CHAPTER 1

Introduction to the Microprocessor and Computer

INTRODUCTION

This chapter provides an overview of the Intel family of microprocessors. Included is a discussion of the history of computers and the function of the microprocessor in the microprocessor-based computer system. Also introduced are terms and jargon used in the computer field, so **computerese** is understood and applied when discussing microprocessors and computers.

The block diagram and a description of the function of each block detail the operation of a computer system. The chapter also shows how the memory and input/output (I/O) system of the personal computer function. Finally, the way that data are stored in the memory is provided, so that each data type can be used as software is developed. Numeric data are stored as integers, floating-point, and binary-coded decimal (BCD); alphanumeric data are stored by using the ASCII (American Standard Code for Information Interchange) code.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Converse by using appropriate computer terminology such as bit, byte, data, real memory system, expanded memory system (EMS), extended memory system (XMS), DOS, BIOS, I/O, and so forth.
2. Briefly detail the history of the computer and list applications performed by computer systems.
3. Provide an overview of the various 80X86 and Pentium–Pentium pro.
4. Convert between binary, decimal, and hexadecimal numbers.
5. Differentiate and represent numeric and alphabetic information as integers, floating-point, BCD, and ASCII data.

1-1 A HISTORICAL BACKGROUND

This first section outlines the historical events leading to the development of the microprocessor and, specifically, the extremely powerful and current 80X86,[1] Pentium, Pentium Pro, Pentium III, and Pentium 4[2] microprocessors. Although a study of history is not essential to understand the microprocessor, it furnishes interesting reading and provides a historical perspective of the fast-paced evolution of the computer.

The Mechanical Age

The idea of a computing system is not new—it has been around long before modern electrical and electronic devices were developed. The idea of calculating with a machine dates to 500 B.C. when the Babylonians invented the **abacus,** the first mechanical calculator. The abacus, which used strings of beads to perform calculations, was used by the ancient Babylonian priests to keep track of their vast storehouses of grain. The abacus, which was used extensively and is still in use today, was not improved until 1642, when mathematician Blaise Pascal invented a calculator that was constructed of gears and wheels. Each gear contained 10 teeth that, when moved one complete revolution, advanced a second gear one place. This is the same principal that is used in the automobile's odometer mechanism and is the basis of all mechanical calculators. Incidentally, the PASCAL programming language is named in honor of Blaise Pascal for his pioneering work in mathematics and with the mechanical calculator.

The arrival of the first practical geared, mechanical machines used to automatically compute information dates to the early 1800s. This is before humans invented the light bulb or before much was known about electricity. In this dawn of the computer age, humans dreamed of mechanical machines that could compute numerical facts with a program—not merely calculating facts, as with a calculator.

In 1937 it was discovered through plans and journals that one early pioneer of mechanical computing machinery was Charles Babbage, aided by Augusta Ada Byron, the Countess of Lovelace. Babbage was commissioned in 1823 by the Royal Astronomical Society of Great Britain to produce a programmable calculating machine. This machine was to generate navigational tables for the Royal Navy. He accepted the challenge and began to create what he called his **Analytical Engine.** This engine was a mechanical computer that stored 1000 20-digit decimal numbers and a variable program that could modify the function of the machine to perform various calculating tasks. Input to his engine was through punched cards, much as computers in the 1950s and 1960s used punched cards. It is assumed that he obtained the idea of using punched cards from Joseph Jacquard, a Frenchman who used punched cards as input to a weaving machine he invented in 1801, which is today called Jacquard's loom. Jacquard's loom used punched cards to select intricate weaving patterns in the cloth that it produced. The punched cards programmed the loom.

After many years of work, Babbage's dream began to fade when he realized that the machinists of his day were unable to create the mechanical parts needed to complete his work. The Analytical Engine required more than 50,000 machined parts, which could not be made with enough precision to allow his engine to function reliably.

The Electrical Age

The 1800s saw the advent of the electric motor (conceived by Michael Faraday); with it came a multitude of motor-driven adding machines, all based on the mechanical calculator developed by Blaise Pascal. These electrically driven mechanical calculators were common pieces of office equipment until well into the early 1970s, when the small hand-held electronic calculator, first introduced by Bomar, appeared. Monroe was also a leading pioneer of electronic calculators, but its machines were desktop, four-function models the size of cash registers.

In 1889, Herman Hollerith developed the punched card for storing data. Like Babbage, he too apparently borrowed the idea of a punched card from Jacquard. He also developed a mechanical machine—driven by one of

[1]80X86 is shorthand notation that includes the 8086, 8088, 80188, 80286, 80386, and 80486 microprocessors.

[2]Pentium, Pentium Pro, Pentium II, Pentium III, and Pentium 4 are registered trademarks of Intel Corporation.

the new electric motors—that counted, sorted, and collated information stored on punched cards. The idea of calculating by machinery intrigued the United States government so much that Hollerith was commissioned to use his punched-card system to store and tabulate information for the 1890 census.

In 1896, Hollerith formed a company called the Tabulating Machine Company, which developed a line of machines that used punched cards for tabulation. After a number of mergers, the Tabulating Machine Company was formed into the International Business Machines Corporation, now referred to more commonly as IBM, Inc. The punched cards used in computer systems are often called **Hollerith cards,** in honor of Herman Hollerith. The 12-bit code used on a punched card is called the **Hollerith code.**

Mechanical machines driven by electric motors continued to dominate the information processing world until the construction of the first electronic calculating machine in 1941 by a German inventor named Konrad Zuse. His Z3 calculating computer, as pictured in Figure 1–1, was used in aircraft and missile design during World War II for the German war effort. Had Zuse been given adequate funding by the German government, he most likely would have developed a much more powerful computer system. Zuse is today finally receiving some belated honor for his pioneering work in the area of digital electronics which began in the 1930s and for his Z3 computer system.

It has recently been discovered (through the declassification of British military documents) that the first electronic computer was placed into operation in 1943 to break secret German military codes. This first electronic computing system, which used vacuum tubes, was invented by Alan Turing. Turing called his machine **Colossus,** probably because of its size. A problem with Colossus was that although its design allowed it to break secret German military codes generated by the mechanical **Enigma machine,** it could not solve other problems.

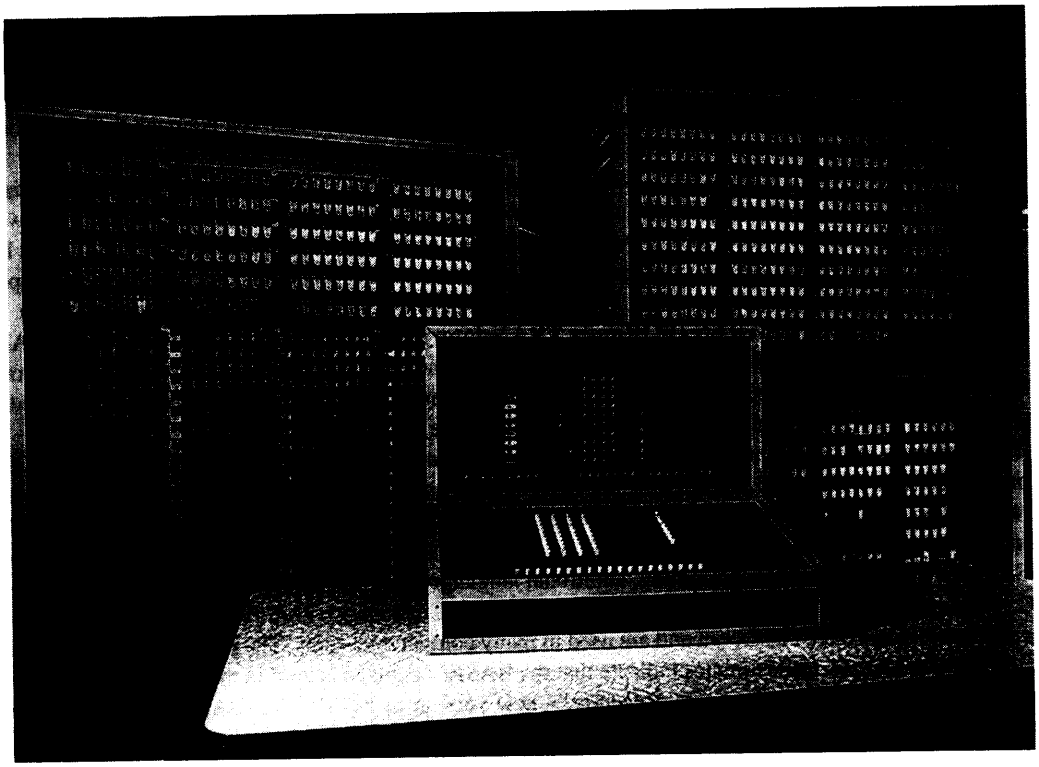


FIGURE 1–1 The Z3 computer developed by Konrad Zuse used a 5.33 Hertz clocking frequency. (Photo courtesy of Horst Zuse, the son of Konrad.)

Colossus was not programmable—it was a fixed-program computer system, which today is often called a **special-purpose computer.**

The first general-purpose, programmable electronic computer system was developed in 1946 at the University of Pennsylvania. This first modern computer was called the ENIAC **(Electronics Numerical Integrator and Calculator).** The ENIAC was a huge machine, containing over 17,000 vacuum tubes and over 500 miles of wires. This massive machine weighed over 30 tons, yet performed only about 100,000 operations per second. The ENIAC thrust the world into the age of electronic computers. The ENIAC was programmed by rewiring its circuits—a process that took many workers several days to accomplish. The workers changed the electrical connections on plug-boards that looked like early telephone switchboards. Another problem with the ENIAC was the life of the vacuum tube components, which required frequent maintenance.

Breakthroughs that followed were the development of the transistor in 1948 at Bell Labs, followed by the 1958 invention of the integrated circuit by Jack Kilby of Texas Instruments. The integrated circuit led to the development of digital integrated circuits (RTL, or resistor-to-transistor logic) in the 1960s and the first microprocessor at Intel Corporation in 1971. At that time, Intel and one of its engineers, Marcian E. Hoff, developed the 4004 microprocessor—the device that started the microprocessor revolution that continues today at an ever-accelerating pace.

Programming Advancements

Now that programmable machines were developed, programs and programming languages began to appear. As mentioned earlier, the first programmable electronic computer system was programmed by rewiring its circuits. Because this proved too cumbersome for practical application, early in the evolution of computer systems, computer languages began to appear in order to control the computer. The first such language, **machine language,** was constructed of ones and zeros using binary codes that were stored in the computer memory system as groups of instructions called programs. This was more efficient than rewiring a machine to program it, but it was still extremely time-consuming to develop a program because of the sheer number of codes that were required. Mathematician John von Neumann was the first person to develop a system that accepted instructions and stored them in memory. Computers are often called **von Neumann machines** in honor of John von Neumann. (Remember that Babbage also had developed the concept long before von Neumann.)

Once computer systems such as the UNIVAC became available in the early 1950s, **assembly language** was used to simplify the chore of entering binary code into a computer as its instructions. The assembler allowed the programmer to use mnemonic codes, such as ADD for addition, in place of a binary number such as 01000111. Although assembly language was an aid to programming, it wasn't until 1957, when Grace Hopper developed the first high-level programming language called **FLOW-MATIC,** that computers became easier to program. In the same year, IBM developed FORTRAN **(FORmula TRANslator)** for its computer systems. The FORTRAN language allowed programmers to develop programs that used formulas to solve mathematical problems. Note that FORTRAN is still used by some scientists for computer programming. Another similar language, introduced about a year after FORTRAN, was ALGOL **(ALGOrithmic Language).**

The first truly successful and widespread programming language for business applications was COBOL **(Computer Business Oriented Language).** Although COBOL usage has diminished somewhat in recent years, it is still a major player in many large business systems. Another once-popular business language is RPG **(Report Program Generator),** which allows programming by specifying the form of the input, output, and calculations.

Since these early days of programming, additional languages have appeared. Some of the more common are BASIC, C/C++, PASCAL, and ADA. The BASIC and PASCAL languages were both designed as teaching languages, but have escaped the classroom and are used in many computer systems. The BASIC language is probably the easiest of all to learn. Some estimates indicate that the BASIC language is used in the personal computer for 80 percent of the programs written by users. Recently, a new version of BASIC, VISUAL BASIC, has made programming in the WINDOWS environment easier. The VISUAL BASIC language may eventually supplant C/C++ and PASCAL.

In the scientific community, C/C++ and (occasionally) PASCAL appear as control programs. Both languages, especially C/C++, allow the programmer almost complete control over the programming environment and

computer system. In many cases, C/C++ is replacing some of the low-level, machine control software normally reserved for assembly language. Even so, assembly language still plays an important role in programming. Most video games written for the personal computer are written almost exclusively in assembly language. Assembly language is also interspersed with C/C++ and PASCAL to perform machine control functions efficiently.

The ADA language is used heavily by the Department of Defense. The ADA language was named in honor of Augusta Ada Byron, Countess of Lovelace. The Countess worked with Charles Babbage in the early 1800s in the development of his Analytical Engine.

The Microprocessor Age

The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor—a programmable controller on a chip. It addressed a mere 4096 4-bit wide memory locations. (A **bit** is a binary digit with a value of one or zero. A 4-bit wide memory location is often called a **nibble.**) The 4004 instruction set contained only 45 instructions. It was fabricated with the then-current state-of-the-art P-channel MOSFET technology that only allowed it to execute instructions at the slow rate of 50 KIPs **(kilo-instructions per second).** This was slow when compared to the 100,000 instructions executed per second by the 30-ton ENIAC computer in 1946. The main difference was that the 4004 weighed much less than an ounce.

At first, applications abounded for this device. The 4-bit microprocessor debuted in early video game systems and small microprocessor-based control systems. One such early video game, a shuffleboard game, was produced by Balley. The main problems with this early microprocessor were its speed, word width, and memory size. The evolution of the 4-bit microprocessor ended when Intel released the 4040, an updated version of the earlier 4004. The 4040 operated at a higher speed, although it lacked improvements in word width and memory size. Other companies, particularly Texas Instruments (TMS-1000), also produced 4-bit microprocessors. The 4-bit microprocessor still survives in low-end applications such as microwave ovens and small control systems, and is still available from some microprocessor manufacturers. Most calculators are still based on 4-bit microprocessors that process 4-bit BCD **(binary-coded decimal)** codes.

Later in 1971, realizing that the microprocessor was a commercially viable product, Intel Corporation released the 8008—an extended 8-bit version of the 4004 microprocessor. The 8008 addressed an expanded memory size (16K bytes) and contained additional instructions (a total of 48) that provided an opportunity for its application in more advanced systems. (A **byte** is generally an 8-bit wide binary number and a **K** is 1024. Often, memory size is specified in K bytes.)

As engineers developed more demanding uses for the 8008 microprocessor, they discovered that its somewhat small memory size, slow speed, and instruction set limited its usefulness. Intel recognized these limitations and introduced the 8080 microprocessor in 1973—the first of the modern 8-bit microprocessors. About six months after Intel released the 8080 microprocessor, Motorola Corporation introduced its MC6800 microprocessor. The floodgates opened and the 8080—and, to a lesser degree, the MC6800—ushered in the age of the microprocessor. Soon, other companies began to introduce their own versions of the 8-bit microprocessor. Table 1–1 lists several of these early microprocessors and their manufacturers. Of these early microprocessor producers, only Intel and

TABLE 1–1 Early 8-bit microprocessors.

Manufacturer	*Part Number*
Fairchild	F-8
Intel	8080
MOS Technology	6502
Motorola	MC6800
National Semiconductor	IMP-8
Rockwell International	PPS-8
Zilog	Z-8

Motorola continue successfully to create newer and improved versions of the microprocessor. Zilog still manufactures microprocessors, but has remained in the background, concentrating on microcontrollers and embedded controllers instead of general-purpose microprocessors. Rockwell has all but abandoned microprocessor development in favor of modem circuitry. Motorola has declined from having nearly 50 percent share of the microprocessor market to a much smaller share.

What Was Special about the 8080? Not only could the 8080 address more memory and execute additional instructions, but it executed them 10 times faster than the 8008. An addition that took 20 μs (50,000 instructions per second) on an 8008-based system required only 2.0 μs (500,000 instructions per second) on an 8080-based system. Also, the 8080 was compatible with TTL (transistor-transistor logic), whereas the 8008 was not directly compatible. This made interfacing much easier and less expensive. The 8080 also addressed four times more memory (64K bytes) than the 8008 (16K bytes). These improvements are responsible for ushering in the era of the 8080 and the continuing saga of the microprocessor. Incidentally, the first personal computer, the MITS Altair 8800, was released in 1974. (Note that the number 8800 was probably chosen to avoid copyright violations with Intel.) The BASIC language interpreter, written for the Altair 8800 computer, was developed by Bill Gates, the founder of Microsoft Corporation. The assembler program for the Altair 8800 was written by Digital Research Corporation, which once produced DR-DOS for the personal computer.

The 8085 Microprocessor. In 1977, Intel Corporation introduced an updated version of the 8080—the 8085. This was to be the last 8-bit, general-purpose microprocessor developed by Intel. Although only slightly more advanced than an 8080 microprocessor, the 8085 executed software at an even higher speed. An addition that took 2.0 μs (500,000 instructions per second) on the 8080 required only 1.3 μs (769,230 instructions per second) on the 8085. The main advantages of the 8085 were its internal clock generator, internal system controller, and higher clock frequency. This higher level of component integration reduced the 8085's cost and increased its usefulness. Intel has managed to sell well over 100 million copies of the 8085 microprocessor, its most successful 8-bit, general-purpose microprocessor. Because the 8085 is also manufactured (second-sourced) by many other companies, there are over 200 million of these microprocessors in existence. Applications that contain the 8085 will likely continue to be popular well into the future. Another company that sold 500 million 8-bit microprocessors is Zilog Corporation, which produced the Z-80 microprocessor. The Z-80 is machine language code-compatible with the 8085, which means that there are over 700 million microprocessors that execute 8085/Z-80 compatible code!

The Modern Microprocessor

In 1978, Intel released the 8086 microprocessor; a year or so later, it released the 8088. Both devices are 16-bit microprocessors, which executed instructions in as little as 400 ns (2.5 MIPs, or 2.5 millions of instructions per second). This represented a major improvement over the execution speed of the 8085. In addition, the 8086 and 8088 addressed 1M bytes of memory, which was 16 times more memory than the 8085. (A **1M byte memory** contains 1024K byte-sized memory locations, or 1,048,576 bytes.) This higher execution speed and larger memory size allowed the 8086 and 8088 to replace smaller minicomputers in many applications. One other feature found in the 8086/8088 was a small 4- or 6-byte instruction cache or queue that prefetched a few instructions before they were executed. The queue sped the operation of many sequences of instructions and proved to be the basis for the much larger instruction caches found in modern microprocessors.

The increased memory size and additional instructions in the 8086 and 8088 have led to many sophisticated applications for microprocessors. Improvements to the instruction set included a multiply-and-divide instruction, which was missing on earlier microprocessors. In addition, the number of instructions increased from 45 on the 4004, to 246 on the 8085, to well over 20,000 variations on the 8086 and 8088 microprocessors. Note that these microprocessors were called CISC (**complex instruction set computers**) because of the number and complexity of instructions. The additional instructions eased the task of developing efficient and sophisticated applications, even though the number of instructions was at first overwhelming and time-consuming to learn. The 16-bit microprocessor also provided more internal register storage space than the 8-bit microprocessor. The additional registers allowed software to be written more efficiently.

The 16-bit microprocessor evolved mainly because of the need for larger memory systems. The popularity of the Intel family was ensured in 1981, when IBM Corporation decided to use the 8088 microprocessor in its personal computer. Applications such as spreadsheets, word processors, spelling checkers, and computer-based thesauruses were memory-intensive and required more than the 64K bytes of memory found in 8-bit microprocessors to execute efficiently. The 16-bit 8086 and 8088 provided 1M bytes of memory for these applications. Soon, even the 1M byte memory system proved limiting for large databases and other applications. This led Intel to introduce the 80286 microprocessor, an updated 8086, in 1983.

The 80286 Microprocessor. The 80286 microprocessor (also a 16-bit architecture microprocessor) was almost identical to the 8086 and 8088, except it addressed a 16M byte memory system instead of a 1M byte system. The instruction set of the 80286 was almost identical to the 8086 and 8088, except for a few additional instructions that managed the extra 15M bytes of memory. The clock speed of the 80286 was increased, so it executed some instructions in as little as 250 ns (4.0 MIPs) with the original release 8.0 MHz version. Some changes also occurred in the internal execution of the instructions, which led to an eight-fold increase in speed for many instructions when compared to 8086/8088 instructions.

The 32-bit Microprocessor. Applications began to demand faster microprocessor speeds, more memory, and wider data paths. This led to the arrival of the 80386 in 1986, by Intel Corporation. The 80386 represented a major overhaul of the 16-bit 8086–80286 architecture. The 80386 was Intel's first practical 32-bit microprocessor that contained a 32-bit data bus and a 32-bit memory address. (Note that Intel produced an earlier, although unsuccessful, 32-bit microprocessor called the iapx-432.) Through these 32-bit buses, the 80386 addressed up to 4G bytes of memory. (**1G of memory** contains 1024M, or 1,073,741,824 locations.) A 4G byte memory can store an astounding 1,000,000 typewritten, double-spaced pages of ASCII text data. The 80386 was available in a few modified versions such as the 80386SX, which addressed 16M bytes of memory through a 16-bit data and 24-bit address bus, and the 80386SL/80386SLC, which addressed 32M bytes of memory through a 16-bit data and 25-bit address bus. An 80386SLC version contained an internal cache memory that allowed it to process data at even higher rates. In 1995, Intel released the 80386EX microprocessor. The 80386EX microprocessor is called an **embedded PC** because it contains all the components of the AT class personal computer on a single integrated circuit. The 80386EX also contains 24 lines for input/output data, a 26-bit address bus, a 16-bit data bus, a DRAM refresh controller, and programmable chip selection logic.

Applications that require higher microprocessor speeds and large memory systems include software systems that use a GUI, or **graphical user interface.** Modern graphical displays often contain 256,000 or more picture elements (**pixels,** or **pels**). The least sophisticated VGA (**variable graphics array**) video display has a resolution of 640 pixels per scanning line with 480 scanning lines. To display one screen of information, each picture element must be changed, which requires a high-speed microprocessor. Many new software packages use this type of video interface. These GUI-based packages require high microprocessor speeds and accelerated video adapters for quick and efficient manipulation of video text and graphical data. The most striking system, which requires high-speed computing for its graphical display interface, is Microsoft Corporation's Windows.[3] We often call a GUI a WYSIWYG (**what you see is what you get**) display.

The 32-bit microprocessor is needed because of the size of its data bus, which transfers real (single-precision floating-point) numbers that require 32-bit wide memory. In order to efficiently process 32-bit real numbers, the microprocessor must efficiently pass them between itself and memory. If the numbers pass through an 8-bit data bus, it takes four read or write cycles; when passed through a 32-bit data bus, however, only one read or write cycle is required. This significantly increases the speed of any program that manipulates real numbers. Most high-level languages, spreadsheets, and database management systems use real numbers for data storage. Real numbers

[3]Windows is a registered trademark of Microsoft Corporation and is currently available as Windows 95, Windows 98, Windows 2000, Windows ME, and Windows XP.

are also used in graphical design packages that use vectors to plot images on the video screen. These include such CAD **(computer aided drafting/design)** systems as AUTOCAD, ORCAD, and so forth.

Besides providing higher clocking speeds, the 80386 included a memory management unit that allowed memory resources to be allocated and managed by the operating system. Earlier microprocessors left memory management completely to the software. The 80386 included hardware circuitry for memory management and memory assignment, which improved its efficiency and reduced software overhead.

The instruction set of the 80386 microprocessor was upward-compatible with the earlier 8086, 8088, and 80286 microprocessors. Additional instructions referenced the 32-bit registers and managed the memory system. Note that memory management instructions and techniques used by the 80286 were also compatible with the 80386 microprocessor. These features allowed older, 16-bit software to operate on the 80386 microprocessor.

The 80486 Microprocessor. In 1989, Intel released the 80486 microprocessor, which incorporated an 80386-like microprocessor, an 80387-like numeric coprocessor, and an 8K byte cache memory system into one integrated package. Although the 80486 microprocessor was not radically different from the 80386, it did include one substantial change. The internal structure of the 80486 was modified from the 80386 so that about half of its instructions executed in one clock instead of two clocks. Because the 80486 was available in a 50 MHz version, about half of the instructions executed in 25ns (50 MIPs). The average speed improvement for a typical mix of instructions was about 50 percent over the 80386 that operated at the same clock speed. Later versions of the 80486 executed instructions at even higher speeds with a 66 MHz double-clocked version (80486DX2). The double-clocked 66 MHz version executed instructions at the rate of 66 MHz, with memory transfers executing at the rate of 33 MHz. (This is why it was called a double-clocked microprocessor.) A triple-clocked version from Intel, the 80486DX4, improved the internal execution speed to 100 MHz with memory transfers at 33 MHz. Note that the 80486DX4 microprocessor executed instructions at about the same speed as the 60 MHz Pentium. It also contained an expanded 16K byte cache in place of the standard 8K byte cache found on earlier 80486 microprocessors. Advanced Micro Devices (AMD) has produced a triple-clocked version that runs with a bus speed of 40 MHz and a clock speed of 120 MHz. The future promises to bring microprocessors that internally execute instructions at rates of up to 1 GHz or higher.

Other versions of the 80486 were called Overdrive[4] processors. The Overdrive processor was actually a double-clocked version of the 80486DX that replaced an 80486SX or slower-speed 80486DX. When the Overdrive processor was plugged into its socket, it disabled or replaced the 80486SX or 80486DX, and functioned as a doubled-clocked version of the microprocessor. For example, if an 80486SX, operating at 25 MHz, was replaced with an Overdrive microprocessor, it functioned as an 80486DX2 50 MHz microprocessor using a memory transfer rate of 25 MHz.

Table 1–2 lists many microprocessors produced by Intel and Motorola with information about their word and memory sizes. Other companies produce microprocessors, but none have attained the success of Intel and, to a lesser degree, Motorola.

The Pentium Microprocessor. The Pentium, introduced in 1993, was similar to the 80386 and 80486 microprocessors. This microprocessor was originally labeled the P5 or 80586, but Intel decided not to use a number because it appeared to be impossible to copyright a number. The two introductory versions of the Pentium operated with a clocking frequency of 60 MHz and 66 MHz, and a speed of 110 MIPs, with a higher-frequency 100 MHz one and one-half clocked version that operated at 150 MIPs. The double-clocked Pentium, operating at 120 MHz and 133 MHz, was also available, as were higher-speed versions. (The fastest version produced by Intel is the 233 MHz Pentium, which is a three and one-half clocked version.) Another difference was that the cache size was increased to 16K bytes from the 8K cache found in the basic version of the 80486. The Pentium contained an 8K byte instruction cache and an 8K byte data cache, which allowed a program that transfers a large amount of memory data to still benefit from a cache. The memory system contained up to 4G bytes, with the data bus width increased from the 32 bits found in the 80386 and 80486 to a full 64 bits. The data bus transfer speed was either 60 MHz or 66 MHz, depending on the version of the Pentium. (Recall that the bus speed of the 80486 was 33 MHz.) This wider data bus

[4]Overdrive is a registered trademark of Intel Corporation.

TABLE 1–2 Many modern Intel and Motorola microprocessors.

Manufacturer	*Part*	*Data Bus Width*	*Memory Size*
Intel	8048	8	2K internal
	8051	8	8K internal
	8085A	8	64K
	8086	16	1M
	8088	8	1M
	8096	16	8K internal
	80186	16	1M
	80188	8	1M
	80251	8	16K internal
	80286	16	16M
	80386EX	16	64M
	80386DX	32	4G
	80386SL	16	32M
	80386SLC	16	32M + 1K cache
	80386SX	16	16M
	80486DX/DX2	32	4G + 8K cache
	80486SX	32	4G + 8K cache
	80486DX4	32	4G + 16K cache
	Pentium	64	4G + 16K cache
	Pentium Overdrive (P24T) (replaces 80486)	32	4G + 16K cache
	Pentium Pro processor	64	64G + 16K L1 cache + 256K L2 cache
	Pentium II	64	64G + 32K L1 cache + 512K L2 cache
	Pentium II Xeon	64	64G + 32K L1 cache + 512K or 1M L2 cache
	Pentium III, Pentium 4	64	64G + 32K L1 cache + 256K L2 cache
Motorola	6800	8	64K
	6805	8	2K
	6809	8	64K
	68000	16	16M
	68008Q	8	1M
	68008D	8	4M
	68010	16	16M
	68020	32	4G
	68030	32	4G + 256 cache
	68040	32	4G + 8K cache
	68050	32	Proposed, but never released
	68060	64	4G + 16K cache
	PowerPC	64	4G + 32K cache

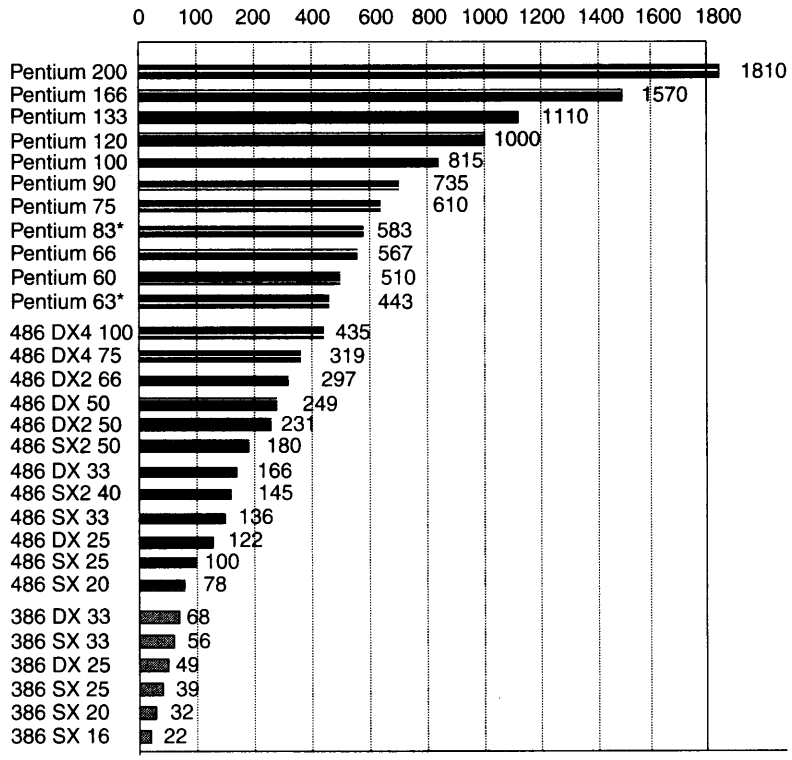


FIGURE 1–2 The Intel iCOMP index.

Note: * = Pentium OverDrive, the first part of the scale is not linear, and the 166 MHz and 200 MHz are MMX technology

width accommodated double-precision floating-point numbers used for modern high-speed, vector-generated graphical displays. These higher bus speeds should allow virtual reality software to operate at more realistic rates on current and future Pentium-based platforms. The widened data bus and higher execution speed of the Pentium allow full-frame video displays to operate at scan rates of 30 Hz or higher—comparable to commercial television. Recent versions of the Pentium also included additional instructions, called multimedia extensions, or MMX instructions. Although Intel hoped that the MMX instructions would be widely used, it appears that few software companies have used them.

Recently, Intel released the long-awaited Pentium OverDrive (P24T) for older 80486 systems that operate at either 63 MHz or 83 MHz clock. The 63 MHz version upgrades older 80486DX2 50 MHz systems; the 83 MHz version upgrades the 80486DX2 66 MHz systems. The upgraded 83 MHz system performs at a rate somewhere between a 66 MHz Pentium and a 75 MHz Pentium. If older VESA local bus video and disk-caching controllers seem too expensive to toss out, the Pentium OverDrive represents an ideal upgrade path from the 80486 to the Pentium.

Probably the most ingenious feature of the Pentium is its dual integer processors. The Pentium executes two instructions, which are not dependent on each other, simultaneously because it contains two independent internal integer processors called superscaler technology. This allows the Pentium to often execute two instructions per clocking period. Another feature that enhances performance is a jump prediction technology that speeds the execution of programs that include loops. As with the 80486, the Pentium also employs an internal floating-point coprocessor to handle floating-point data, albeit at five times the speed improvement. These features portend continued success for the Intel family of microprocessors. They also may allow the Pentium to replace some of the RISC (**reduced instruction set computer**) machines that currently execute one instruction per clock. Note that some newer RISC processors execute more than one instruction per clock through the introduction of superscaler technology. Motorola, Apple, and IBM have recently produced the PowerPC, a RISC microprocessor that has two

integer units and one floating-point unit. The PowerPC certainly boosts the performance of the Apple Macintosh[5], but at present is slow to emulate the Intel family of microprocessors. Tests indicate that the current emulation software executes DOS and Windows applications at speed slower than the 80486SX 25 MHz microprocessor. Because of this, the Intel family should survive for many years in personal computer systems. Note that there are currently 6 million Apple Macintosh[5] systems and well over 260 million personal computers based on Intel microprocessors. In 1998, reports stated that 96 percent of all PCs were shipped with the Windows operating system.

In order to compare the speeds of various microprocessors, Intel devised the iCOMP-rating index. This index is a composite of SPEC92, ZD Bench, and Power Meter. The iCOMP1 rating was used to rate the speed of all Intel microprocessors through the Pentium. Figure 1–2 shows the relative speeds of the 80386DX 25 MHz version at the low end to the Pentium 233 MHz version at the high end of the spectrum.

Since the release of the Pentium Pro and Pentium II, Intel has switched to the iCOMP2 index, which is scaled by a factor of 10 from the iCOMP1 index. A microprocessor with an index of 1000 using iCOMP1 is rated as 100 using iCOMP2. Another difference is the benchmarks used for the scores. Figure 1–3 shows the iCOMP2 index listing the Pentium II at speed up to 450 MHz.

Pentium Pro Processor. A recent entry from Intel is the Pentium Pro processor, formerly code-named the P6 microprocessor. The Pentium Pro processor contains 21 million transistors, 3 integer units, as well as a floating-point unit to increase the performance of most software. The basic clock frequency was 150 MHz and 166 MHz in the initial offering made available in late 1995. In addition to the internal 16K level-one (L1) cache (8K for data and 8K for instructions), the Pentium Pro processor also contains a 256K level-two (L2) cache. One other significant change is that the Pentium Pro processor uses three execution engines, so it can execute up to three instructions at a time, which can conflict and still execute in parallel. This represents a change from the Pentium, which executes two instructions simultaneously as long as they do not conflict. The Pentium Pro microprocessor is optimized to efficiently execute 32-bit code and is used in servers. Pentium Pro can address either 4G Byte or 64G Byte memory systems. For accessing 64G Byte memory Pentium pro can be configured of with 36 bit address lines.

1–2 NUMBER SYSTEMS

A sound working knowledge of Binary, Decimal and Hexadecimal number system is required for using Microprocessors. Those not familiar with these will find this section useful for Interconversion between Decimal, Binary, Hexadecimal number systems.

Digits

Before numbers are converted from one number base to another, the digits of a number system must be understood. Early in our education, we learned that a decimal, or base 10, number was constructed with 10 digits: 0 through 9. The first digit in any numbering system is always a zero. For example, a base 8 **(octal)** number contains 8 digits: 0 through 7; a base 2 **(binary)** number contains 2 digits: 0 and 1. If the base of a number exceeds 10, the additional digits use the letters of the alphabet, beginning with an A. For example, a base 12 number contains 12 digits: 0 through 9, followed by A for 10 and B for 11. Note that a base 10 number does not contain a *10* digit, just as a base 8 number does not contain an *8* digit. The most common numbering systems used with computers are decimal, binary, and hexadecimal (base 16). (Many years ago octal numbers were popular.) Each system is described and used in this section of the chapter.

[5]Macintosh is a registered trademark of Apple Computer Corporation.

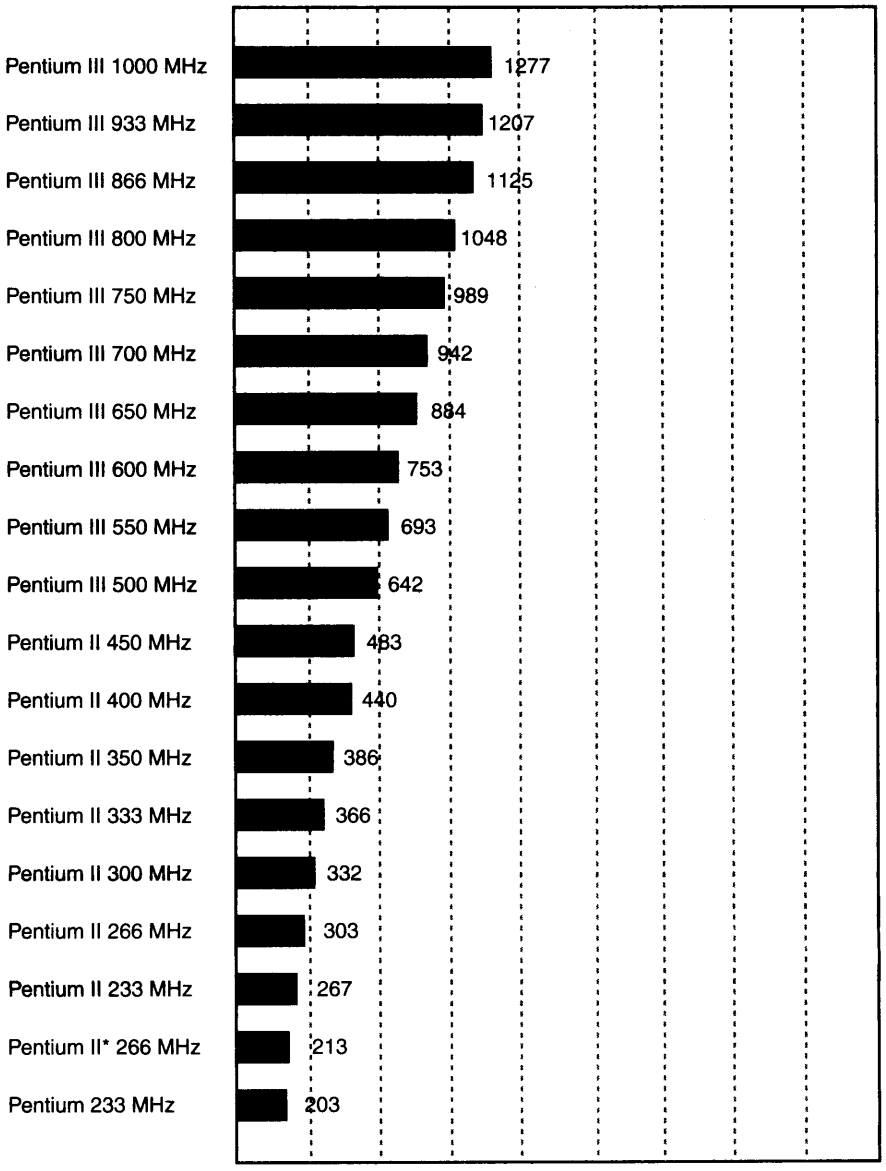


FIGURE 1–3 The Intel iCOMP2 index.

Positional Notation

Once the digits of a number system are understood, larger numbers are constructed by using positional notation. In grade school, we learned that the position to the left of the units position was the tens position, the position to the left of the tens position was the hundreds position, and so forth. (An example is the decimal number 132: This

number has 1 hundred, 3 tens, and 2 units.) What probably was not learned was the exponential value of each position: The units position has a weight of 10^0, or 1; the tens position has weight of 10^1, or 10; and the hundreds position has a weight of 10^2, or 100. The exponential powers of the positions are critical for understanding numbers in other numbering systems. The position to the left of the radix (**number base**) point, called a **decimal point** only in the decimal system, is always the units position in any number system. For example, the position to the left of the binary point is always 2^0, or 1; the position to the left of the octal point is 8^0, or 1. In any case, any number raised to its zero power is always 1, or the units position.

The position to the left of the units position is always the number base raised to the first power; in a decimal system, this is 10^1, or 10. In a binary system, it is 2^1, or 2; and in an octal system, it is 8^1, or 8. Therefore, an 11 decimal has a different value from an 11 binary. The 11 decimal is composed of 1 ten plus 1 unit, and has a value of 11 units; while the binary number 11 is composed of 1 two plus 1 unit, for a value of 3 decimal units. The 11 octal has a value of 9 units.

In the decimal system, positions to the right of the decimal point have negative powers. The first digit to the right of the decimal point has a value of 10^{-1}, or 0.1. In the binary system, the first digit to the right of the binary point has a value of 2^{-1}, or 0.5. In general, the principles that apply to decimal numbers also apply to numbers in any other number system.

Example 1–1 shows a 110.101 in binary (often written as 110.101_2). It also shows the power and weight or value of each digit position. To convert a binary number to decimal, add the weights of each digit to form its decimal equivalent. The 110.101_2 is equivalent to a 6.625 in decimal (4 + 2 + 0.5 + 0.125). Notice that this is the sum of 2^2 (or 4) plus 2^1 (or 2), but 2^0 (or 1) is not added because there are no digits under this position. The fraction part is composed of 2^{-1} (0.5) plus 2^{-3} (or .125), but there is no digit under the 2^{-2} (or .25).

EXAMPLE 1–1

Power	2^2		2^1		2^0		2^{-1}		2^{-2}		2^{-3}				
Weight	4		2		1		0.5		0.25		.125				
Number	1		1		0	.	1		0		1				
Numeric Value	4	+	2	+	0	+	0.5	+	0	+	.125	=	6.625		

Suppose that the conversion technique is applied to a base 6 number, such as 25.2_6. Example 1–2 shows this number placed under the powers and weights of each position. In the example, there is a 2 under 6^1, which has a value of 12_{10} (2 × 6), and a 5 under 6^0, which has a value of 5 (5 × 1). The whole number portion has a decimal value of 12 + 5, or 17. The number to the right of the hex point is a 2 under 6^{-1}, which has a value of .333 (2 × .167). The number 25.2_6, therefore, has a value of 17.333.

EXAMPLE 1–2

Power	6^1		6^0		6^{-1}			
Weight	6		1		.167			
Number	2		5	.	2			
Numeric Value	12	+	5	+	.333	=	17.333	

Conversion to Decimal

The prior examples have shown that to convert from any number base to decimal, determine the weights or values of each position of the number, and then sum the weights to form the decimal equivalent. Suppose that a 125.7_8 octal is converted to decimal. To accomplish this conversion, first write down the weights of each position of the number. This appears in Example 1–3 The value of 125.7_8 is 85.875 decimal, or 1 × 64 plus 2 × 8 plus 5 × 1 plus 7 × 125.

EXAMPLE 1–3

Power	8^2	8^1	8^0	8^{-1}		
Weight	64	8	1	.125		
Number	1	2	5.	7		
Numeric Value	64	+ 16	+ 5	+ .875	=	85.875

Notice that the weight of the position to the left of the units position is 8. This is 8 times 1. Then notice that the weight of the next position is 64, or 8 times 8. If another position existed, it would be 64 times 8, or 512. To find the weight of the next higher-order position, multiply the weight of the current position by the number base (or 8, in this example). To calculate the weights of position to the right of the radix point, divide by the number base. In the octal system, the position immediately to the right of the octal point is $^1/_8$, or .125. The next position is $^{.125}/_8$, or .015625, which can also be written as $^1/_{64}$. Also note that the number in Example 1–3 can also be written as the decimal number $85^7/_8$.

Example 1–4 shows the binary number 11011.0111 written with the weights and powers of each position. If these weights are summed, the value of the binary number converted to decimal is 27.4375.

EXAMPLE 1–4

Power	2^4		2^3		2^2		2^1		2^0		2^{-1}	2^{-2}		2^{-3}		2^{-4}			
Weight	16		8		4		2		1		0.5	0.25		.125		.0625			
Number	1		1		0		1		1	.	0	1		1		1			
Numeric Value	16	+	8	+	0	+	2	+	1	+	0	+ .25	+	.125	+	.0625	=	27.4375	

It is interesting to note that 2^{-1} is also $^1/_2$, 2^{-2} is $^1/_4$, and so forth. It is also interesting to note that 2^{-4} is $^1/_{16}$, or .0625. The fractional part of this number is $^7/_{16}$ or .4375 decimal. Notice that 0111 is a 7 in binary code for the numerator and the rightmost one is in the $^1/_{16}$ position for the denominator. Other examples: the binary fraction of .101 is $^5/_8$ and the binary fraction of .001101 is $^{13}/_{64}$.

Hexadecimal numbers are often used with computers. A 6A.CH (H for hexadecimal) is illustrated with its weights in Example 1–5. The sum of its digits is 106.75, or $106^3/_4$. The whole number part is represented with 6×16 plus 10 (A) $\times$ 1. The fraction part is 12 (C) as a numerator and 16 (16^{-1}) as the denominator, or $^{12}/_{16}$, which is reduced to $^3/_4$.

EXAMPLE 1–5

Power	16^1		16^0		16^{-1}		
Weight	16		1		.0625		
Number	6		A		C		
Numeric Value	96	+	10	+	.75	=	106.75

Conversion From Decimal

Conversions from decimal to other number systems are more difficult to accomplish than conversion to decimal. To convert the whole number portion of a number to decimal, divide by the radix. To convert the fractional portion, multiply by the radix.

Whole Number Conversion from Decimal. To convert a decimal whole number to another number system, divide by the radix and save the remainders as significant digits of the result. An algorithm for this conversion as is follows:

1. Divide the decimal number by the radix (number base).
2. Save the remainder (first remainder is the least significant digit).
3. Repeat steps 1 and 2 until the quotient is zero.

For example, to convert a 10 decimal to binary, divide it by 2. The result is 5, with a remainder of 0. The first remainder is the units position of the result (in this example, a 0). Next, divide the 5 by 2. The result is 2, with a

remainder of 1. The 1 is the value of the two's (2^1) position. Continue the division until the quotient is a zero. Example 1–6 shows this conversion process. The result is written as 1010_2, from the bottom to the top.

EXAMPLE 1–6

```
2) 10   remainder = 0
 2) 5   remainder = 1
  2) 2  remainder = 0
   2) 1 remainder = 1          result = 1010
     0
```

To convert a 10 decimal into base 8, divide by 8, as shown in Example 1–7. A 10 decimal is a 12 octal.

EXAMPLE 1–7

```
8) 10   remainder = 2
 8) 1   remainder = 1          result = 12
   0
```

Conversion from decimal to hexadecimal is accomplished by dividing by 16. The remainders will range in value from 0 through 15. Any remainder of 10 though 15 is then converted to the letters A through F for the hexadecimal number. Example 1–8 shows the decimal number 109 converted to a 6DH.

EXAMPLE 1–8

```
16) 109   remainder = 13 (D)
 16) 6    remainder = 6          result = 6D
    0
```

Converting from a Decimal Fraction. Conversion from a decimal fraction to another number base is accomplished with multiplication by the radix. For example, to convert a decimal fraction into binary, multiply by 2. After the multiplication, the whole number portion of the result is saved as a significant digit of the result, and the fractional remainder is again multiplied by the radix. When the fraction remainder is zero, multiplication ends. Note that some numbers are never-ending. That is, a zero is never a remainder. An algorithm for conversion from a decimal fraction is as follows:

1. Multiply the decimal fraction by the radix (number base).
2. Save the whole number portion of the result (even if zero) as a digit. Note that the first result is written immediately to the right of the radix point.
3. Repeat steps 1 and 2, using the fractional part of step 2 until the fractional part of step 2 is zero.

Suppose that a .125 decimal is converted to binary. This is accomplished with multiplications by 2, as illustrated in Example 1–9. Notice that the multiplication continues until the fractional remainder is zero. The whole number portions are written as the binary fraction (0.001) in this example.

EXAMPLE 1–9

```
  .125
x   2
 0.25 digit is 0

  .25
x  2
 0.5  digit is 0

  .5
x 2
 1.0  digit is 1. The result is written as 0.001 binary
```

This same technique is used to convert a decimal fraction into any number base. Example 1–10 shows the same decimal fraction of .125 from Example 1–9 converted to octal by multiplying with an 8.

EXAMPLE 1–10

```
   .125
x ____8
   1.0 digit is 1. The result is written as 0.1 octal
```

Conversion to a hexadecimal fraction appears in Example 1–11. Here, a decimal .046875 is converted to hexadecimal by multiplying by 16. Note that a .046875 is a 0.0CH.

EXAMPLE 1–11

```
   .046875
x _____16
     0.75 digit is 0

   .75
x __16
 12.0 digit is 12 (C). The result is written as 0.0C hexadecimal
```

Binary-Coded Hexadecimal

Binary-coded hexadecimal (BCH) is used to represent hexadecimal data in binary code. A binary-coded hexadecimal number is a hexadecimal number written so that each digit is represented by a 4-bit binary number. The values for the BCH digits appear in Table 1–3.

Hexadecimal numbers are represented in BCH code by converting each digit to BCH code, with a space between each coded digit. Example 1–12 shows a 2AC converted to BCH code. Note that each BCH digit is separated by a space.

TABLE 1–3 Binary-coded hexadecimal (BCH) code.

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

EXAMPLE 1–12

```
2AC = 0010 1010 1100
```

The purpose of BCH code is to allow a binary version of a hexadecimal number to be written in a form that can easily be converted between BCH and hexadecimal. Example 1–13 shows a BCH coded number converted back to hexadecimal code.

EXAMPLE 1–13

```
1000 0011 1101 . 1110 = 83D.E
```

Complements

At times, data are stored in complement form to represent negative numbers. There are two systems that are used to represent negative data: **radix** and **radix –1** complements. The earliest system was the radix –1 complement, in which each digit of the number is subtracted from the radix –1 to generate the radix –1 complement to represent a negative number.

Example 1–14 shows how the 8-bit binary number 01001100 is one's (radix –1) complemented to represent it as a negative value. Notice that each digit of the number is subtracted from one to generate the radix –1 (one's) complement. In this example, the negative of 01001100 is 10110011. The same technique can be applied to any number system, as illustrated in Example 1–15, in which the fifteen's (radix –1) complement of a 5CD hexadecimal is computed by subtracting each digit from a fifteen.

EXAMPLE 1–14

```
  1 1 1 1  1 1 1 1
- 0 1 0 0  1 1 0 0
  ---------------
  1 0 1 1  0 0 1 1
```

EXAMPLE 1–15

```
  15 15 15
-  5  C  D
  ---------
   A  3  2
```

Today, the radix –1 complement is not used by itself; it is used as a step for finding the radix complement. The radix complement is used to represent negative numbers in modern computer systems. (The radix –1 complement was used in the early days of computer technology.) The main problem with the radix –1 complement is that a negative or a positive zero exists; in the radix complement system, only a positive zero can exist.

To form the radix complement, first find the radix –1 complement, and then add a one to the result. Example 1–16 shows how the number 0100 1000 is converted to a negative value by two's (radix) complementing it.

EXAMPLE 1–16

```
  1 1 1 1  1 1 1 1
- 0 1 0 0  1 0 0 0
  ---------------
  1 0 1 1  0 1 1 1   (one's complement)
+ 0 0 0 0  0 0 0 1
  ---------------
  1 0 1 1  1 0 0 0   (two's complement)
```

To prove that a 0100 1000 is the inverse (negative) of a 1011 0111, add the two together to form an 8-digit result. The ninth digit is dropped and the result is zero because a 0100 1000 is a positive 72, while a 1011 0111 is a negative 72. The same technique applies to any number system. Example 1–17 shows how the inverse of a 345 hexadecimal is found by first fifteen's complementing the number, and then by adding one to the result to form the

sixteen's complement. As before, if the original 3-digit number 345 is added to the inverse of CBB, the result is a 3-digit 000. As before, the fourth bit (carry) is dropped. This proves that 345 is the inverse of CBB. Additional information about one's and two's complements is presented with signed numbers in the next section of the text.

EXAMPLE 1–17

```
  15 15 15
-  3  4  5
   C  B  A   (fifteen's complement)
+  0  0  1
   C  B  B   (sixteen's complement)
```

1–3 COMPUTER DATA FORMATS

Successful programming requires a precise understanding of data formats. In this section, many common computer data formats are described as they are used with the Intel family of microprocessors. Commonly, data appear as ASCII, BCD, signed and unsigned integers, and floating-point numbers (real numbers). Other forms are available, but are not presented here because they are not commonly found.

ASCII Data

ASCII **(American Standard Code for Information Interchange)** data represent alphanumeric characters in the memory of a computer system (see Table 1–4). The standard ASCII code is a 7-bit code, with the eighth and most significant bit used to hold parity in some systems. If ASCII data are used with a printer, the most significant bits are a 0 for alphanumeric printing and 1 for graphics printing. In the personal computer, an extended ASCII character set is selected by placing a logic 1 in the left-most bit. Table 1–5 shows the extended ASCII character set, using code 80H–FFH. The extended ASCII characters store some foreign letters and punctuation, Greek characters, mathematical characters, box-drawing characters, and other special characters. Note that extended characters can vary from one printer to another. The list provided is designed to be use with the IBM ProPrinter[6] which also matches the special character set found with some word processors.

The ASCII control characters, also listed in Table 1–4, perform control functions in a computer system, including clear screen, backspace, line feed, and so on. To enter the control codes through the computer keyboard,

TABLE 1–4 ASCII code

	Second															
	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
First																
0X	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1X	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EMS	SUB	ESC	FS	GS	RS	US
2X	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⋮⋮⋮

[6]The IBM ProPrinter is a product of IBM Corporation.

TABLE 1–5 Extended ASCII code, as printed by the IBM ProPrinter.

First	Second															
	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X		☺	☻	♥	♦	♣	♠	•	◘	○	■	♂	♀	♪	♫	☼
1X	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
8X	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9X	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
AX	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
BX	▒	▒	▒	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
CX	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
DX	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	▒	▒	▒	▒	▒
EX	α	β	Γ	π	Σ	σ	µ	γ	Φ	Θ	Ω	δ	∞	φ	∈	∩
FX	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	n	2	■	

hold down the Control key while typing a letter. To obtain the control code 01H, type a Control-A; a 02H is obtained by a Control-B, etc. Note that the control codes appear on the screen, from the DOS prompt, as ^A for Control-A, ^B for Control-B, and so forth. Also note that the carriage return code (CR) is the Enter key on most modern keyboards. The purpose of CR is to return the cursor or print-head to the left margin. Another code that appears in many programs is the line feed code (LF), which moves the cursor down one line.

To use Table 1–4 or 1–5 for converting alphanumeric or control characters into ASCII characters, first locate the alphanumeric code for conversion. Next, find the first digit of the hexadecimal ASCII code. Then find the second digit. For example, the capital letter A is ASCII code 41H, and the lowercase letter a is ASCII code 61H. Many Windows-based applications use the *Unicode* system to store alphanumeric data. This system stores each character as 16-bit data. The codes 0000H–00FFH are the same as standard ASCII code. The remaining codes 0100H– FFFFH are used to store all special characters from all world-wide character sets. This allows software written for the Windows environment to be used in any country in the world.

ASCII data are most often stored in memory by using a special directive to the assembler program called define byte(s), or DB. (The assembler is a program that is used to program a computer in its native binary machine language.) An alternative to DB is the word BYTE. The DB and BYTE directives, and several examples of their usage with ASCII-coded character strings, are listed in Example 1–18. Notice how each character string is surrounded by apostrophes (')—never use the quote ("). Also notice that the assembler lists the ASCII-coded value for each character to the left of the character string. To the far left is the hexadecimal memory location where the character string is first stored in the memory system. For example, the character string WHAT is stored beginning at memory address 001D, and the first letter is stored as 57 (W) followed by 68 (H), and so forth.

EXAMPLE 1–18

```
0000  42 61 72 72 79   NAMES  DB   'Barry B. Brey'
      20 42 2E 20 42
      72 65 79
000D  57 68 65 72 65   MESS   DB   'Where can it be?'
      20 63 61 6E 20
      69 74 20 62 65
      3F
```

```
001D  57 68 61 74 20    WHAT    DB    'What is on first.'
      69 73 20 6F 6E
      20 66 69 72 73
      74 2E
```

BCD (Binary-Coded Decimal) Data

Binary-coded decimal (BCD) information is stored in either packed or unpacked forms. **Packed BCD** data are stored as two digits per byte and **unpacked BCD** data are stored as one digit per byte. The range of a BCD digit extends from 00002 to 10012, or 0–9 decimal. Unpacked BCD data are returned from a keypad or keyboard. Packed BCD data are used for some of the instructions included for BCD addition and subtraction in the instruction set of the microprocessor.

Table 1–6 shows some decimal numbers converted to both the packed and unpacked BCD forms. Applications that require BCD data are point-of-sales terminals and almost any device that performs a minimal amount of simple arithmetic. If a system requires complex arithmetic, BCD data are seldom used because there is no simple and efficient method of performing complex BCD arithmetic.

Example 1–19 shows how to use the assembler to define both packed and unpacked BCD data. In all cases, the convention of storing the least-significant data first is followed. This means that to store an 83 into memory, the 3 is stored first, and then followed by the 8. Also note that with packed BCD data, the letter H (hexadecimal) follows the number to ensure that the assembler stores the BCD value rather than a decimal value for packed BCD data. Notice how the numbers are stored in memory as unpacked, one digit per byte; or packed, as two digits per byte.

EXAMPLE 1–19

```
                    ;Unpacked BCD data (least-significant data first)
                    ;
0000  03 04 05      NUMB1  DB     3,4,5          ;defines the number 543
0003  07 08         NUMB2  DB     7,8            ;defines the number 87
                    ;
                    ;Packed BCD data (least-significant data first)
                    ;
0005  37 34         NUMB3  DB     37H,34H        ;defines the number 3437
0007  03 45         NUMB4  DB     3,45H          ;defines the number 4503
```

Byte-Sized Data

Byte-sized data are stored as *unsigned* and *signed* integers. Figure 1–4 illustrates both the unsigned and signed forms of the byte-sized integer. The difference in these forms is the weight of the leftmost bit position. Its value is 128 for the unsigned integer and minus 128 for the signed integer. In the signed integer format, the leftmost bit represents the sign bit of the number, as well as a weight of minus 128. For example, an 80H represents a value of 128 as an unsigned number; as a signed number, it represents a value of minus 128. Unsigned integers range in value from 00H–FFH (0–255). Signed integers range in value from –128 to 0 to +127.

TABLE 1–6 Packed and unpacked BCD data.

Decimal	*Packed*	*Unpacked*
12	0001 0010	0000 0001 0000 0010
623	0000 0110 0010 0011	0000 0110 0000 0010 0000 0011
910	0000 1001 0001 0000	0000 1001 0000 0001 0000 0000

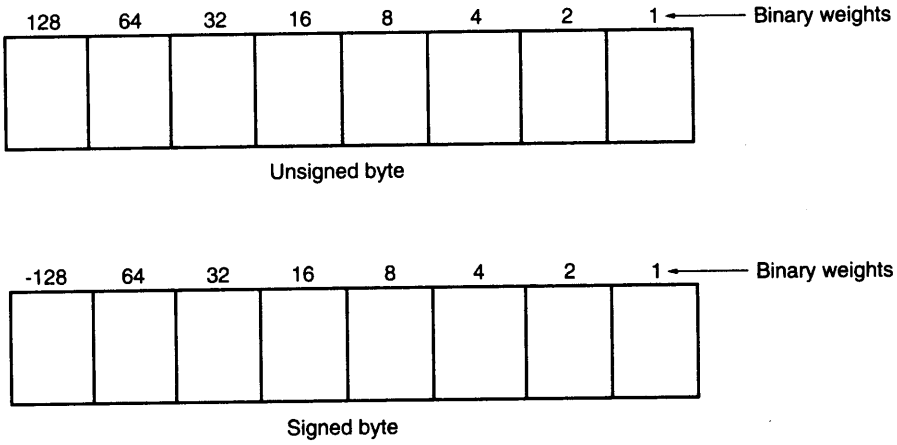


FIGURE 1–4 The unsigned and signed bytes illustrating the weights of each binary-bit position.

Although negative signed numbers are represented in this way, they are stored in the two's complement form. The method of evaluating a signed number by using the weights of each bit position is much easier than the act of two's complementing a number to find its value. This is especially true in the world of calculators designed for programmers.

Whenever a number is two's complemented, its sign changes from negative to positive or positive to negative. For example, the number 00001000 is a +8. Its negative value (–8) is found by two's complementing the +8. To form a two's complement, first one's complement the number. To one's complement a number, invert each bit of a number from zero to one or from one to zero. Once the one's complement is formed, the two's complement is found by adding a one to the one's complement. Example 1–20 shows how numbers are two's complemented using this technique.

EXAMPLE 1–20

```
+8 = 00001000
     11110111 (one's complement)
+ _________1
-8 = 11111000 (two's complement)
```

Another, and probably simpler, technique for two's complementing a number starts with the rightmost digit. Start writing down the number from right to left. Write the number exactly as it appears until the first one. Write down the first one, and then invert of complement all remaining ones to its left. Example 1–21 shows this technique with the same number as in Example 1–20.

EXAMPLE 1–21

```
+8 = 00001000
         1000 (write number to first 1)
     1111     (invert the remaining bits)
-8 = 11111000
```

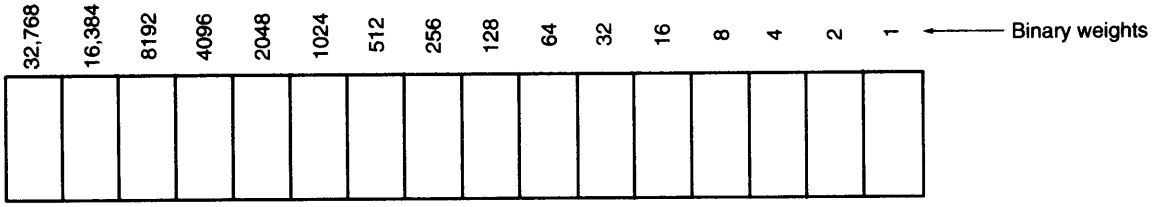
To store 8-bit data in memory using the assembler program, use the DB directive as in prior examples. Example 1–22 lists many forms of 8-bit numbers stored in memory using the assembler program. Notice in the example that a hexadecimal number is defined with the letter H following the number, and that a decimal number is written as is, without anything special.

EXAMPLE 1–22

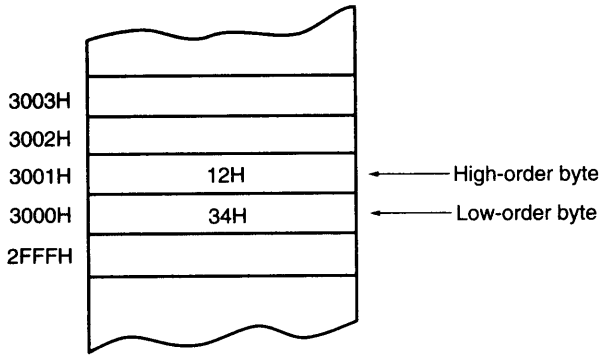
```
                    ;Unsigned byte-sized data
                    ;
0000    FE          DATA1   DB      254             ;define 254 decimal
0001    87          DATA2   DB      87H             ;define 87 hexadecimal
0002    47          DATA3   DB      71              ;define 71 decimal
                    ;
                    ;Signed byte-sized data
                    ;
0003    9C          DATA4   DB      -100            ;define a -100 decimal
0004    64          DATA5   DB      +100            ;define a +100 decimal
0005    FF          DATA6   DB      -1              ;define a -1 decimal
0006    38          DATA7   DB      56              ;define a 56 decimal
```

Word-Sized Data

A word (16-bits) is formed with two bytes of data. The least significant byte is always stored in the lowest-numbered memory location, and the most significant byte is stored in the highest. This method of storing a number is called the **little endian** format. An alternate method, not used with the Intel family of microprocessors, is called the **big endian** format. In the big endian format, numbers are stored with the lowest location containing the most significant data. The big endian format is used with the Motorola family of microprocessors. Figure 1–5 (a) shows the weights of each bit position in a word of data, and Figure 1–5 (b) shows how the number 1234H appears when stored in the memory location 3000H and 3001H. The only difference between a signed and an unsigned word is the leftmost bit position. In the unsigned form, the leftmost bit is unsigned; in the signed form, its weight is a –32,768. As with byte-sized signed data, the signed word is in two's complement form when representing a



(a) Unsigned word



(b) The contents of memory location 3000H and 3001H are the word 1234H.

FIGURE 1–5 The storage format for a 16-bit word in (a) a register and (b) two bytes of memory.

negative number. Also, notice that the low-order byte is stored in the lowest-numbered memory location (3000H) and the high-order byte is stored in the highest-numbered location (3001H).

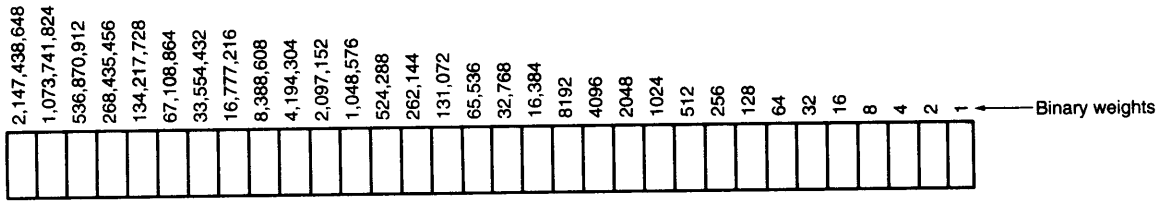
Example 1–23 shows several signed and unsigned word-sized data stored in memory using the assembler program. Notice that the **define word(s) directive,** or DW, causes the assembler to store words in the memory instead of bytes, as in prior examples. The WORD directive can also be used to define a word. Notice that the word *data* is displayed by the assembler in the same form as entered. For example, a 1000H is displayed by the assembler as a 1000. This is for our convenience because the number is actually stored in the memory as 00 10 in two consecutive memory bytes.

EXAMPLE 1–23

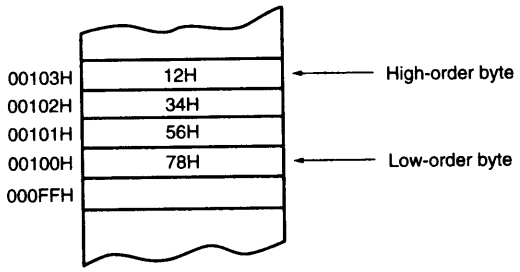
```
                        ;Unsigned word-sized data
                        ;
0000    09F0            DATA1   DW    2544          ;define 2544 decimal
0002    87AC            DATA2   DW    87ACH         ;define 87AC hexadecimal
0004    02C6            DATA3   DW    710           ;define 710 decimal
                        ;
                        ;Signed word-sized data
                        ;
0006    CBA8            DATA4   DW    -13400        ;define a -13400 decimal
0008    00C6            DATA5   DW    +198          ;define a +198 decimal
000A    FFFF            DATA6   DW    -1            ;define a -1 decimal
```

Doubleword-Sized Data

Doubleword-sized data requires four bytes of memory because it is a 32-bit number. Doubleword data appear as a product after a multiplication and also as a dividend before a division. In the 80386 through the Pentium 4, memory and registers are also 32 bits in width. Figure 1–6 shows the form used to store doublewords in the memory and the binary weights of each bit position.



(a) Unsigned doubleword



(b) The contents of memory location 00100H –00103H are the doubleword 12345678H.

FIGURE 1–6 The storage format for a 32-bit word in (a) a register and (b) four bytes of memory.

When a doubleword is stored in memory, its least significant byte is stored in the lowest-numbered memory location, and its most significant byte is stored in the highest-numbered memory location using the little endian format. Recall that this is also true for word-sized data. For example, a 12345678H that is stored in memory location 00100H–00103H is stored with the 78H in memory location 00100H, the 56H in location 00101H, the 34H in location 00102H, and the 12H in location 00103H.

To define doubleword-sized data, use the assembler directive **define doubleword(s),** or DD. (You can also use the DWORD directive in place of DD.) Example 1–24 shows both signed and unsigned numbers stored in memory using the DD directive.

EXAMPLE 1–24

```
                        ;Unsigned doubleword-sized data
                        ;
0000    0003E1C0        DATA1   DD      254400      ;define 254400 decimal
0004    87AC1234        DATA2   DD      87AC1234H   ;define 87AC1234 hexadecimal
0008    00000046        DATA3   DD      70          ;define 70 decimal
                        ;
                        ;Signed doubleword-sized data
                        ;
000C    FFEB8058        DATA4   DD      -1343400    ;define a -1343400 decimal
0010    000000C6        DATA5   DD      +198        ;define a +198 decimal
0014    FFFFFFFF        DATA6   DWORD   -1          ;define a -1 decimal
```

Integers may also be stored in memory that is of any width. The forms listed here are standard forms, but that doesn't mean that a 128-byte wide integer can't be stored in the memory. The microprocessor is flexible enough to allow any size of data. When nonstandard width numbers are stored in memory, the DB directive is normally used to store them. For example, the 24-bit number 123456H is stored using a DB 56H,34H,12H directive. Note that this conforms to the little endian format.

Real Numbers

Because many high-level languages use the Intel family of microprocessors, real numbers are often encountered. A real number, or a **floating-point number,** as it is often called, contains two parts: a mantissa, significand, or fraction; and an exponent. Figure 1–7 depicts both the 4- and 8-byte forms of real numbers as they are stored in any Intel system. Note that the 4-byte real number is called **single-precision** and the 8-byte form is called **double-precision.**

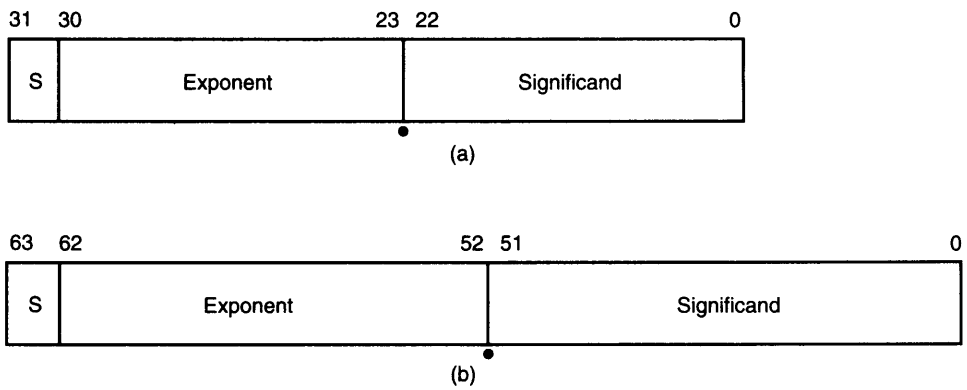


FIGURE 1–7 The floating-point numbers (a) single-precision using a bias of 7FH and (b) double-precision using a bias of 3FFH.

TABLE 1–7 Single-precision real numbers

Decimal	*Binary*	*Normalized*	*Sign*	*Biased Exponent*	*Mantissa*
+12	1100	1.1×2^3	0	10000010	1000000 00000000 00000000
−12	1100	-1.1×2^3	1	10000010	1000000 00000000 00000000
+100	1100100	1.1001×2^6	0	10000101	1001000 00000000 00000000
−1.75	1.11	-1.11×2^0	1	01111111	1100000 00000000 00000000
+0.25	.01	1.0×2^{-2}	0	01111101	0000000 00000000 00000000
+0.0	0	0	0	00000000	0000000 00000000 00000000

The form presented here is the same form specified by the IEEE[7] standard, IEEE-754, version 10.0. This standard has been adopted as the standard form of real numbers with virtually all high-level programming languages and many applications packages. The standard also applies to data manipulated by the numeric coprocessor in the personal computer. Figure 1–7 (a) shows the single-precision form that contains a sign-bit, an 8-bit exponent, and a 24-bit fraction (mantissa). Note that because applications often require double-precision floating-point numbers [see Figure 1–7 (b)], the Pentium–Pentium 4 with their 64-bit data bus perform memory transfers at twice the speed of the 80386/80486 microprocessors.

Simple arithmetic indicates that it should take 33 bits to store all three pieces of data. Not true—the 24-bit mantissa contains an **implied** (hidden) one-bit that allows the mantissa to represent 24 bits while being stored in only 23 bits. The hidden bit is the first bit of the normalized real number. When normalizing a number, it is adjusted so that its value is at least 1, but less than 2. For example, if 12 is converted to binary (1100_2), it is normalized and the result is 1.1×2^3. The 1 is not stored in the 23-bit mantissa portion of the number; the 1 is the hidden one-bit. Table 1–7 shows the single-precision form of this number and others.

The exponent is stored as a **biased exponent.** With the single-precision form of the real number, the bias is 127 (7FH) and with the double-precision form, it is 1023 (3FFH). The bias adds to the exponent before is stored into the exponent portion of the floating-point number. In the previous example, there is an exponent of 2^3, represented as a biased exponent of 127 + 3 or 130 (82H) in the single-precision form, or as 1026 (402H) in the double-precision form.

There are two exceptions to the rules for floating-point numbers. The number 0.0 is stored as all zeros. The number infinity is stored as all ones in the exponent and all zeros in the mantissa. The sign-bit indicates either a positive or a negative infinity.

As with other data types, the assembler can be used to define real numbers in both single- and double-precision forms. Because single-precision numbers are 32-bit numbers, use the DD directive or use the **define quadwords(s),** or DQ, directive to define 64-bit double-precision real numbers. Optional directives for real numbers are REAL4, REAL8, and REAL10 for defining single-, double-, and extended precision real numbers. Example 1–25 shows numbers defined in real number format.

EXAMPLE 1–25

```
                    ;Single-precision real numbers
                    ;
0000 3F9DF3B6       NUMB1   DD      1.234       ;define 1.234
0004 C1BB3333       NUMB2   DD      -23.4       ;define -23.4
0008 43D20000       NUMB3   REAL4   4.2E2       ;define 420
000C 3F9DF3B6       NUMB4   REAL4   1.234       ;define a 4-byte real number
                    ;
```

[7]IEEE is the Institute of Electrical and Electronic Engineers.

```
                ;Double-precision real numbers
                ;
0010            NUMB5  DQ      123.4        ;define 123.4
    405ED9999999999A
0018            NUMB6  REAL8   -23.4        ;define -23.4
    C1BB333333333333
0028            NUMB7  REAL8   123.4        ;define an 8-byte real number
    405ED9999999999A
                ;
                ;Extended-precision real numbers
                ;
0030            NUMB8  REAL10  123.4        ;define a 10-byte real number
    4005F6CCCCCCCCCCCCCD
```

1-4 SUMMARY

1. The mechanical computer age began with the advent of the abacus in 500 B.C. This first mechanical calculator remained unchanged until 1642, when Blaise Pascal improved it. An early mechanical computer system was the Analytical Engine developed by Charles Babbage in 1823. Unfortunately, this machine never functioned because of his inability to create the necessary machine parts.
2. The first electronic calculating machine was developed during World War II by Konrad Zuse, an early pioneer of digital electronics. His computer, the Z3, was used in aircraft and missile design for the German war effort.
3. The first electronic computer, which used vacuum tubes, was placed into operation in 1943 to break secret German military codes. This first electronic computer system, the Colossus, was invented by Alan Turing. Its only problem was that the program was fixed and could not be changed.
4. The first general-purpose, programmable electronic computer system was developed in 1946 at the University of Pennsylvania. This first modern computer was called the ENIAC (Electronics Numerical Integrator and Calculator).
5. The first high-level programming language, called FLOW-MATIC, was developed for the UNIVAC I computer by Grace Hopper in the early 1950s. This led to FORTRAN and other early programming languages such as COBOL.
6. The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor—a programmable controller on a chip—that was meager by today's standards. It addressed a mere 4096 four-bit memory locations. Its instruction set contained only 45 different instructions.
7. Microprocessors that are common today include the 8086/8088, which were the first 16-bit microprocessors. Following these early 16-bit machines were the 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, and Pentium 4 processors. The architecture has changed from 16 bits to 32 bits and soon, with the Merced, to 64 bits. With each newer version, improvements followed that increased the processor's speed and performance. From all indications, this process of speed and performance improvement will continue.
8. Microprocessor-based personal computers contain memory systems that include three main areas: TPA (transient program area), system area, and extended memory. The TPA holds application programs, the operating system, and drivers. The system area contains memory used for video display cards, disk drives, and the BIOS ROM. The extended memory area is only available to the 80286 through the Pentium 4 microprocessor in an AT-style personal computer system.
9. The 8086/8088 address 1M byte of memory from location 00000H–FFFFFH. The 80286 and 80386SX address 16M bytes of memory from location 000000H–FFFFFFH. The 80386SL addresses 32M bytes of memory from location 0000000H–1FFFFFFH. The 80386DX and 80486 through Pentium 4 processors address 4G bytes

of memory from location 00000000H–FFFFFFFFH. In addition, the Pentium Pro through the Pentium 4 can run with a 36-bit address and access up to 64G bytes of memory from location 000000000H–FFFFFFFFH.

10. All versions of the 8086–80486 and Pentium–Pentium 4 microprocessors address 64K bytes of I/O address space. These I/O ports are numbered from 0000H–FFFFH with I/O ports 0000H–03FFH reserved for use by the personal computer system.
11. The operating system in many personal computers is either MSDOS (Microsoft disk operating system) or PCDOS (personal computer disk operating system from IBM). The operating system performs the task of operating or controlling the computer system, along with its I/O devices.
12. The microprocessor is the controlling element in a computer system. The microprocessor performs data transfers, does simple arithmetic and logic operations, and makes simple decisions. The microprocessor executes programs stored in the memory system to perform complex operations in short periods of time.
13. All computer systems contain three buses to control memory and I/O. The address bus is used to request a memory location or I/O device. The data bus transfers data between the microprocessor and its memory and I/O spaces. The control bus controls the memory and I/O, and requests reading or writing of data. Control is accomplished with IORC (I/O read control), IOWC (I/O write control), MRDC (memory read control), and MWTC (memory write control).
14. Numbers are converted from any number base to decimal by noting the weights of each position. The weight of the position to the left of the radix point is always the units position in any number system. The position to the left of the units position is always the radix times one. Succeeding positions are determined by multiplying by the radix. The weight of the position to the right of the radix point is always determined by dividing by the radix.
15. Conversion from a whole decimal number to any other base is accomplished by dividing by the radix. Conversion from a fractional decimal number is accomplished by multiplying by the radix.
16. Hexadecimal data are represented in hexadecimal form or in a code called binary-coded hexadecimal (BCH). A binary-coded hexadecimal number is one that is written with a 4-bit binary number that represents each hexadecimal digit.
17. The ASCII code is used to store alphabetic or numeric data. The ASCII code is a 7-bit code; it can have an eighth bit that is used to extend the character set from 128 codes to 256 codes. The carriage return (Enter) code returns the print head or cursor to the left margin. The line feed code moves the cursor or print head down one line.
18. Binary-coded decimal (BCD) data are sometimes used in a computer system to store decimal data. These data are stored either in packed (two digits per byte) or unpacked (one digit per byte) form.
19. Binary data are stored as a byte (8 bits), word (16 bits), or doubleword (32 bits) in a computer system. These data may be unsigned or signed. Signed negative data are always stored in the two's complement form. Data that are wider than 8 bits are always stored using the little endian format.
20. Floating-point data are used in computer systems to store whole, mixed, and fractional numbers. A floating-point number is composed of a sign, a mantissa, and an exponent.
21. The assembler directives DB or BYTE define bytes, DW or WORD define words, DD or DWORD define doublewords, and DQ or QWORD define quadwords.
22. Example 1–26 shows the assembly language formats for storing numbers as bytes, words, doublewords, and real numbers. Also shown are ASCII-coded character strings.

EXAMPLE 1–26

```
                        ;ASCII data
                        ;
0000  54 68 69 73 20 69 MES1    DB      'This is a character string in ASCII'
      73 20 61 20 63 68
```

```
      61 72 61 63 74 65
      72 20 73 74 72 69
      6E 67 20 69 6E 20
      41 53 43 49 49
0023  53 6F 20 69 73 20   MES2    DB      'So is this'
      74 68 69 73
                          ;BYTE data
                          ;
002D  17                  DATA1   DB      23          ;23 decimal
002E  DE                  DATA2   DB      -34         ;-34 decimal
002F  34                  DATA3   DB      34H         ;34 hexadecimal
                          ;
                          ;WORD data
                          ;
0030  1000                DATA4   DW      1000H       ;1000 hexadecimal
0032  FF9C                DATA5   DW      -100        ;-100 decimal
0034  000C                DATA6   DW      +12         ;=12 decimal
                          ;
                          ;DOUBLEWORD data
                          ;
0036  00001000            DATA7   DD      1000H       ;1000 hexadecimal
003A  FFFFFED4            DATA8   DD      -300        ;-300 decimal
003E  00012345            DATA9   DD      12345H      ;12345 hexadecimal
                          ;
                          ;Real data
                          ;
0042  4015C28F            DATA10  REAL4   2.34       ;2.34 decimal
0046  C00CCCCD            DATA11  REAL4   -2.2       ;-2.2 decimal
004A                      DATA12  REAL8   100.3      ;100.3 decimal
      4059133333333333
```

1–5 QUESTIONS AND PROBLEMS

1. Who developed the Analytical Engine?
2. The 1890 census used a new device called a punched card. Who developed the punch card?
3. Who was the founder of IBM Corporation?
4. Who developed the first electronic calculator?
5. The first electronic computer system was developed for what purpose?
6. The first general-purpose, programmable computer was called the _____.
7. The world's first microprocessor was developed in 1971 by _____.
8. Who was the Countess of Lovelace?
9. Who developed the first high-level programming language called FLOW-MATIC?
10. What is a von Neumann machine?
11. Which 8-bit microprocessor ushered in the age of the microprocessor?
12. The 8085 microprocessor, introduced in 1977, has sold _____ copies.
13. Which Intel microprocessor was the first to address 1M bytes of memory?
14. The 80386SL addresses _____ bytes of memory.
15. How much memory is available to the 80486 microprocessor?
16. When did Intel introduce the Pentium microprocessor?
17. When did Intel introduce the Pentium Pro processor?
18. When did Intel introduce the Pentium 4 microprocessor?
19. Which Intel microprocessors address 64G of memory?

20. What is the acronym MIPs?
21. What is the acronym CISC?
22. A binary bit stores a(n) _____ or a(n) _____.
23. A computer K is equal to _____ bytes.
24. A computer M is equal to _____ K bytes.
25. A computer G is equal to _____ M bytes.
26. How many typewritten pages of information are stored in a 4G-byte memory system?
27. The first 1M byte of memory in a computer system contains a(n) _____ and a(n) _____ area.
28. How much memory is found in the transient program area?
29. How much memory is found in the systems area?
30. The 8086 microprocessor addresses _____ bytes of memory.
31. The Pentium 4 microprocessor addresses _____ bytes of memory.
32. Which microprocessors address 4G bytes of memory?
33. Convert the following binary numbers into decimal:
    (a) 1101.01
    (b) 111001.0011
    (c) 101011.0101
    (d) 111.0001
34. Convert the following octal numbers into decimal:
    (a) 234.5
    (b) 12.3
    (c) 7767.07
    (d) 123.45
    (e) 72.72
35. Convert the following hexadecimal numbers into decimal:
    (a) A3.3
    (b) 129.C
    (c) AC.DC
    (d) FAB.3
    (e) BB8.0D
36. Convert the following decimal integers into binary, octal, and hexadecimal:
    (a) 23
    (b) 107
    (c) 1238
    (d) 92
    (e) 173
37. Convert the following decimal numbers into binary, octal, and hexadecimal:
    (a) 0.625
    (b) .00390625
    (c) .62890625
    (d) 0.75
    (e) .9375
38. Convert the following hexadecimal numbers into binary-coded hexadecimal code (BCH):
    (a) 23
    (b) AD4
    (c) 34.AD

(d) BD32
(e) 234.3

39. Convert the following binary-coded hexadecimal numbers into hexadecimal:
    (a) 1100 0010
    (b) 0001 0000 1111 1101
    (c) 1011 1100
    (d) 0001 0000
    (e) 1000 1011 1010
40. Convert the following binary numbers to the one's complement form:
    (a) 1000 1000
    (b) 0101 1010
    (c) 0111 0111
    (d) 1000 0000
41. Convert the following binary numbers to the two's complement form:
    (a) 1000 0001
    (b) 1010 1100
    (c) 1010 1111
    (d) 1000 0000
42. Define byte, word, and doubleword.
43. Convert the following words into ASCII-coded character strings:
    (a) FROG
    (b) Arc
    (c) Water
    (d) Well
44. What is the ASCII code for the Enter key and what is its purpose?
45. Use an assembler directive to store the ASCII-character string 'What time is it?' in the memory.
46. Convert the following decimal numbers into 8-bit signed binary numbers:
    (a) +32
    (b) –12
    (c) +100
    (d) –92
47. Convert the following decimal numbers into signed binary words:
    (a) +1000
    (b) –120
    (c) +800
    (d) –3212
48. Use an assembler directive to store –34 into the memory.
49. Show how the following 16-bit hexadecimal numbers are stored in the memory system (use the standard Intel format):
    (a) 1234H
    (b) A122H
    (c) B100H
50. What is the difference between the big endian and little endian formats for storing numbers that are larger than eight bits in width?
51. Use an assembler directive to store a 123A hexadecimal into the memory.

52. Convert the following decimal numbers into both packed and unpacked BCD forms:
    (a) 102
    (b) 44
    (c) 301
    (d) 1000
53. Convert the following binary numbers into signed decimal numbers:
    (a) 10000000
    (b) 00110011
    (c) 10010010
    (d) 10001001
54. Convert the following BCD numbers (assume that these are packed numbers) into decimal numbers:
    (a) 10001001
    (b) 00001001
    (c) 00110010
    (d) 00000001
55. Convert the following decimal numbers into single-precision floating-point numbers:
    (a) +1.5
    (b) −10.625
    (c) +100.25
    (d) −1200
56. Convert the following single-precision floating-point numbers into decimal numbers:
    (a) 0 10000000 11000000000000000000000
    (b) 1 01111111 00000000000000000000000
    (c) 0 10000010 10010000000000000000000
57. Use the Internet to write a short report about any one of the following computer pioneers:
    (a) Charles Babbage
    (b) Konrad Zuse
    (c) Joseph Jacquard
    (d) Herman Hollerith
58. Use the Internet to write a short report about any one of the following computer languages:
    (a) COBOL
    (b) ALGOL
    (c) FORTRAN
    (d) PASCAL
59. Use the Internet to write a short report detailing the features of the Merced microprocessor.

CHAPTER 2

The Microprocessor and its Architecture

INTRODUCTION

This chapter presents the microprocessor as a programmable device by first looking at its internal programming model and then at how it addresses its memory space. The architecture of the entire family of Intel microprocessors is presented simultaneously, as are the ways that the family members address the memory system.

The addressing modes for this powerful family of microprocessors are described for both the real and protected modes of operation. Real mode memory exists at locations 00000H–FFFFFH—the first 1M byte of the memory system—and is present on all versions of the microprocessor. Protected mode memory exists at any location in the entire memory system, but is available only to the 80286–Pentium 4, not to the earlier 8086 or 8088 microprocessors. Protected mode memory for the 80286 contains 16M bytes; for the 80386–Pentium, 4G bytes; and for the Pentium Pro through the Pentium 4, either 4G or 64G bytes.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Describe the function and purpose of each program-visible register in the 8086–80486 and Pentium–Pentium 4 microprocessors.
2. Detail the flag register and the purpose of each flag bit.
3. Describe how memory is accessed using real mode memory-addressing techniques.
4. Describe how memory is accessed using protected mode memory-addressing techniques.
5. Describe the program-invisible registers found within the 80286 through Pentium 4 microprocessors.
6. Detail the operation of the memory-paging mechanism.

2–1 INTERNAL MICROPROCESSOR ARCHITECTURE

Before a program is written or any instruction investigated, the internal configuration of the microprocessor must be known. This section of the chapter details the program-visible internal architecture of the 8086–80486 and the Pentium–Pentium 4 microprocessors. Also detailed are the function and purpose of each of these internal registers.

The Programming Model

The programming model of the 8086 through the Pentium 4 is considered to be **program visible** because its registers are used during application programming and are specified by the instructions. Other registers, detailed later in this chapter, are considered to be **program invisible** because they are not addressable directly during applications programming, but may be used indirectly during system programming. Only the 80286 and above contain the program-invisible registers used to control and operate the protected memory system.

Figure 2–1 illustrates the programming model of the 8086 through the Pentium 4 microprocessor. The earlier 8086, 8088, and 80286 contain **16-bit** internal architectures, a subset of the registers shown in

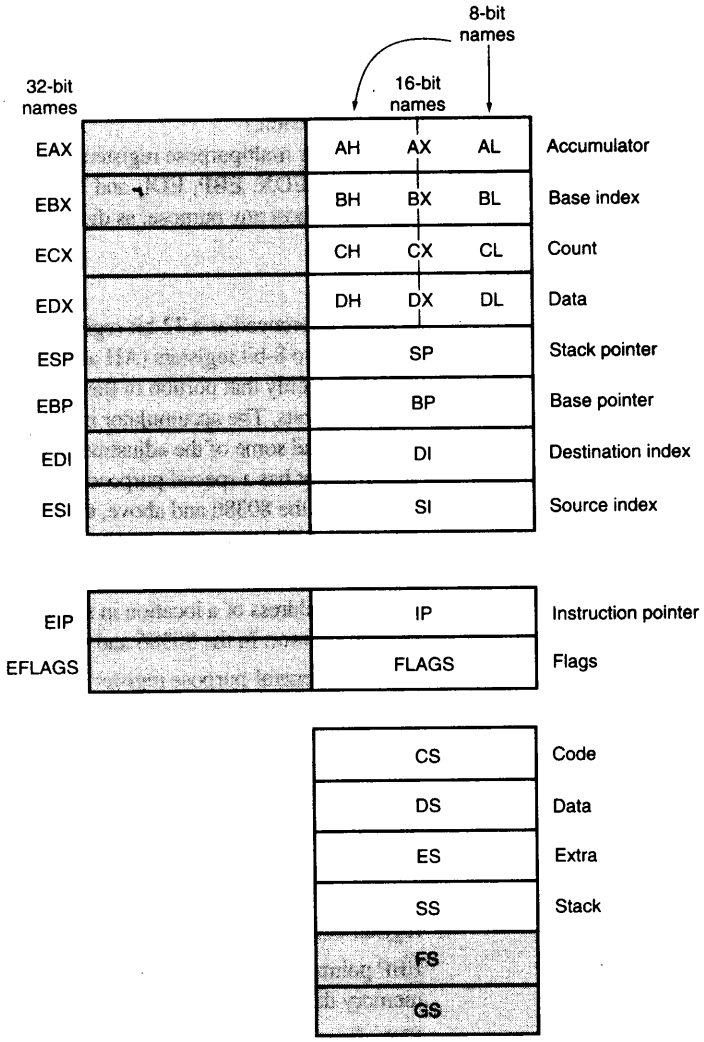


FIGURE 2–1 The programming model of the Intel 8086 through the Pentium 4.

Notes:
1. The shaded areas registers exist only on the 80386 through the Pentium 4.

2. The FS and GS register have no special names.

Figure 2–1. The 80386 through the Pentium 4 microprocessors contain full **32-bit** internal architectures. The architectures of the earlier 8086 through the 80286 are fully upward-compatible to the 80386 through the Pentium 4. The shaded areas in this illustration represent registers that are not found in the 8086, 8088, or 80286 microprocessors and are enhancements provided on the 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, and Pentium 4 microprocessors.

The programming model contains 8-, 16-, and 32-bit registers. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL and are referred to when an instruction is formed using these two-letter designations. For example, an ADD AL,AH instruction adds the 8-bit contents of AH to AL. (Only AL changes due to this instruction.) The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, SI, IP, FLAGS, CS, DS, ES, SS, FS, and GS. These registers are also referenced with the two-letter designations. For example, an ADD DX,CX instruction adds the 16-bit contents of CX to DX. (Only DX changes due to this instruction.) The extended 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, EIP, and EFLAGS. These 32-bit extended registers, and 16-bit registers FS and GS, are available only in the 80386 and above. These registers are referenced by the designations FS or GS for the two new 16-bit registers, and by a three-letter designation for the 32-bit registers. For example, an ADD ECX,EBX instruction adds the 32-bit contents of EBX to ECX. (Only ECX changes due to this instruction.)

Some registers are general-purpose or multipurpose registers, while some have special purposes. The **multipurpose registers** include EAX, EBX, ECX, EDX, EBP, EDI, and ESI. These registers hold various data sizes (bytes, words, or doublewords) and are used for almost any purpose, as dictated by a program.

Multipurpose Registers

EAX (accumulator) — EAX is referenced as a 32-bit register (EAX), as a 16-bit register (AX), or as either of two 8-bit registers (AH and AL). Note that if an 8- or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered to be a multipurpose register. In the 80386 and above, the EAX register may also hold the offset address of a location in the memory system.

EBX (base index) — EBX is addressable as EBX, BX, BH, or BL. The BX register sometimes holds the offset address of a location in the memory system in all versions of the microprocessor. In the 80386 and above, EBX also can address memory data.

ECX (count) — ECX is a general-purpose register that also holds the count for various instructions. In the 80386 and above, the ECX register also can hold the offset address of memory data. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate, and LOOP/LOOPD instructions. The shift and rotate instructions use CL as the count, the repeated string instructions use CX, and the LOOP/LOOPD instructions use either CX or ECX.

EDX (data) — EDX is a general-purpose register that holds a part of the result from a multiplication or part of the dividend before a division. In the 80386 and above, this register can also address memory data. DX is used in 16 bit, DH and DL are pairs.

EBP (base pointer) — EBP points to a memory location in all versions of the microprocessor for memory data transfers. This register is addressed as either BP or EBP.

EDI (destination index) — EDI often addresses string destination data for the string instructions. It also functions as either a 32-bit (EDI) or 16-bit (DI) general-purpose register.

ESI (source index) — ESI is used as either ESI or SI. The source index register often addresses source string data for the string instructions. Like EDI, ESI also functions as a general purpose register. As a 16-bit register, it is addressed as SI; as a 32-bit register, it is addressed as ESI.

Special-purpose Registers. The special-purpose registers include EIP, ESP, EFLAGS;| and the segment registers CS, DS, ES, SS, FS, and GS.

EIP (instruction pointer) — EIP addresses the next instruction in a section of memory defined as segment. This register is IP (16 bits) when the microprocessor operates in the real mode and EIP (32 bits) when the 80386 and above operate in the protected mode. Note that the 8086, 8088, and 80286 do contain EIP, and only the 80286 and above operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction.

ESP (stack pointer) — ESP addresses an area of memory called the stack. The stack memory stores data through this pointer and is explained later in the text with instructions that address stack data. This register is referred to as SP if used as a 16-bit register and ESP if referred to as a 32-bit register.

EFLAGS — EFLAGS indicate the condition of the microprocessor and control its operation. Figure 2–2 shows the flag registers of all versions of the microprocessor. Note that the flags are upward-compatible from the 8086/8088 to the Pentium 4 microprocessor. The 8086–80286 contain a FLAG register (16 bits) and the 80386 and above contain an EFLAG register (32-bit extended flag register).

The rightmost five flag bits and the overflow flag change after many arithmetic and logic instructions execute. The flags never change for any data transfer or program control operation. Some of the flags are also used to control features found in the microprocessor. Following is a list of each flag bit, with a brief description of their function. As instructions are introduced in subsequent chapters, additional detail on the flag bits is provided. The rightmost five flags and the overflow flag are changed by most arithmetic and logic operations, while data transfers do not affect them.

C (carry) — Carry holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions, as dictated by some programs and procedures. This is especially true of the DOS function calls detailed in later chapters and Appendix A.

P (parity) — Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is a count of ones in a number expressed as even or odd. For example, if a number contains three binary one bits, it has odd parity. If a number contains zero one bits, it has even parity. The parity flag finds little application in modern programming and was implemented in early Intel microprocessors for checking data in data communications environments. Today parity checking is often accomplished by the data communications equipment instead of the microprocessor.

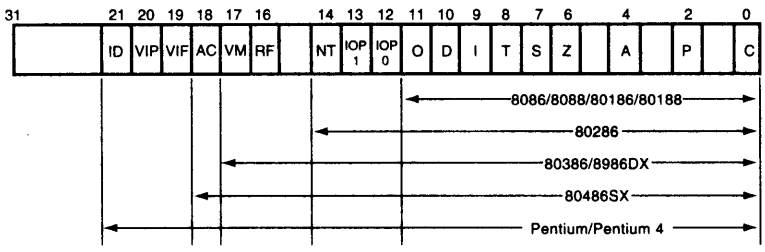


FIGURE 2–2 The EFLAG and FLAG register counts for the entire 80X86 and Pentium microprocessor family.

A (auxiliary carry) The auxiliary carry holds the carry (half-carry) after addition or the borrow after subtraction between bits positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z (zero) The zero flag shows that the result of an arithmetic or logic operation is zero. If Z = 1, the result is zero; if Z = 0, the result is not zero.

S (sign) The sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If S = 1, the sign bit (leftmost bit of a number) is set or negative; if S = 0, the sign bit is cleared or positive.

T (trap) The trap flag enables trapping through an on-chip debugging feature. (A program is debugged to find an error or bug.) If the T flag is enabled (1), the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control registers. If the T flag is a logic 0, the trapping (debugging) feature is disabled. The CodeView program can use the trap feature and debug registers to debug faulty software.

I (interrupt) The interrupt flag controls the operation of the INTR (interrupt request) input pin. If I = 1, the INTR pin is enabled; if I = 0, the INTR pin is disabled. The state of the I flag bit is controlled by the STI **(set I flag)** and CLI **(clear I flag)** instructions.

D (direction) The direction flag selects either the increment or decrement mode for the DI and/or SI registers during string instructions. If D = 1, the registers are automatically decremented; if D = 0, the registers are automatically incremented. The D flag is set with the STD **(set direction)** and cleared with the CLD **(clear direction)** instructions.

O (overflow) Overflows occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. For example, if a 7FH (+127) is added—using an 8-bit addition—to a 01H (+1), the result is 80H (–128). This result represents an overflow condition indicated by the overflow flag for signed addition. For unsigned operations, the overflow flag is ignored.

IOPL (I/O privilege level) IOPL is used in protected mode operation to select the privilege level for I/O devices. If the current privilege level is higher or more trusted than the IOPL, I/O executes without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs, causing execution to suspend. Note that an IOPL of 00 is the highest or most trusted; if IOPL is 11, it's the lowest or least trusted.

NT (nested task) The nested task flag indicates that the current task is nested within another task in protected mode operation. This flag is set when the task is nested by software.

RF (resume) The resume flag is used with debugging to control the resumption of execution after the next instruction.

VM (virtual mode) The VM flag bit selects virtual mode operation in a protected mode system. A virtual mode system allows multiple DOS memory partitions that are 1M byte in length to coexist in the memory system. Essentially, this allows the system program to execute multiple DOS programs.

AC (alignment check) The alignment check flag bit activates if a word or doubleword is addressed on a non-word or non-doubleword boundary. Only the 80486SX microprocessor contains the alignment check bit that is primarily used by its companion numeric coprocessor, the 80487SX, for synchronization.

VIF (virtual interrupt flag) The VIF is a copy of the interrupt flag bit available to the Pentium–Pentium 4 microprocessors.

VIP (virtual interrupt pending) VIP provides information about a virtual mode interrupt for the Pentium–Pentium 4 microprocessors. This is used in multitasking environments to provide the operating system with virtual interrupt flags and interrupt pending information.

ID (identification) The ID flag indicates that the Pentium–Pentium 4 microprocessors support the CPUID instruction. The CPUID instruction provides the system with information about the Pentium microprocessor, such as its version number and manufacturer.

Segment Registers. Additional registers, called segment registers, generate memory addresses when combined with other registers in the microprocessor. There are either four or six segment registers in various versions of the microprocessor. A segment register functions differently in the real mode when compared to the protected mode operation of the microprocessor. Details on their function in real and protected mode are provided later in this chapter. Following is a list of each segment register, along with its function in the system:

CS (code) The code segment is a section of memory that holds the code (programs and procedures) used by the microprocessor. The code segment register defines the starting address of the section of memory holding code. In real mode operation, it defines the start of a 64K-byte section of memory; in protected mode, it selects a descriptor that describes the starting address and length of a section of memory holding code. The code segment is limited to 64K bytes in the 8088–80286, and 4G bytes in the 80386 and above when these microprocessors operate in the protected mode.

DS (data) The data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address. As with the code segment and other segments, the length is limited to 64K bytes in the 8086–80286, and 4G bytes in the 80386 and above.

ES (extra) The extra segment is an additional data segment that is used by some of the string instructions to hold destination data.

SS (stack) The stack segment defines the area of memory used for the stack. The stack entry point is determined by the stack segment and stack pointer registers. The BP register also addresses data within the stack segment.

FS and GS The FS and GS segments are supplemental segment registers available in the 80386, 80486, and Pentium through the Pentium 4 microprocessors to allow two additional memory segments for access by programs.

2-2 REAL MODE MEMORY ADDRESSING

The 80286 and above operate in either the real or protected mode. Only the 8086 and 8088 operate exclusively in the real mode. This section of the text details the operation of the microprocessor in the real mode. **Real mode operation** allows the microprocessor to address only the first 1M byte of memory space—even if it is the Pentium 4 microprocessor. Note that the first 1M byte of memory is called either the **real memory** or **conventional memory** system. The DOS operating system requires the microprocessor to operate in the real mode. Real mode operation allows application software written for the 8086/8088, which contain only 1M byte of memory, to function in the

80286 and above without changing the software. The upward compatibility of software is partially responsible for the continuing success of the Intel family of microprocessors. In all cases, each of these microprocessors begins operation in the real mode by default whenever power is applied or the microprocessor is reset.

Segments And Offsets

A combination of a segment address and an offset address access a memory location in the real mode. All real mode memory addresses must consist of a segment address plus an offset address. The **segment address,** located within one of the segment registers, defines the beginning address of any 64K-byte memory segment. The **offset address** selects any location within the 64K byte memory segment. Segments in the real mode always have a length of 64K bytes. Figure 2–3 shows how the segment plus offset addressing scheme selects a memory location. This illustration shows a memory segment that begins at location 10000H and ends at location 1FFFFH—64K bytes in length. It also shows how an offset address, sometimes called a **displacement,** of F000H selects location 1F000H in the memory system. Note that the offset or displacement is the distance above the start of the segment, as shown in Figure 2–3.

The segment register in Figure 2–3 contains a 1000H, yet it addresses a starting segment at location 10000H. In the real mode, each segment register is internally appended with a **0H** on its rightmost end. This forms a 20-bit memory address, allowing it to access the start of a segment. The microprocessor must generate a 20-bit memory address to access a location within the first 1M of memory. For example, when a segment register contains a 1200H, it addresses a 64K-byte memory segment beginning at location 12000H. Likewise, if a segment register contains a 1201H, it addresses a memory segment beginning at location 12010H. Because of the internally appended 0H, real mode segments can begin only at a 16-byte boundary in the memory system. This 16-byte boundary is often called a **paragraph.**

Because a real mode segment of memory is 64K in length, once the beginning address is known, the **ending address** is found by adding FFFFH. For example, if a segment register contains 3000H, the first address of the segment is 30000H, and the last address is 30000H + FFFFH or 3FFFFH. Table 2–1 shows several examples of segment register contents, and the starting and ending addresses of the memory segments selected by each segment address.

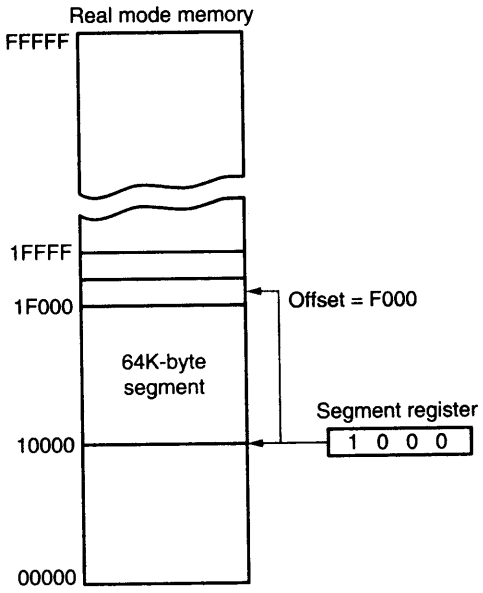


FIGURE 2–3 The real mode memory-addressing scheme, using a segment address plus an offset.

TABLE 2–1 Example segment addresses.

Segment Register	*Starting Address*	*Ending Address*
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

The offset address, which is a part of the address, is added to the start of the segment to address a memory location within the memory segment. For example, if the segment address is 1000H and the offset address is 2000H, the microprocessor addresses memory location 12000H. The offset address is always added to the starting address of the segment to locate the data. The segment and offset address is sometimes written as 1000:2000 for a segment address of 1000H with an offset of 2000H.

In the 80286 (with special external circuitry), and the 80386 through the Pentium 4, an extra 64K minus 16 bytes of memory is addressable when the segment address is FFFFH and the HIMEM.SYS driver is installed in the system. This area of memory (0FFFF0H–10FFEFH) is referred to as **high memory.** When an address is generated using a segment address of FFFFH, the A20 address pin is enabled (if supported) when an offset is added. For example, if the segment address is FFFFH and the offset address is 4000H, the machine addresses memory location FFFF0H + 4000H or 103FF0H. Notice that the A20 address line is the one in address 103FF0H. If A20 is not supported, the address is generated as 03FF0H because A20 remains a logic zero.

Some addressing modes combine more than one register and an offset value to form an offset address. When this occurs, the sum of these values may exceed FFFFH. For example, the address accessed in a segment whose segment address is 4000H, and whose offset address is specified as the sum of F000H plus 3000H, will access memory location 42000H instead of location 52000H. When the F000H and 3000H are added, they form a 16-bit **(modulo 16)** sum of 2000H used as the offset address; not 12000H, the true sum. Note that the carry of 1 (F000H + 3000H = 12000H) is dropped for this addition to form the offset address of 2000H. This means that the address is generated as 4000:2000 or 42000H.

Default Segment and Offset Registers

The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules, which apply in the real and protected mode, define the segment register and offset register combination. For example, the code segment register is always used with the instruction pointer to address the next instruction in a program. This combination is **CS:IP** or **CS:EIP,** depending upon the microprocessor's mode of operation. The **code segment** register defines the start of the code segment and the **instruction pointer** locates the next instruction within the code segment. This combination (CS:IP or CS:EIP) locates the next instruction executed by the microprocessor. For example, if CS = 1400H and IP/EIP = 1200H, the microprocessor fetches its next instruction from memory location 14000H + 1200H or 15200H.

Another of the default combinations is the **stack.** Stack data are referenced through the stack segment at the memory location addressed by either the stack pointer (SP/ESP) or the base pointer (BP/EBP). These combinations are referred to as SS:SP (SS:ESP) or SS:BP (SS:EBP). For example, if SS = 2000H and BP = 3000H, the microprocessor addresses memory location 23000H for the stack segment memory location. Note that in real mode, only the rightmost 16 bits of the extended register address a location within the memory segment. In the 80386–Pentium 4, never place a number larger than FFFFH into an offset register if the microprocessor is operated in the real mode. This causes the system to halt and indicate an addressing error.

Other defaults are shown in Table 2–2 for addressing memory using any Intel microprocessor with 16-bit registers. Table 2–3 shows the defaults assumed in the 80386 and above when using 32-bit registers. Note that the 80386 and above have a far greater selection of segment/ offset address combinations than do the 8086 through the 80286 microprocessors.

TABLE 2–2 8086–80486 and Pentium–Pentium 4 default 16-bit segment and offset address combinations.

Segment	*Offset*	*Special Purpose*
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8-bit number, or a 16-bit number	Data address
ES	DI for string instructions	String destination address

TABLE 2–3 80386 through the Pentium 4 default 32-bit segment and offset address combinations.

Segment	*Offset*	*Special Purpose*
CS	EIP	Instruction address
SS	ESP and EBP	Stack address
DS	EAX, EBX, ECX, EDX, ESI, EDI, an 8-bit number, or a 32-bit number	Data address
ES	EDI for string instructions	String destination address
FS	No default	General address
GS	No default	General address

The 8086–80286 microprocessors allow four memory segments and the 80386 and above allow six memory segments. Figure 2–4 shows a system that contains four memory segments. Note that a memory segment can touch or even overlap if 64K bytes of memory are not required for a segment. Think of segments as windows that can be moved over any area of memory to access data or code. Also note that a program can have more than four or six segments, but can only access four or six segments at a time.

Suppose that an application program requires 1000H bytes of memory for its code, 190H bytes of memory for its data, and 200H bytes of memory for its stack. This application does not require an extra segment. When this program is placed in the memory system by DOS, it is loaded in the TPA* at the first available area of memory above the drivers and other TPA programs. This area is indicated by a **free-pointer** that is maintained by DOS. Program loading is handled automatically by the **program loader** located within DOS. Figure 2–5 shows how this application is stored in the memory system. The segments show an overlap because the amount of data in them does not require 64K bytes of memory. The side view of the segments clearly shows the overlap. It also shows how segments can be moved over any area of memory by changing the segment starting address. Fortunately, the DOS program loader calculates and assigns segment starting addresses. This is explained in Chapter 7, which details the operation of the assembler, BIOS, and DOS for an assembly language program.

Segment and Offset Addressing Scheme Allows Relocation

The segment and offset addressing scheme seems unduly complicated. It is complicated, but it also affords an advantage to the system. This complicated scheme of segment plus offset addressing allows programs to be relocated in the memory system. It also allows programs written to function in the real mode to operate in a

* Transient processor area.

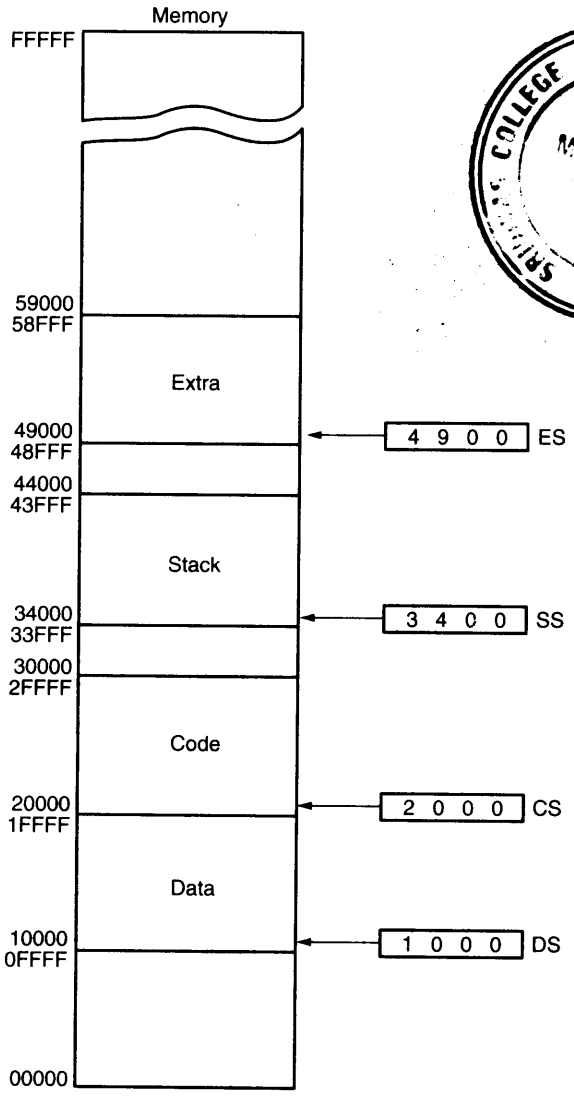


FIGURE 2–4 A memory system showing the placement of four memory segments.

protected mode system. A relocatable program is one that can be placed into any area of memory and executed without change. Relocatable data are data that can be placed in any area of memory and used without any change to the program. The segment and offset addressing scheme allows both programs and data to be relocated without changing a thing in a program or data. This is ideal for use in a general-purpose computer system in which not all machines contain the same memory areas. The personal computer memory structure is different from machine to machine, requiring relocatable software and data.

Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses. This is accomplished by moving the entire program, as a block, to a new area and then changing only the contents of the segment registers. If an instruction is 4 bytes above the start of the segment, its offset address is 4. If the entire program is moved to a new area of memory, this offset address of 4 still points to 4 bytes above the start of the segment. Only the contents of the segment register must be changed to address the program in the new area of memory. Without this feature, a program would have to be extensively rewritten or altered before it is moved. This would require additional time or many versions of a program for the many different configurations of computer systems.

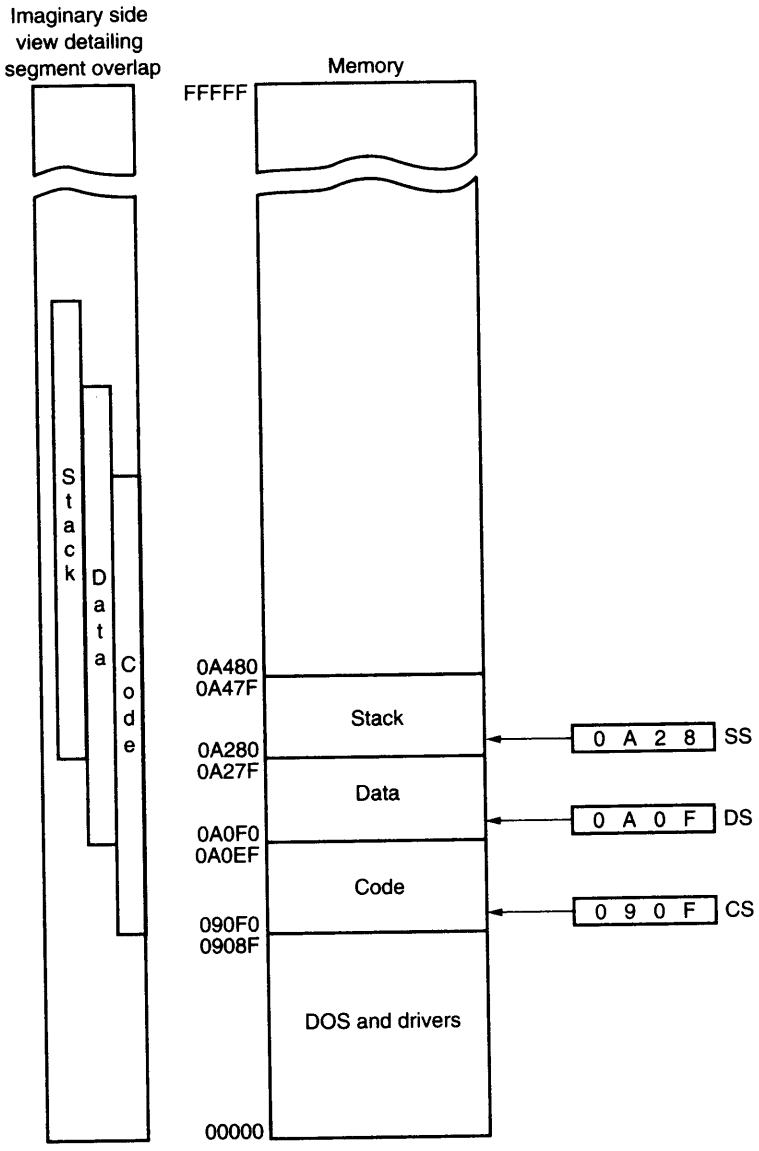


FIGURE 2–5 An application program containing a code, data, and stack segment loaded into a DOS system memory.

2–3 INTRODUCTION TO PROTECTED MODE MEMORY ADDRESSING

Protected mode memory addressing (80286 and above) allows access to data and programs located above the first 1M byte of memory, as well as within the first 1M byte of memory. Addressing this extended section of the memory system requires a change to the segment plus an offset addressing scheme used with real mode memory addressing. When data and programs are addressed in extended memory, the offset address is still used to access information located within the memory segment. One difference is that the segment address, as discussed with real mode memory addressing, is no longer present in the protected mode. In place of the segment address, the segment register contains a **selector** that selects a descriptor from a descriptor table. The **descriptor** describes the memory

segment's location, length, and access rights. Because the segment register and offset address still access memory, protected mode instructions are identical to real mode instructions. In fact, most programs written to function in the real mode will function without change in the protected mode. The difference between modes is in the way that the segment register is interpreted by the microprocessor to access the memory segment. Another difference, in the 80386 and above, is that the offset address can be a 32-bit number instead of a 16-bit number in the protected mode. A 32-bit offset address allows the microprocessor to access data within a segment that can be up to 4G bytes in length.

Selectors And Descriptors

The selector, located in the segment register, selects one of 8192 descriptors from one of two tables of descriptors. The **descriptor** describes the location, length, and access rights of the segment of memory. Indirectly, the segment register still selects a memory segment, but not directly as in the real mode. For example, in the real mode, if CS = 0008H, the code segment begins at location 00080H. In the protected mode, this segment number can address any memory location in the entire system for the code segment, as explained shortly.

There are two descriptor tables used with the segment registers: one contains global descriptors and the other contains local descriptors. The **global descriptors** contain segment definitions that apply to all programs, while the **local descriptors** are usually unique to an application. You might call a global descriptor a **system descriptor** and call a local descriptor an **application descriptor.** Each descriptor table contains 8192 descriptors, so a total of 16,384 total descriptors are available to an application at any time. Because the descriptor describes a memory segment, this allows up to 16,384 memory segments to be described for each application.

Figure 2–6 shows the format of a descriptor for the 80286 through the Pentium 4. Note that each descriptor is 8 bytes in length, so the global and local descriptor tables are each a maximum of 64K bytes in length. Descriptors for the 80286 and the 80386 through the Pentium 4 differ slightly, but the 80286 descriptor is upward-compatible.

The **base address** portion of the descriptor indicates the starting location of the memory segment. For the 80286 microprocessor, the base address is a 24-bit address, so segments begin at any location in its 16M bytes of memory. Note that the paragraph boundary limitation is removed in these microprocessors when operated in the protected mode. The 80386 and above use a 32-bit base address that allows segments to begin at any location in its 4G bytes of memory. Notice how the 80286 descriptor's base address is upward-compatible to the 80386 through the Pentium II descriptor because its most-significant 16 bits are 0000H. Refer to Chapters 16 and 17 for additional detail on the 64G memory space provided by the Pentium Pro and the Pentium II.

The **segment limit** contains the last offset address found in a segment. For example, if a segment begins at memory location F00000H and ends at location F000FFH, the base address is F00000H and the limit is FFH. For the 80286 microprocessor, the base address is F00000H and the limit is 00FFH. For the 80386 and above, the base address is 00F00000H and the limit is 000FFH. Notice the limit the 80286 has a 16-bit limit and the 80386 through the Pentium II have a 20-bit limit. The 80286 accesses memory segments that are between 1 and 64K bytes in

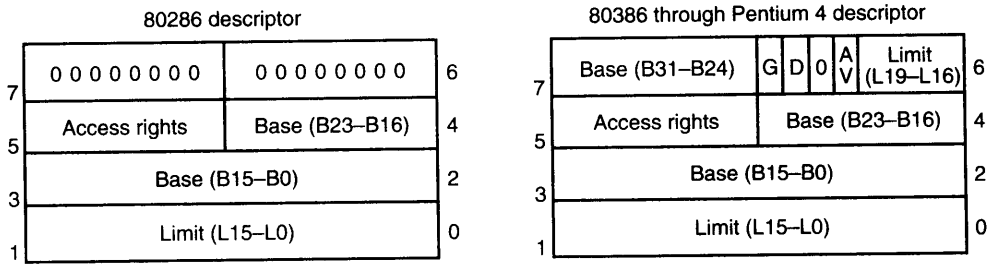


FIGURE 2–6 The descriptor formats for the 80286 and 80386 through Pentium 4 microprocessors.

length. The 80386 and above access memory segments that are between 1 and 1M byte, or 4K and 4G bytes in length.

There is another feature found in the 80386 through the Pentium II descriptor that is not found in the 80286 descriptor: the G bit, or **granularity bit.** If G = 0, the limit specifies a segment limit of 00000H to FFFFFH. If G = 1, the value of the limit is multiplied by 4K bytes (appended with XXXH). The limit is then 00000XXXH to FFFFFXXXH, if G = 1. This allows a segment length of 4K to 4G bytes in steps of 4K bytes. The reason that the segment length is 64K bytes in the 80286 is that the offset address is always 16 bits because of its 16-bit internal architecture. The 80386 and above use a 32-bit architecture that allows an offset address, in the protected mode operation, of the 32 bits. This 32-bit offset address allows segment lengths of 4G bytes and the 16-bit offset address allows segment lengths of 64K bytes. Operating systems operate in a 16- or 32-bit environment. For example, DOS uses a 16-bit environment, while most Windows applications use a 32-bit environment.

Example 2–1 shows the segment start and end if the base address is 10000000H, the limit is 001FFH, and the G bit = 0.

EXAMPLE 2–1

```
Base = Start = 10000000H
G = 0
End = Base + Limit = 10000000H + 001FFH = 100001FFH
```

Example 2–2 uses the same data as Example 2–1, except that the G bit = 1. Notice that the limit is appended with XXXH to determine the ending segment address. The XXXH can be any number between 000H and FFFH. In the example, the XXXH is replaced with FFFH because that is the highest possible memory location within the segment.

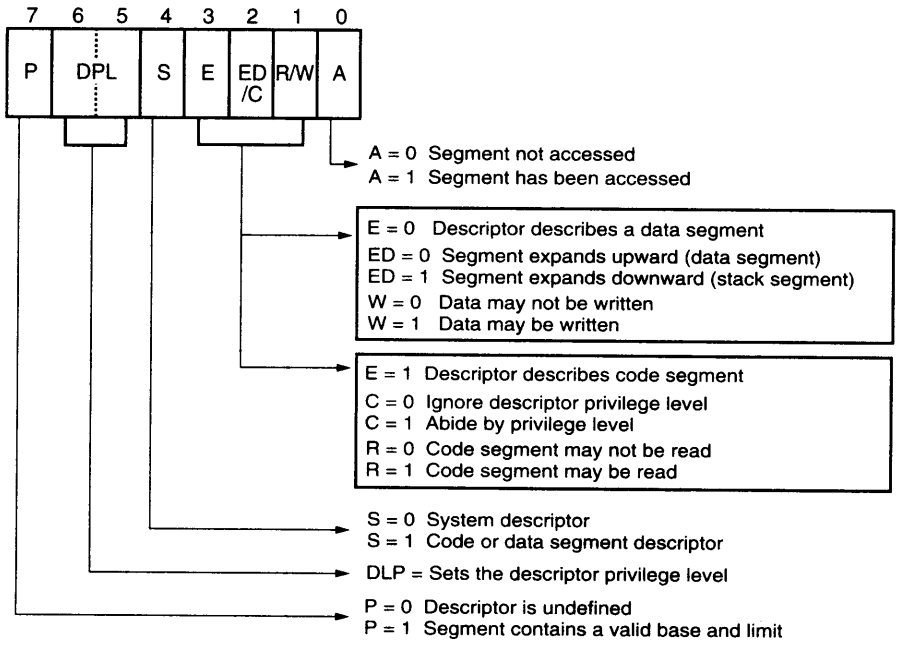
EXAMPLE 2–2

```
Base = Start = 10000000H
G = 1
End = Base + Limit = 10000000H + 001FFXXXH = 101FFFFFH
```

The AV bit, in the 80386 and above descriptor, is used by some operating systems to indicate that the segment is available (AV = 1) or not available (AV = 0). The D bit indicates how the 80386 through the Pentium II instructions access register and memory data in the protected or real mode. If D = 0, the instructions are 16-bit instructions, compatible with the 8086–80286 microprocessors. This means that the instructions use 16-bit offset addresses and 16-bit registers by default. This mode is often called the 16-bit instruction mode. If D = 1, the instructions are 32-bit instructions. By default, the 32-bit instruction mode assumes that all offset addresses and all registers are 32 bits. Note that the default for register size and offset address size can be overridden in both the 16- and 32-bit instruction modes. Both the MSDOS and PCDOS operating systems require that the instructions are always used in the 16-bit instruction mode. Windows 3.1 also requires that the 16-bit instruction mode is selected. Note that the 32-bit instruction mode is accessible only in a protected-mode system such as Windows NT, Windows 95, Windows 98, or OS/2. More detail on these modes and their application to the instruction set appears in Chapters 3 and 4.

The **access rights byte** (see Figure 2–7) controls access to the protected mode memory segment. This byte describes how the segment functions in the system. The access rights byte allows complete control over the segment. If the segment is a data segment, the direction of growth is specified. If the segment grows beyond its limit, the microprocessor's program is interrupted, indicating a general protection fault. You can even specify whether a data segment can be written or is write-protected. The code segment is also controlled in a similar fashion and can have reading inhibited to protect software.

Descriptors are chosen from the descriptor table by the segment register. Figure 2–8 shows how the segment register functions in the protected mode system. The segment register contains a 13-bit selector field, a table selector bit, and a requested privilege level field. The 13-bit selector chooses one of the 8192 descriptors



Note: Some of the letters used to describe the bits in the access rights bytes vary in Intel documentation.

FIGURE 2–7 The access rights byte for the 80286 through Pentium 4 descriptor.

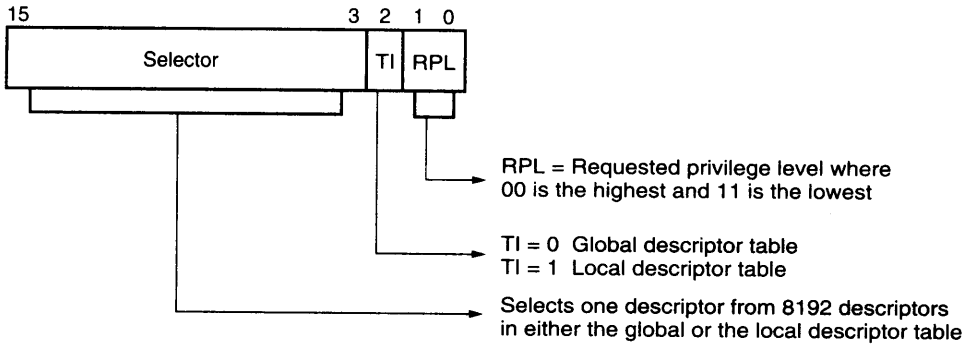


FIGURE 2–8 The contents of a segment register during protected mode operation of the 80286 through Pentium 4 microprocessors.

from the descriptor table. The **TI bit** selects either the global descriptor table (TI = 0) or the local descriptor table (TI = 1). The **requested privilege level** (RPL) requests the access privilege level of a memory segment. The highest privilege level is 00 and the lowest is 11. If the requested privilege level matches or is higher in priority than the privilege level set by the access rights byte, access is granted. For example, if the requested privilege level is 10 and the access rights byte sets the segment privilege level at 11, access is granted because 10 is higher in priority than privilege level 11. Privilege levels are used in multiuser environments. If the privilege level is violated, the system normally indicates a privilege violation.

Figure 2–9 shows how the segment register, containing a selector, chooses a descriptor from the global descriptor table. The entry in the global descriptor table selects a segment in the memory system. In this illustration, DS

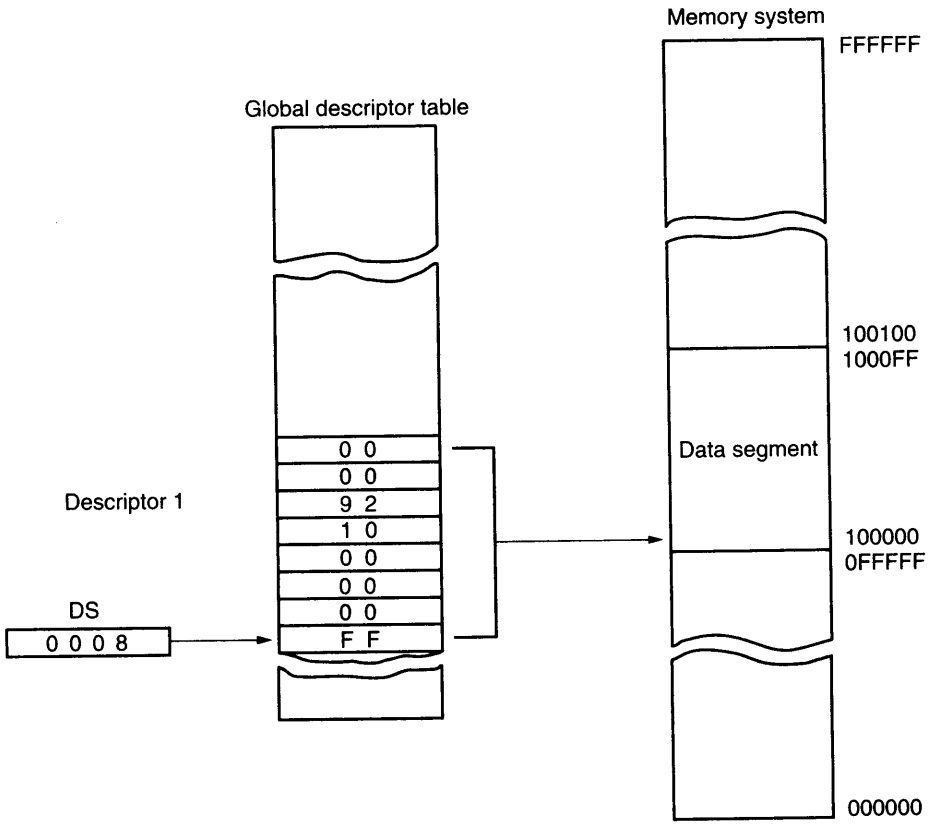


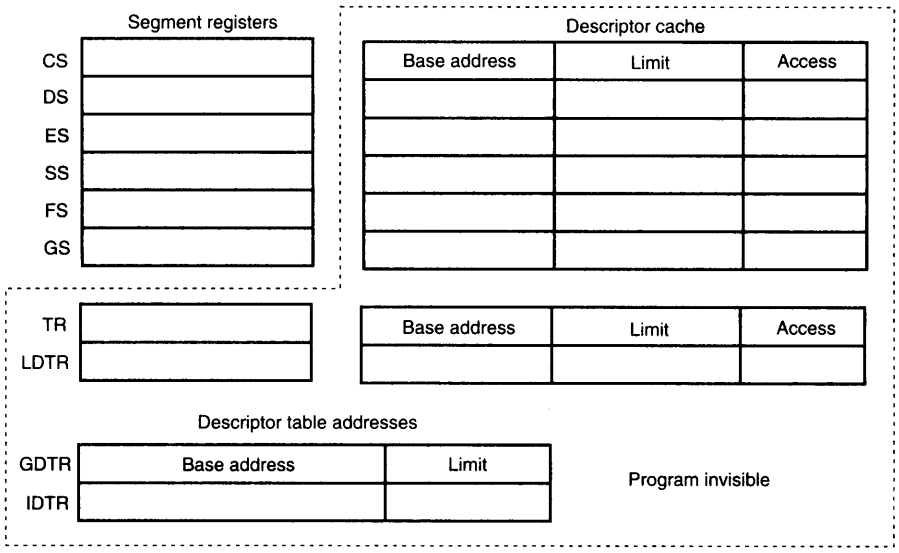
FIGURE 2–9 Using the DS register to select a descriptor from the global descriptor table. In this example, the DS register accesses memory locations 100000H–1000FFH as a data segment.

contains 0008H, which accesses the descriptor number 1 from the global descriptor table by using a requested privilege level of 00. Descriptor number 1 contains a descriptor that defines the base address as 00100000H with a segment limit of 000FFH. This means that a value of 0008H loaded into DS causes the microprocessor to use memory locations 00100000H–001000FFH for the data segment with this example descriptor table. Note that descriptor zero is called the null descriptor and may not be used for accessing memory.

Program-Invisible Registers

The global and local descriptor tables are found in the memory system. In order to access and specify the address of these tables, the 80286, 80386, 80486, Pentium, Pentium Pro, and Pentium 4 contain program-invisible registers. The program-invisible registers are not directly addressed by software so they are given this name (although some of these registers are accessed by the system software). Figure 2–10 illustrates the program-invisible registers as they appear in the 80286 through the Pentium 4. These registers control the microprocessor when operated in the protected mode.

Each of the segment registers contains a program-invisible portion used in the protected mode. The program-invisible portion of these registers is often called cache memory because a cache is any memory that stores information. This cache is not to be confused with the normal level 1 or level 2 caches found with the microprocessor.



Notes:
1. The 80286 does not contain FS and GS nor the program-invisible portions of these registers.
2. The 80286 contains a base address that is 24-bits and a limit that is 16-bits.
3. The 80386/80486/Pentium/Pentium Pro contain a base address that is 32-bits and a limit that is 20-bits.
4. The access rights are 8-bits in the 80286 and 12-bits in the 80386/80486/Pentium.

FIGURE 2–10 The program-invisible register within the 80286–Pentium 4 microprocessors.

- The program-invisible portion of the segment register is loaded with the base address, limit, and access rights each time the number in the segment register is changed. When a new segment number is placed in a segment register, the microprocessor accesses a descriptor table and loads the descriptor into the program-invisible cache portion of the segment register. It is held there and used to access the memory segment until the segment number is again changed. This allows the microprocessor to repeatedly access a memory segment without referring to the descriptor table for each access (hence the term cache).

The GDTR **(global descriptor table register)** and IDTR **(interrupt descriptor table register)** contain the base address of the descriptor table and its limit. The limit of each descriptor table is 16 bits because the maximum table length is 64K bytes. When the protected mode operation is desired, the address of the global descriptor table and its limit are loaded into the GDTR. Before using the protected mode, the interrupt descriptor table and the IDTR must also be initialized. More detail is provided on protected mode operation later in the text. At this point, the programming and additional description of these registers are impossible.

The location of the local descriptor table is selected from the global descriptor table. One of the global descriptors is set up to address the local descriptor table. To access the local descriptor table, the LDTR **(local descriptor table register)** is loaded with a selector, just as a segment register is loaded with a selector. This selector accesses the global descriptor table and loads the base address, limit, and access rights of the local descriptor table into the cache portion of the LDTR.

The TR **(task register)** holds a selector, which accesses a descriptor that defines a task. A task is most often a procedure or application program. The descriptor for the procedure or application program is stored in the global descriptor table, so access can be controlled through the privilege levels. The task register allows a context or task switch in about 17 μs. Task switching allows the microprocessor to switch between tasks in a fairly short amount of time. The task switch allows multitasking systems to switch from one task to another in a simple and orderly fashion.

2–4 MEMORY PAGING

The **memory paging mechanism** located within the 80386 and above allows any physical memory location to be assigned to any linear address. The **linear address** is defined as the address generated by a program. With the memory paging unit, the linear address is invisibly translated into any **physical address,** which allows an application written to function at a specific address to be relocated through the paging mechanism. It also allows memory to be placed into areas where no memory exists. An example is the upper memory blocks provided by EMM386.EXE.

The EMM386.EXE program reassigns extended memory, in 4K blocks, to the system memory between the video BIOS and the system BIOS ROMS for upper memory blocks. Without the paging mechanism, the use of this area of memory is impossible.

Paging Registers

The paging unit is controlled by the contents of the microprocessor's control registers. See Figure 2–11 for the contents of control registers CR0 through CR3. Note that these registers are only available to the 80386 through the Pentium microprocessors. Beginning with the Pentium, an additional control register labeled CR4 controls extensions to the basic architecture provided in the Pentium and above microprocessors. One of these features is a 4M-byte page that is enabled by setting bit position 4, or CR4. Refer to Chapters 16 and 17 for additional details on 4M-byte memory paging.

The registers important to the paging unit are CR0 and CR3. The leftmost bit (PG) position of CR0 selects paging when placed at a logic 1 level. If the PG bit is cleared (0), the linear address generated by the program becomes the physical address used to access memory. If the PG bit is set (1), the linear address is converted to a physical address through the paging mechanism. The paging mechanism functions in both the real and protected modes.

CR3 contains the page directory base address, and the PCD and PWT bits. The PCD and PWT bits control the operation of the PCD and PWT pins on the microprocessor. If PCD is set (1), the PCD pin becomes a logic one during bus cycles that are not pages. This allows the external hardware to control the level 2 cache memory. (Note that the

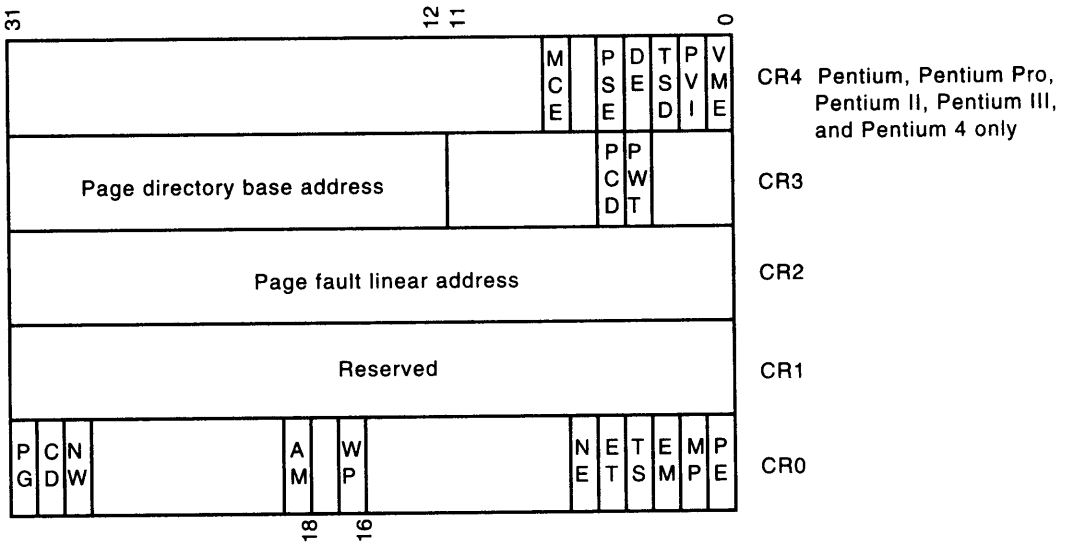


FIGURE 2–11 The control register structure of the microprocessor.

level 2 cache memory is an external high-speed memory that functions as a buffer between the microprocessor and the main DRAM memory system.) The PWT bit also appears on the PWT pin, during bus cycles that are not pages, to control the write-through cache in the system. The page directory base address locates the page directory for the page translation unit. Note that this address locates the page directory at any 4K boundary in the memory system because it is appended internally with a 000H. The page directory contains 1024 directory entries of 4 bytes each. Each page directory entry addresses a page table that contains 1024 entries.

The linear address, as it is generated by the software, is broken into three sections that are used to access the **page directory entry, page table entry,** and **page offset address.** Figure 2–12 shows the linear address and its makeup for paging. Notice how the leftmost 10 bits address an entry in the page directory. For linear address 00000000H–003FFFFFH, the first entry of the page directory is accessed. Each page directory entry represents or repages a 4M-byte section of the memory system. The contents of the page directory select a page table that is indexed by the next 10 bits of the linear address (bit positions 12–21). This means that address 00000000H–00000FFFH selects page directory entry 0 and page table entry 0. Notice this is a 4K-byte address range. The offset part of the linear address (bit positions 0–11) next selects a byte in the 4K-byte memory page. In Figure 2–12, if the page table 0 entry contains address 00100000H, then the physical address is 00100000H–00100FFFH for linear address 00000000H–00000FFFH. This means that when the program accesses a location between 00000000H and 00000FFFH, the microprocessor physically addresses location 00100000H–00100FFFH.

Because the act of repaging a 4K-byte section of memory requires access to the page directory and a page table, which are both located in memory, Intel has incorporated a cache called the TLB **(translation look-aside buffer).** In the 80486 microprocessor, the cache holds the 32 most recent page translation addresses. This means that the last 32 page table translations are stored in the TLB, so if the same area of memory is accessed, the address is already present in the TLB, and access to the page directory and page tables is not required. This speeds program execution. If a translation is not in the TLB, the page directory and page table must be accessed, which requires

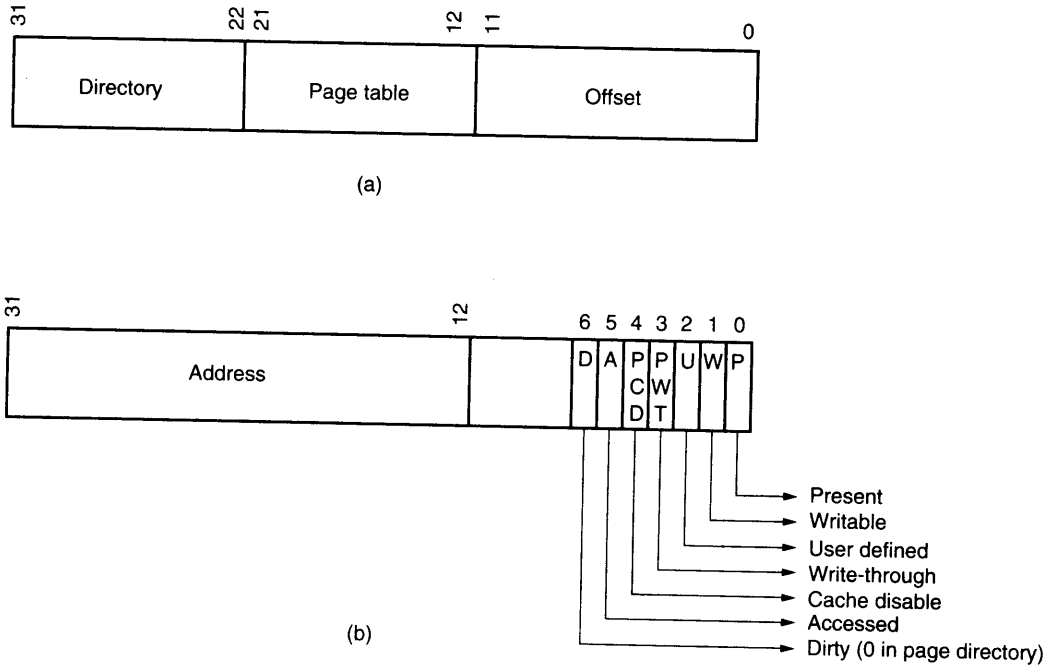


FIGURE 2–12 The format for the linear address (a) and a page directory or page table entry (b).

additional execution time. The Pentium, Pentium Pro, Pentium II, Pentium III, and Pentium 4 contain separate TLBs for each of their instruction and data caches.

The Page Directory and Page Table

Figure 2–13 shows the page directory, a few page tables, and some memory pages. There is only one page directory in the system. The page directory contains 1024 doubleword addresses that locate up to 1024 page tables. The page directory and each page table are 4K bytes in length. If the entire 4G byte of memory is paged, the system must allocate 4K bytes of memory for the page directory, and 4K times 1024 or 4M bytes for the 1024 page tables. This represents a considerable investment in memory resources.

The DOS system and EMM386.EXE use page tables to redefine the area of memory between locations C8000H–EFFFFH as upper memory blocks. It does this by repaging extended memory to back-fill this part of the conventional memory system to allow DOS access to additional memory. Suppose that the EMM386.EXE program allows access to 16M bytes of extended and conventional memory through paging and locations C8000H–EFFFFH must be repaged to locations 110000–138000H, with all other areas of memory paged to their normal locations. Such a scheme is depicted in Figure 2–14.

Here, the page directory contains four entries. Recall that each entry in the page directory corresponds to 4M bytes of physical memory. The system also contains four page tables with 1024 entries each. Recall that each entry in the page table repages 4K bytes of physical memory. This scheme requires a total of 16K of memory for the four page tables and 16 bytes of memory for the page directory.

As with DOS, the Windows program also repages the memory system. At present, Windows version 3.11 supports paging for only 16M bytes of memory because of the amount of memory required to store the page tables. On

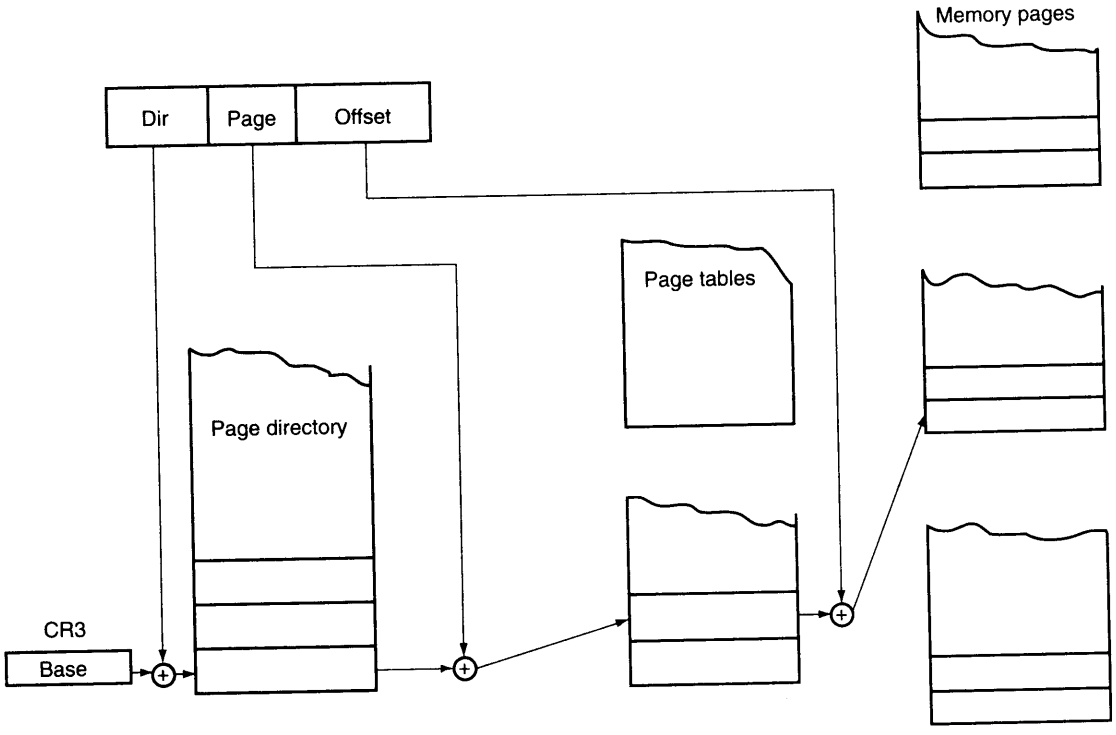


FIGURE 2–13 The paging mechanism in the 80386 through Pentium 4 microprocessors.