260

317

350

419

2;

457

CHAPTER 1

Introduction to Computer Architecture

The PIC (*programmable interface controller*) microcontrollers are a series of RISC (*reduced instruction set computer*) integrated circuits produced by Microchip Technology Incorporated (http://www.microchip.com). Variations are currently available in package sizes of as small as 18 pins and as large as 128 pins. A series of products are produced for the PIC microcontroller called the *BASIC Stamp*® from Parallax, Incorporated (http://www.parallax.com), which is programmed in the BASIC language or the Java language instead of assembly language. This chapter introduces the microcontroller and system architecture as the basis for controlling machinery using the microcontroller. Also defined are many of the terms used with microcontroller technology as an introduction to this fascinating field of study.

Upon completion of this chapter you will be able to:

1. Define the terms used in the field of microcontrollers.
2. Describe the purpose of each component part of a computer system.
3. Relate the operation of each section of a computer system and its interrelation to other system components.
4. Define data types used with microcontrollers.
5. Convert between common number systems.

1-1 BASIC COMPUTER ARCHITECTURE

The most common computer architecture is represented by the block diagram illustrated in Figure 1-1. The first evidence of this architecture was proposed and used by Charles Babbage in his *Analytical Engine* in 1856 (http://www.cbi.umn.edu) and continues as the basic system architecture used in most modern digital computers. The Analytical Engine was a mechanical computer system powered by a hand crank. In more recent years, before the rediscovery of the work of Charles Babbage, the architecture of a computer was attributed to John von Neumann, as he described it in the spring of 1945, and today is often referred to as the *von Neumann architecture*. Chapter 2 explains a different architecture called the *Harvard architecture,* which is not as common as the von Neumann architecture, but used inside the PIC family of microcontrollers.

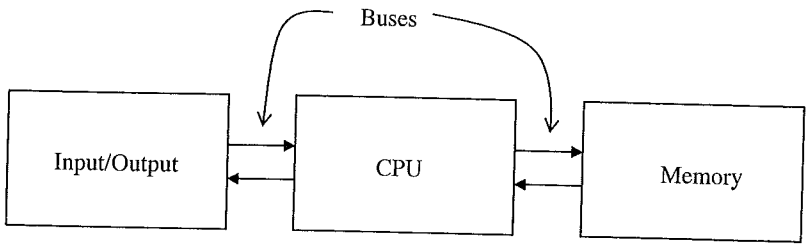

FIGURE 1-1 Block diagram of a computer system.

CPU

The block diagram of a computer system, although simple, contains three main blocks that represent the architecture of most modern digital computer systems. At its core is the CPU (*central processing unit*)—the controlling element of the system. Early CPUs were mechanical systems, such as the analytical engine, calculators, and accounting machines, commonly used into the 1970s. Early electronics digital computers were originally constructed with vacuum tubes and later with transistors. Modern computer systems are constructed with complementary MOSFET (*CMOS*) technology integrated circuits.

Central processors can be of any size and have been anything from 4 bits in width to 64 bits. (A *bit* is a binary digit with the value of zero or one). Most electronic digital computer systems are based on the binary number system because the circuitry, which is currently in use, accurately represents only binary data. Examples of the 4-bit processors are early microprocessors such as the 4004 or 4040 from Intel Corporation. Currently, the latest microprocessors are 64 bits in width from AMD (*Advanced Micro Devices*) and Intel. The size of the CPU is determined by its memory address width and its arithmetic and logic unit, which is responsible for performing arithmetic and logic operations on integer numbers in the system. A 4-bit microprocessor performs these operations on 4-bit integers (called *nibbles*), whereas a 64-bit microprocessor performs them on integers of a maximum size of 64-bits. The 64-bit machines also perform arithmetic and logic on 32-bit, 16-bit, and 8-bit numbers. Today the most common smallest directly addressable number is usually an 8-bit number called a *byte*. A microprocessor or microcontroller can be of any width from 1 bit to any number of bits, although most microprocessors are a multiple of 8 bits and many microcontrollers are either 8 bits or 16 bits.

Another variable in the CPU is its clock speed. The clock speed varies considerably and is currently a few megahertz for microcontrollers to several gigahertz for more powerful microprocessors used in desktop computer systems. A *megahertz* system (MHz) is one based on a clocking frequency of a million pulses per second called Hertz (Hz), whereas a *gigahertz* system (GHz) is one based on a clocking frequency of a billion pulses a second. *Heinrich Rudolf Hertz* (http://www.ideafinder.com/history/inventors/hertz.htm) is honored by name to define the number of pulses per second of an alternating current (originally called cycles per second or cps). Although the clock frequency of a microprocessor or microcontroller may not indicate the number of instructions it executes per second, it directly determines the speed of the system. The exact number of instructions executed per second is based on the internal design of the CPU, the complexity of the instructions that it executes, and the program being executed. One thing is certain: If a 10-MHz machine is compared to a 2-GHz machine, the 2-GHz machine will be the faster of the two machines.

The type of microprocessor can also determine its speed. A RISC (*reduced instruction set computer*) is a machine that executes one instruction per clock and contains only basic instructions. A CISC (*complex instruction set computer*) executes many more different instructions than a RISC machine, but some of the instructions require more than one clock to complete. Today

these terms blur, because most newer microprocessors use a combination of CISC and RISC technologies to perform most instructions in a single clock and some, such as the Pentium® family from Intel, perform many instructions in as little as one-third of a clock. Realize that these newer machines often contain multiple integer units that function simultaneously.

CPU Task. The CPU is responsible for executing a series of sequential instructions organized into a grouping of instructions called a *program*. In fact, the only thing that the CPU is capable of doing is executing instructions from the time it is powered until the time it is switched off. It relentlessly fetches instructions from the memory system and executes them. This *fetch* and *execute* sequence continues for as long as the CPU is powered. The idea of the CPU and the way that it operates, first proposed by Charles Babbage, has not changed since, and will likely remain the same for quite some time. Newer machines execute more than one instruction simultaneously because many have multiple arithmetic and logic processors located within the CPU. Some of the latest machines also have multiple cores. A multiple core machine is a machine that has more than one microprocessor in the integrated circuit.

The program in a computer system is a collection of instructions aligned in a sequential fashion, which is executed by the CPU in a likewise manner. For example, suppose a program is needed to add a 6 to a 2. The first step (or instruction) obtains the number 6. The second obtains the number 2. The third step adds the 6 and 2 together. The final operation stores the sum somewhere in the memory system. As illustrated in this example, the simple task of adding 6 and 2 together is broken down into steps called *instructions*. These instructions form a program that is executed by the microprocessor or microcontroller.

1. Get a 6.
2. Get a 2.
3. Add 6 and 2.
4. Store the sum in memory.

Each step of this simple program is converted to the language that the CPU understands, called *machine language*. Machine language is a series of numbers that represent instructions. In a hypothetical CPU, the "get instruction" might be the number 00, the add instruction might be the number 01, and the store instruction might be the number 02. The program, in numeric machine language, appears as 00 06 00 02 01 02 00, or as shown in Example 1-1.

EXAMPLE 1-1

```
00 06
00 02
01
02 00
```

As can be seen in this example, writing numeric machine code is cumbersome and cryptic. Luckily, no one writes programs in numeric machine code, although they did in the early days of computing. Instead, a tool called an *assembler* is used to write the program in its symbolic form using *mnemonic opcodes* such as GET and ADD. The assembler, which is also a program, converts the symbolic assembly language program into numeric machine code. An assembly language program that generates the program in Example 1-1 might appear as shown in Example 1-2.

EXAMPLE 1-2

```
GET 6
GET 2
ADD
STORE 0
```

Even this is cumbersome, but at least it is more readable than a series of numbers. Because an assembly language program can be quite long and challenging to write, modern programming typically uses a *high-level language* to generate the machine code. A high-level language is a program that accepts as its input a pseudo-English-like language and then converts it into numeric machine code. Some of these languages are BASIC, C, C++, or Java. The same program might appear in C language as depicted in Example 1-3.

EXAMPLE 1-3

```
char answer;            //set aside a place for the answer
answer = 6 + 2;         //perform the addition and store sum in answer
```

As seen from this C program, there is less typing, it requires no knowledge of machine code, or numeric memory locations, and is fairly easy to read. The machine code generated by the C language program may not be as efficient as the code generated by an assembler, but it is much easier to write, understand, maintain, and requires less time and skill to write. It is for these reasons that software is often developed using a high-level language. The *software* of a computer system is its programs and the *hardware* is the computer system circuitry. At times software is referred to as *variable hardware* because it changes or modifies the way the hardware functions. This is especially true in a microcontroller-based system.

It is interesting to note that Augusta Ada Byron (later named the Countess of Lovelace after a marriage to Lord Byron) earned the reputation for being the first computer programmer. She earned this distinction because of the programs that she wrote in numeric machine language for Charles Babbage and his Analytical Engine. A computer language called Ada, which is used by the U.S. Department of Defense, is named in her honor.

CPU Function. A CPU performs three main tasks in a computer system: (1) data transfer, (2) arithmetic and logic, and (3) program flow control. Data transfers represent the most common CPU task. Most of a CPU's time is spent transferring data. Data transfers include fetching an instruction from memory, transferring data between registers or memory locations, and transferring the result from some arithmetic or logic operation between memory and the CPU or between an I/O (*input/output*) device and the CPU. The instruction fetch portion of a data transfer operation is the most common and important operation performed by the CPU. The instruction fetch allows the CPU to execute the program from the memory system at a high rate of speed. This *stored program* concept makes the computer very powerful and was first envisioned by Charles Babbage. Most programs transfer data for at least 50% of the time and often a much higher percentage. Table 1-1 lists some of the most common data transfer operations performed by most CPUs.

A CPU spends much less time performing arithmetic and logic operations on numbers than data transfers. Often a program requires that the CPU devotes a small percentage of its time to

TABLE 1-1 Most common data transfer operations.

Operation	*Comment*
Memory → Opcode Register	Fetch an operation to opcode register
Memory → Data Register	Fetch memory data to a data register
Data Register → Memory	Store register data in a memory location
Data Register → Data Register	Transfer contents of a data register to another data register
Data Register → I/O	Transfer contents of a data register to an I/O device
I/O → Data Register	Transfer contents of an I/O device to a data register

TABLE 1-2 Common arithmetic and logic operations.

Operation	*Comment*
Addition	Between registers, a register and memory, or with immediate data
Subtraction	Between registers, a register and memory, or with immediate data
Multiplication	Between registers, a register and memory, or with immediate data; result can be double the size
Division	Between registers, a register and memory, or with immediate data; dividend can be double the size (some microcontrollers do not have division)
Negation	The sign of a number is changed
AND	Logical bit-wise AND between registers, a register and memory, or with immediate data
OR	Logical bit-wise OR between registers, a register and memory, or with immediate data
Exclusive-OR	Logical bit-wise Exclusive-OR between registers, a register and memory, or with immediate data
NOT	Logical bit-wise NOT between registers, a register and memory, or with immediate data

arithmetic and logic operations. Table 1-2 lists typical arithmetic and logic operations performed by the CPU. These operations are normally performed by an integer arithmetic unit. Real numbers cannot be operated on by the CPU without a program or special processing unit called an *arithmetic unit* or *numeric processor*. Microprocessors often contain a numeric processor as well as the integer processor, whereas the microcontroller seldom contains a numeric processor.

The most powerful function performed by the CPU is its ability to modify the flow of a program through the use of simple numeric decisions. *Program flow control* instructions allow a section or sections of code to be used multiple times in a program. Program flow control instructions also allow the flow of a program to be transferred to a function or procedure. There are unconditional and conditional program flow control instructions. The unconditional program flow control instructions are the GOTO and function CALL instructions. The conditional program flow control instructions allow a number to be tested for a condition to determine if the flow of the program is modified.

Conditional program flow control instructions test a number to determine if it's zero, if a carry occurred after addition, or if the result of an operation is negative or positive. Table 1-3 lists many of the commonly testable conditions in most CPUs. The conditional program flow instructions represent a small percentage of the instructions in a typical program, yet these instructions are what make a CPU a powerful component in a computer system with the perceived ability to think. For example, how can a computer determine if the number typed on a keyboard is a 3? It does this by subtracting a 3 from the number typed and then it tests to see if the result is zero. If the result it is zero, the flow of the program is modified to perform some task in response to the 3 that was typed. The illusion here is that the computer is able to think and reason that a

TABLE 1-3 Conditions tested by many CPUs.

Condition	*Comment*
Zero	Is a number zero or not zero?
Carry	Did a carry occur or no carry?
Sign	Is a number positive or negative?
Overflow	Some CPUs test for an arithmetic overflow.

3 was typed on the keyboard, when in reality all that was done was to subtract a 3 from the number that was typed and test the result for a zero. What makes the computer system a powerful tool? The software, which is written by programmers, is the key because it makes the computer appear to think and reason.

Memory

The *memory* is an extremely important component of a computer system because it provides a place to store programs and the data used by a program. Without the memory, a computer system would be no more powerful or faster than a simple four-function calculator. With memory, the computer accesses the instructions in a program from the memory at a very high rate of speed. Incidentally, Charles Babbage's Analytical Engine contained memory that stored 1000 numbers that were each 20 decimal digits wide. It too executed a program stored in its memory.

Two types of memory are used with microprocessors and microcontrollers: *read-only memory* (ROM) and read/write memory, typically called *random access memory* (RAM). The ROM in a microcontroller-based system stores static data as a program and as constant data used by the program. Most embedded microcontroller-based or microcontroller-based systems store only one program in the memory that functions as an operating system. The RAM in a microcontroller-based system stores dynamic data for the system. It is important to note that a disk drive and similar devices are not considered memory, but are considered I/O devices even though they store programs and data. The distinction is made because the disk drive and similar devices are treated differently when connected to the system. The *driver* (a program that controls an I/O device) for a disk drive must send quite a bit of information to the disk drive to cause it to operate. The difference is that the ROM or RAM memory is not accessed by a program, but by direct connection through a series of wires to the CPU and is directly controlled by the signals that emanate from the CPU.

ROM. The ROM in a system is an *erasable/programmable read-only memory* (EPROM), an *electrically erasable/programmable read-only memory* (EEPROM or E^2PROM), or occasionally, a factory-programmed read-only memory. Both the EPROM and EEPROM store data for approximately 20 years or until erased. The factory-programmed ROM is permanent. The EPROM can be erased up to approximately 100 times, whereas the EEPROM can be erased from 10,000 to 1,000,000 times, depending on its manufacturer and when it was manufactured.

The EPROM has been in use for many years and must be removed from the system to erase. The EPROM is erased by placing it under a high-intensity ultraviolet light for anywhere from 10 to 20 minutes. EPROM memory is available with speeds of up to about 100 ns for a read access. The EEPROM is erased electrically without removing it from the system. EEPROM memory is often called *Flash memory* and also has an access time of about 100 ns. The main difference is that the erase time for an EEPROM is much shorter than for the EPROM. Erase times are typically in the milliseconds for the EEPROM and minutes for the EPROM. Recent common applications for the EEPROM include the system BIOS in a personal computer; USB flash drives (called *JumpDrives*™, *PocketDrives*™, *PenDrives*™, or *ThumbDrives*™), that have replaced the floppy disk drive in most computer systems; and MP3 audio players.

RAM. The RAM in a system is either SRAM (*static RAM*) or DRAM (*dynamic RAM*). Both memory types are *volatile memory,* which fail to retain data after removed from power. The SRAM stores data in memory cells that are continuously powered; this makes them fast, but consumes a fair degree of power. The SRAM devices have access times of 1 ns or less, which makes them excellent candidates for computer system memory except for the power consumption. The DRAM stores data in a memory cell that is a capacitor. Because capacitors retain a charge for an indefinite time, a DRAM must be periodically rewritten to assure the integrity of the data. The process of periodically rewriting the data to a DRAM is called a *refresh*. A DRAM usually

retains data for 2 to 4 ms before it must be refreshed. Refreshing a DRAM requires time from the system. Also, data cannot be stored and retrieved at nearly the same speed as an SRAM. Access times for the DRAM are around 40 to 50 ns, whereas access times for the SRAM are 1 ns or less. Today we use DRAM for large memory systems and SRAM for small systems. This means that most microcontrollers do not use DRAM memory because the memory size is small. The technology is changing, and it is hoped that soon SRAM power consumption will be reduced enough to use SRAM for large memory systems, which should yield a 50-time boost in the speed of system memory.

I/O

The *input/output* (I/O) in a system is a CPU's connection to other machines and humans. Without I/O, the CPU is able to solve problems, but it cannot provide the result to humans or other machines. Because of the diversity of I/O devices connected to a CPU, much of the work with microprocessors or microcontrollers involves interfacing and controlling, through software drivers, the I/O devices in a computer system. Today, I/O devices perform just about any task or function imaginable. Much of this text is devoted to interfacing the microcontroller to many of these I/O devices and providing software drivers to control them. Some common I/O devices are keyboards for inputting information and LCD or video displays for outputting data. In general, anything that produces or accepts an electrical signal can be and often is interfaced to the computer as an I/O device.

Buses

The connections between the blocks in the block diagram of a computer system are called buses. A *bus* is a common set of wires that carry a specific type of information. A computer system has three buses: (1) address, (2) data, and (3) control. The address bus carries address information for selecting a memory location or an I/O location, the data bus carries the data that is transferred between the memory or I/O and the microcontroller, and the control bus controls whether the memory and I/O is read or written. It is through these three buses that the microcontroller is able to execute programs from the memory system and control the I/O devices attached to the system.

Address Bus. All computers have an *address bus* with signal lines that are almost always labeled A0, A1, A2, and so forth. The A0 signal is the least significant address bus bit position and the A1 signal is the next to the least significant address bus bit, and so forth. The number of address bus pins or signals varies between computers. For example, if a computer has a 16-bit address bus (numbered from A0 to A15), it addresses 64K of memory or 64 × 1,024 (65,536) locations in the memory system. A *computer K* (pronounced *kay*) is 1,024, or 2^{10}. We know this, because the number of locations is 2 raised to the number of address pins in a system, or 2^{16} in a system with a 16-bit address bus. This is the number of binary combinations (64K) for a 16-bit address. Likewise, a system that contains a 20-bit address bus addresses 1M (pronounced *1 Meg*) memory locations (2^{20}). A 1M memory contains 1K times 1K locations or 1,048,576 locations. Memory addresses range in size from 12 bits (4K) on some microcontrollers to as many as 40 bits (1T, or *1 Tera*) on the latest Pentium 64-bit microprocessors (EMT-64) from Intel Corporation. Table 1-4 lists the address and memory sizes for the microprocessors and microcontrollers.

Memory is numbered in hexadecimal (*number base* or *radix 16*) from location zero to the maximum number for the address width. For example, a 4K memory, which has a 12-bit address, is numbered from memory location 000 to FFF in hexadecimal. It contains 1000 hexadecimal locations. Likewise a 64K memory, which has a 16-bit address, is addressed at locations 0000 to FFFF hexadecimal and contains 10,000 hexadecimal locations. A 4-bit binary number is required to represent each hexadecimal digit of the address. Hexadecimal numbers are often denoted by using a 0x in front of the number or the letter H following the number. Examples are 0x3A and 3AH.

TABLE 1-4 Address and memory sizes.

Address Size	*Memory Capacity*	*Capacity in Hexadecimal*	*Note*
10 bits	1K (1,024) or 2^{10}	400	Kilo Kibi
12 bits	4K (4,096) or 2^{12}	1000	
16 bits	64K (65,536) or 2^{16}	1 0000	
20 bits	1M (1,048,576) or 2^{20}	10 0000	Meg Mebi
30 bits	1G (1,073,741,824) or 2^{30}	4000 0000	Gig Gibi
40 bits	1T (1,099,511,627,766) or 2^{40}	100 0000 0000	Tera Tebi
50 bits	1P (1K * 1T) or 2^{50}	4 0000 0000 0000	Peta Pebi
60 bits	IE (1K * 1P) or 2^{60}	1000 0000 0000 0000	Exa Exbi

Note: Kibi, Mebi, Gibi, Tebi, Pebi, and Exbi are proposed names.

Data Bus. The *data bus* in a computer system is usually a 2-way bus or a bidirectional bus that carries data between the CPU and the memory and I/O system. The width of the data bus, which is numbered from D0 to Dn, varies. As with the memory address, the zero bit (D0) is the least significant data bus bit. An 8-bit computer usually has an 8-bit data bus, a 16-bit computer usually has a 16-bit data bus, and so forth. In some machines, the data bus is twice the width of the CPU to accomplish data transfers at higher speeds. The Pentium microprocessor from Intel is a 32-bit machine that uses a 64-bit data bus.

Control Bus. Memory and I/O connected to the microprocessor need to be controlled. The *control bus* performs this task through a *read signal* often called $\overline{RD}$ and a *write signal* often called $\overline{WR}$. Note that the overbar indicates a signal is an active low signal. For example, the $\overline{RD}$ (read) signal causes a read when it becomes a logic zero. At times the # symbol is used to indicate that a signal is active low, as in #RD. If the microprocessor fetches an instruction from memory, it places the memory address on the address bus, the read signal is forced low to inform the memory to do a read operation, and the data are then transferred from the memory to the microprocessor through the data bus. Likewise, the same steps are performed for a write, except that the microprocessor issues a write signal in place of the read signal.

Any other control bus signal is specific to a particular microprocessor. Some of the additional control bus signals provide interrupt inputs and various other control functions besides memory and direct I/O control to the computer system.

Microprocessor and Microcontroller

Charles Babbage built his computer from the technology of his time, which was mechanical. His machine was constructed with gears and levers using odometer-style mechanisms to store 1000, 20-digit decimal numbers in its memory. I/O was accomplished through holes punched on a paper tape. These early ideas of mechanical computing were carried forward for quite some time until more modern computers first appeared in the twentieth century.

The IBM Corporation built a vast industry and empire based on punched cards that survived in computing applications well into the 1980s. These punched cards (called *Hollerith cards* named for Herman Hollerith, the founder of IBM) were mechanical devices that functioned as input/output media to mechanical computers in the 1950s and then electronic computers in the 1960s. The early

mechanical computers were often called *accounting machines* and were programmed through a series of jumper wires. These early machines gave way to electronic computers, which were often called *mainframes*. The mainframe computers were expensive and large, and with the advent of modern integrated circuits, became obsolete when the microprocessor overtook the mainframe beginning in the 1980s.

The microprocessor (4004) was invented by Intel in 1971 as a 4-bit CPU that operated with a 20-KHz clock and addressed a 4-bit-wide memory system with a 12-bit memory address (4K). The technology progressed quickly and, by the end of the 1970s, the size of the microprocessor had become 16 bits at Intel with a 1M-byte memory and a 6-MHz clock (the 8086 microprocessor). Motorola also was producing microprocessors and it too had advanced the technology with a 32-bit microprocessor that addressed a 16M-byte memory using an 8-MHz clock (the 68000 microprocessor). Clearly it appears that Motorola had the advantage, yet it did not become dominant. This probably occurred because IBM decided to use the Intel microprocessor in its personal computer. The architecture, and Intel, boomed because of the IBM moniker, and Motorola processors were relegated to the Apple Macintosh computer, which did not enjoy the success of the personal computer from IBM. Recently Apple computer announced that it is using the Pentium microprocessor in its new line of Apple computers.

The *microcontroller,* which is a self-contained computer system, also appeared at Intel in 1977 as an 8-bit machine (8048) that contained a microprocessor, memory, and I/O connections. The microcontroller was designed to control machinery and the microprocessor was designed as a replacement for the CPU in a mainframe. The first major applications of the microcontroller were in the Epson FX-80 dot matrix printer and in the keyboard on an IBM personal computer. The printer was built around the 8048 microcontroller and brought low-cost printers to the marketplace for the first time. The keyboard contained a universal peripheral interface (UPI), the 8042, which appeared in 1977 as a microcontroller used to read keystrokes from the computer user. This gave rise to the term *embedded controller,* or *embedded microprocessor,* because the computer system was hidden or embedded in another product, such as a printer. From outward appearances the FX-80 was a printer, even though internally it was a computer system. The same is true for the keyboard; from all appearances the keyboard was a keyboard, but internally it was a computer system.

When Intel abandoned the microcontroller and universal peripheral interface in favor of the microprocessor in the late 1980s, Microchip Incorporated was formed to fill a void in the microcontroller arena in 1989. Since its inception, Microchip has become the leading provider of 8-bit microcontroller technology. Even the name of their device, a PIC (peripheral interface controller) was initiated by Intel years earlier as the closely named universal peripheral interface (UPI) in the 8042.

1-2 NUMBER SYSTEMS

The use of the microprocessor or microcontroller requires a working knowledge of binary, decimal, and hexadecimal numbering systems. This section provides a background for those who are unfamiliar with these numbering systems. Conversions between decimal and binary, decimal and hexadecimal, and binary and hexadecimal are described.

Digits

Before numbers are converted from one number base to another, the digits of a number system must be understood. Early in our education, it was learned that a decimal (*base 10*) number is constructed with 10 digits: 0 through 9. The first digit in any numbering system is always zero.

For example, a base 8 (octal) number contains 8 digits: 0 through 7; a base 2 (binary) numb contains 2 digits: 0 and 1. If the base of a number exceeds 10, the additional digits use the le ters of the alphabet, beginning with an A. For example, a base 12 number contains digits through 9, followed by A for 10 and B for 11. Note that a base 10 number does contain a 1 digit, just as a base 8 number does not contain an 8 digit. The most common numbering sy tems used with modern computers are decimal, binary, and hexadecimal (base 16). (Man years ago octal numbers were popular.) Each of these number systems are described and use in this section.

Positional Notation

Once the digits of a number system are understood, larger numbers are constructed by using po sitional notation. In elementary school, it was learned that the position to the left of the units po sition is the tens position, the position to the left of the tens position is the hundreds position, an forth. An example is the decimal number 132: This number has 1 hundred, 3 tens, and 2 units What probably was not learned in elementary school was the exponential value of each position The unit's position has a weight of 10^0, or 1; the tens position has a weight of 10^1, or 10; and th hundreds position has a weight of 10^2, or 100. The exponential powers of the positions are crit ical for understanding numbers in other numbering systems. The position to the left of the radi (*number base*) point, called a decimal point only in the decimal system, is always the units posi tion 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 zer power is always 1, or the unit's position.

The position to the left of the units position is always the number base raised to the firs power; in a decimal system, this is 10^1, or 10. In a binary system, it is 2^1, or 2; and in an octa system, it is 8^1, or 8. Therefore, an 11 decimal has a different value or number of units than a 11 binary. The decimal number is composed of 1 ten plus 1 unit, and has a value of 11 units; whereas the binary number 11 is composed of 1 two plus 1 unit, for a value of 3 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 1-4 shows 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 weights of each digit to form its decimal equivalent. The 110.101_2 is equivalent to a 6.625 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 fractional part is composed of 2^{-1} (.5) plus 2^{-3} (or .125), but there is no digit under the 2^{-2} (or .25) so .25 is not added.

EXAMPLE 1-4

Power	2^2		2^1		2^0		2^{-1}		2^{-2}		2^{-3}											
Weight	4		2		1		.5		.25		.125											
Number	1		1		0	.	1		0		1											
Numeric Value	4	+	2	+	0	+	.5	+	0	+	.125	=	6.625									

Suppose that the conversion technique is applied to a base 6 number, such as 25.2_6. Example 1-5 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 (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 ($2 \times .167$). Therefore the number 25.2_6 has a value of 17.333 in decimal.

EXAMPLE 1-5

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 1-6. The value of 125.7_8 is 85.875 decimal, or 1×64 plus 2×8 plus 5×1 plus $7 \times .125$.

EXAMPLE 1-6

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 $\frac{1}{8}$, or .125. The next position is $\frac{.125}{8}$, or .015625, which can also be written as $\frac{1}{64}$. Also note that the number in Example 1-6 can be written as the decimal number $85\frac{7}{8}$.

Example 1-7 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 1-7

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	.5	.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 $\frac{1}{2}$, 2^{-2} is $\frac{1}{4}$, and so forth. It is also interesting to note that 2^{-4} is $\frac{1}{16}$, or .0625. The fractional part of this number is $\frac{7}{16}$, or .4375 decimal. Notice that 0111 is a 7 in binary code for the numerator and the rightmost 1 is in the $\frac{1}{16}$ position for the denominator. Other examples: the binary fraction of .101 is $\frac{5}{8}$ and the binary fraction of .001101 is $\frac{13}{64}$.

Hexadecimal numbers are often used with computers. The 6A.C is illustrated with its weights in Example 1-8. The sum of its digits is 106.75, or $106\frac{3}{4}$. 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 $\frac{12}{16}$, which is reduced to $\frac{3}{4}$.

EXAMPLE 1-8

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 is as 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 unit's 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 1-9 shows this conversion process. The result is written as 1010_2 from the bottom to the top.

EXAMPLE 1-9

```
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 1-10. A 10 decimal is a 12 octal.

EXAMPLE 1-10

```
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 through 15 is then converted to the letters A through F for the hexadecimal number. Example 1-11 shows the decimal number 109 converted to a 6D hexadecimal.

EXAMPLE 1-11

```
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 (repetend). 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 is zero.

Suppose that a .125 decimal is converted to binary. This is accomplished with multiplications by 2, as illustrated in Example 1-12. 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 1-12

```
  .125
x    2
  0.25        digit is 0

   .25
x    2
   0.5        digit is 0

    .5
x    2
   1.0        digit is 1          result = 0.001₂
```

This same technique is used to convert a decimal fraction into any number base. Example 1-13 shows the same decimal fraction of .125 from Example 1-12 converted to octal by multiplying by 8.

EXAMPLE 1-13

```
  .125
x    8
   1.0        digit is 1          result = 0.1₈
```

Conversion to a hexadecimal fraction appears in Example 1-14. Here, a decimal .046875 is converted to hexadecimal by multiplying by 16. Note that a .046875 is a 0.0C in hexadecimal.

EXAMPLE 1-14

```
  .046875
x      16
    0.75      digit is 0

   .75
x   16
  12.0        digit is 12(C)      result = 0.0C₁₆
```

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 1-5. Note that we often represent a hexadecimal number as 0x8A where the 0x is a signal to the computer that the number that follows is hexadecimal.

Hexadecimal numbers are represented in BCH code by converting each digit to BCH code with a space between each coded digit. Example 1-15 shows 2AC converted to BCH code. Note that each BCH digit is separated by a space.

TABLE 1-5 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 1-15

```
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 1-16 shows a BCH coded number converted back to hexadecimal code.

EXAMPLE 1-16

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

Complements

At times, data are stored in complement form to represent negative numbers. Two systems 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 1-17 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 1-18, in which the fifteen's (radix − 1) complement of a 5CD hexadecimal is computed by subtracting each digit from a fifteen.

EXAMPLE 1-17

```
  1111 1111
- 0100 1100
  1011 0011
```

EXAMPLE 1-18

```
  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 modem 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 1-19 shows how the number 0100 1000 is converted to a negative value by two's (radix) complementing it.

EXAMPLE 1-19

```
  1111 1111
- 0100 1000
  1011 0111   (one's complement)
+        1
  1011 1000   (two's complement)
```

To prove that 0100 1000 is the inverse (negative) of 1011 1000, add them together to form an 8-digit result. The ninth digit is dropped and the result is zero because 0100 1000 is a positive 72, whereas 1011 1000 is a negative 72. The same technique applies to any number system. Example 1-20 shows how the inverse of a 345 hexadecimal is found first by 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 complements, of one's and two's is presented with signed numbers in the next section.

EXAMPLE 1-20

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

1-3 COMPUTER DATA FORMATS

Successful programming requires a precise understanding of data formats. This section describes many common computer data formats as they are used with the PIC family of microcontrollers. Commonly, data appear as ASCII, BCD, signed and unsigned integers, and occasionally as 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 1-6). The standard ASCII code is a 7-bit code, with the eighth and most significant bit used to hold parity in some antiquated systems. If ASCII data are used with a printer, the most significant bits are 0 for alphanumeric printing and 1 for graphics printing. In the personal computer, an extended ASCII character set is selected by placing a 1 in the left-most bit. Table 1-7 shows the extended ASCII character set, using code 0x80–0xFF. The extended ASCII characters store some foreign letters and punctuation, Greek

TABLE 1-6 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	{	|	}	~	⌂

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 used with the IBM ProPrinter, which also matches the special character set found with most word processors.

The ASCII control characters, also listed in Table 1-6, perform control functions in a computer system, including clear screen, backspace, line-feed, and so forth. To enter the control codes through the computer keyboard, hold down the Control key while typing a letter. To obtain the control code 0x01, type a Control-A; a 0x02 is obtained by a Control-B, and so forth. Note that the control codes often appear on the screen 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 modem 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 1-6 or 1-7 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 0x41 and the lowercase letter *a* is ASCII code 0x61.

ASCII data are most often stored in memory by using a special directive to the assembler program called *declare byte,* or DB, or by the DATA directive for strings. (The assembler is a program that is used to program a computer in its native binary machine language.) The DB

TABLE 1-7 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	¢	£	¥	P_t	ƒ
AX	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
BX	▒	▒	▒	╢	┤	╡	╣	╖	╕	╜	╝	╗	╝	╜	┘	┐
CX	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
DX	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	▀	▄	▌	▐	▀
EX	α	ß	Γ	π	Σ	σ	µ	γ	Φ	Θ	Ω	δ	∞	φ	∈	∩
FX	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	^{n}	^{2}	■	

directive and the DATA directive, and several examples of their usage with ASCII-coded characters, are listed in Example 1-21. Notice how each character using DB is surrounded by apostrophes (')—never use the quote (") unless you are using DATA.

EXAMPLE 1-21

```
DB    'B'
DB    'r'
DB    'e'
DB    'y'
DB    0x00
DB    'Brey', 0

DATA  "Brey", 0
DATA  "Barry", 0
```

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 0000_2 to 1001_2, 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 1-8 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 other 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 1-22 shows how to use the assembler to define both packed and unpacked BCD data. Example 1-23 shows how to do this using C/C++ and char or bytes. In all cases, the convention of storing the least-significant data first is followed. This means that to store 83 into memory, the 3 is stored first, followed by the 8. Also note that with packed BCD data, the 0x (hexadecimal) precedes 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, two digits per byte.

EXAMPLE 1-22

```
;Unpacked BCD data (least-significant data first)
;
NUMB1  DB    3,4,5          ;defines number 543
NUMB2  DB    7,8            ;defines number 87
                            ;
;Packed BCD data (least-significant data first)
                            ;
NUMB3  DB    0x37,0x34      ;defines number 3437
NUMB4  DB    3,0x45         ;defines number 4503
```

TABLE 1-8 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

3. Who was the Countess of Lovelace?
4. What is a von Neumann machine?
5. What company developed the first microcontroller?
6. What is the acronym CISC?
7. What is the acronym RISC?
8. A binary bit stores a(n) _____ or a(n) _____.
9. A computer K is equal to _____ bytes.
10. A computer M is equal to _____ K bytes.
11. A computer G is equal to _____ M bytes.
12. What is a nibble?
13. Draw the block diagram of a computer system.
14. List the three buses found in all computer systems.
15. Which bus transfers the memory address to the I/O device or to the memory?
16. Which control signal causes the memory to perform a read operation?
17. What is the stored program concept?
18. What is the difference between an EPROM and an EEPROM?
19. If a memory has a 14-bit address, how many locations does it contain?
20. How many memory locations are found in a 4K $\times$ 8 memory?
21. What is an SRAM?
22. What type of memory device would be used to store dynamic data?
23. Define the purpose of the following assembler directives:
    a. DB
    b. DATA
    c. DW
24. Define the purpose of the following C/C++ directives:
    a. char
    b. short
    c. int
    d. float
    e. double
25. If a memory address bus contains the following signal lines, determine the number of memory locations that can be addressed.
    a. 12
    b. 14
    c. 16
    d. 18
    e. 32
26. Convert the following binary numbers into decimal:
    a. 1101.01
    b. 111001.0011
    c. 101011.0101
    d. 111.0001
27. Convert the following octal numbers into decimal:
    a. 234.5
    b. 12.3
    c. 7767.07
    d. 123.45
    e. 72.72
28. Convert the following hexadecimal numbers into decimal:
    a. A3.3
    b. 129.C

c. AC.DC
d. FAB.3
e. BB8.0D

29. Convert the following decimal integers into binary, octal, and hexadecimal:
    a. 23
    b. 107
    c. 1238
    d. 92
    e. 173
30. Convert the following decimal numbers into binary, octal, and hexadecimal:
    a. 0.625
    b. 0.00390625
    c. 0.62890625
    d. 0.75
    e. 0.9375
31. Convert the following hexadecimal numbers into binary-coded hexadecimal code (BCH):
    a. 23
    b. AD4
    c. 34.AD
    d. BD32
    e. 234.3
32. 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
33. Convert the following binary numbers to the one's complement form:
    a. 1000 1000
    b. 0101 1010
    c. 0111 0111
    d. 1000 0000
34. Convert the following binary numbers to the two's complement form:
    a. 1000 0001
    b. 1010 1100
    c. 1010 1111
    d. 1000 0000
35. How is a byte declared using the assembler?
36. Convert the following words into ASCII-coded character strings using the assembler:
    a. FROG
    b. Arc
    c. Water
    d. Well
37. What is the ASCII code for the Enter key and what is its purpose?
38. Use an assembler directive to store the ASCII-character string 'What time is it?' in the memory.
39. Convert the following decimal numbers into 8-bit signed binary numbers:
    a. +32
    b. −12
    c. +100
    d. −92