flip-flop, respectively. Observe that if S becomes 1, this will force the output of the upper NOR gate to be 0. With R 0 and $\bar{Q}$ 0, the output of the lower NOR gate is 1, so the value stored is now 1. A similar process, if R is set to 1, forces Q to 1 and Q to 0.

Unfortunately, if S and R are both 1, then bad things happen. According to the notation, Q and $\bar{Q}$ should always be opposites of each other, but if both inputs are 1, both outputs will be 0 (you can confirm this for yourself). In a purely mathematical sense, we can regard this as the logical equivalent of dividing by 0—something to be avoided instead of analyzed. Similarly, even a brief power spike on one input wire can cause the flip-flop to unexpectedly change state—something to be avoided if possible.

However, since this is to be avoided, other sorts of sequential circuits are more commonly used in computers. Most common circuits combine control signals with timing signals (usually called a **clock signal**) to synchronize the control signals and keep short-term fluctuations from influencing the circuit's memory. Notice that in figure A.11, the clock signal acts to enable the flip-flop to change state. If the clock signal is low, then changes on S or R cannot affect the memory state of the circuit.

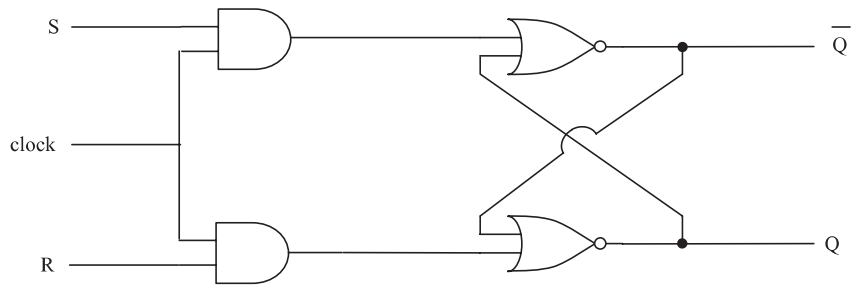


Figure A.11 A clocked S–R flip-flop

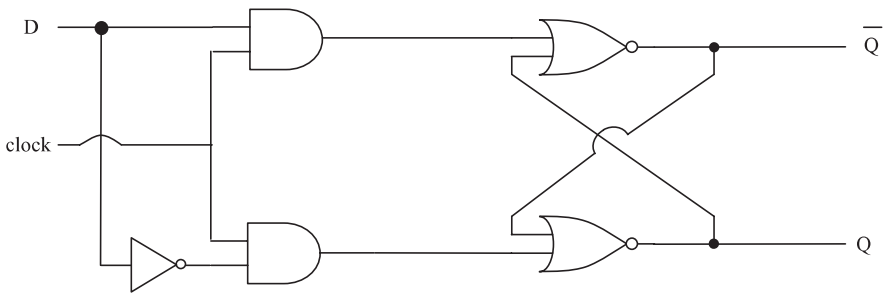


Figure A.12 A D flip-flop

We can extend this clocked flip-flop further to something perhaps more useful as well as safer. The circuit diagram in figure A.12 illustrates a D flip-flop. As you can see, this circuit has only one input beyond the clock. The D input is tied to the S input of the flip-flop, while $\bar{D}$, the complement of D, is tied to the R input. This keeps both S and R from being 1 at the same time. When a clock pulse occurs, the value of D will be stored in the flip-flop (if the value of D is 1, then Q becomes 1; if the value of D is 0, Q becomes 0). Until a clock pulse occurs, changes in D will have no effect on the value of the flip-flop. This makes a D flip-flop very valuable for copying and storing data—for example, in reading from an I/O device to a register for later use.

Another variation on the SR flip-flop yields the T flip-flop (figure A.13). Like the D flip flop, this is a single-input extension of the clocked S–R flip-flop, but the additional feedback wires control how the input/clock pulse is gated through; in this circuit, the flip-flop will change state ("toggle") each time the input is triggered. For example, if Q is 1 (and $\bar{Q}$ is 0, by extension), then the next pulse will trigger the bottom input wire (essentially the R input in the previous circuits) and reset the flip-flop so that Q is 0.

A.4 Computer Operations

Using these building blocks, it's possible to build the higher-level structures associated with computer architectures. In particular, a collection of simple 1-bit flip-flops (such as the S–R flip-flop) can implement a simple register and hold data until it is needed. To add, for example,

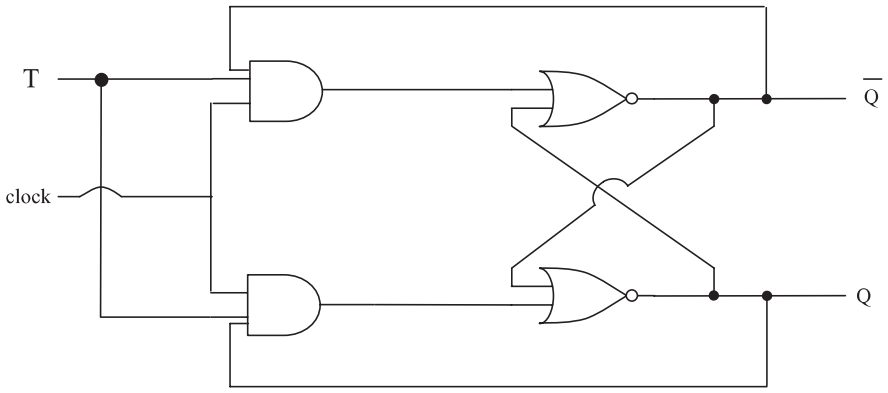


Figure A.13 A T flip-flop

two 1-bit registers, the Q output of each register can be electrically connected to one input of a simple addition circuit. The Q output of the other register would be connected to the other adder input. The resulting output bit would be the addition of these 2 register bits, which could be captured and stored (via a D flip-flop) in yet another register. The T flip-flop could be used in conjunction with adder circuits to build a simple pulse counter. Of course, these descriptions oversimplify dreadfully, but they give something of the feel for the task presented to the computer designer.

MOSAID Technologies Inc., Exhibit 2029
Page 2029 - 51
IPR2025-01489, Infineon Technologies Americas Corp. v. MOSAID Technologies Inc.