Chapter 1

Introduction

In our modern society Electronic Digital Computer Systems, commonly referred to as **computer systems** or **computers**, are everywhere. We find them in offices, factories, hospitals, schools, stores, libraries, and now in many homes. Computers show up in sometimes unexpected places – in your car, your television and your microwave, for example. We use computers to perform tasks in science, engineering, medicine, business, government, education, entertainment, and many other human endeavors. Computers are in demand wherever complex and/or high speed tasks are to be performed.

Computers have become indispensable tools of modern society. They work at high speed, are able to handle large amounts of data with great accuracy, and have the ability to carry out a specified sequence of operations, i.e. a **program** without human intervention and are able to change from one program to another on command.

Computer systems are general purpose *information processing machines* used to solve problems. Solving these problems may involve processing information (i.e., **data**) which represent numbers, words, pictures, sounds, and many other abstractions. Because we are talking about digital computers, the information to be processed must be represented as discrete values selected from a (possibly very large but finite) set of individual values. For example, integer numbers (the counting numbers) can be represented in a computer by giving a unique *pattern* to each integer up to the maximum number of patterns available to the particular machine. We will see how these patterns are defined in a later section of this Chapter. This mapping of an internal machine *pattern* to a *meaning* is refered to as a **data type**.

Given a representation of information, we would like to be able to perform operations on this data such as addition or comparison. The fundamental operations provided in a computer are very simple logical and arithmetic operations; however, these simple operations can be combined to perform more complex operations. For example, multiplication can be performed by doing repeated additions. The basic operations provided by a particular computer are called **instructions** and a well defined sequence of these instructions is called a **program**. It is the job of the programmer, then, to represent the information of the problem using the data types provided and to specify the sequence of operations which must be performed to solve the problem. As we will

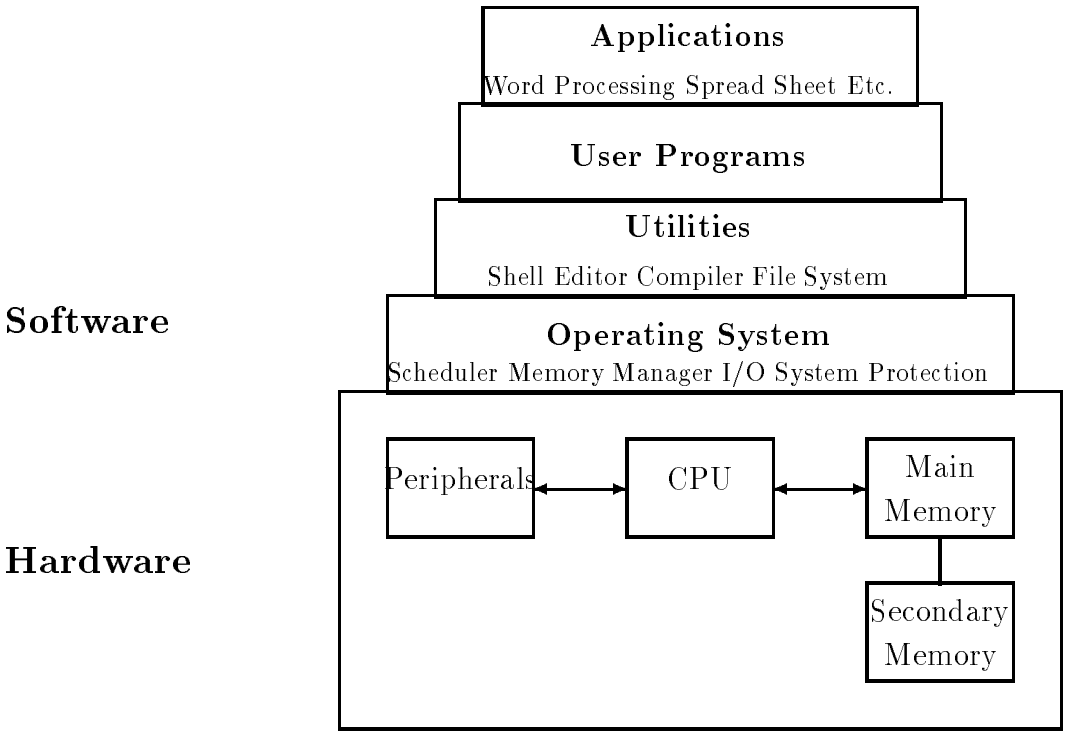

Figure 1.1: Computer System Block Diagram

see in Section 1.2.3, because of the simple nature of the operations available, specifying the proper sequence of instructions to perform a task can be a very complex and tedious task. Fortunately for us, this task has been made simpler these days (using the computers themselves) through the use of high level programming languages. It is one of these languages, the C language that we will discuss in this text.

1.1 Computer System Organization

Before we look at the C language, let us look at the overall organization of computing systems. Figure 1.1 shows a block diagram of a typical computer system. Notice it is divided into two major sections; *hardware* and *software*.

1.1.1 Computer Hardware

The physical machine, consisting of electronic circuits, is called the **hardware**. It consists of several major units: the *Central Processing Unit* (CPU), *Main Memory*, *Secondary Memory* and *Peripherals*.

The CPU is the major component of a computer; the "electronic brain" of the machine. It consists of the electronic circuits needed to perform operations on the data. Main Memory is where programs that are currently being executed as well as their data are stored. The CPU

fetches program instructions in sequence, together with the required data, from Main Memory and then performs the operation specified by the instruction. Information may be both read from and written to any location in Main Memory so the devices used to implement this block are called **random access memory** chips (RAM). The contents of Main Memory (often simply called **memory**) are both temporary (the programs and data reside there only when they are needed) and volatile (the contents are lost when power to the machine is turned off).

The Secondary Memory provides more long term and stable storage for both programs and data. In modern computing systems this Secondary Memory is most often implemented using *rotating magnetic storage devices*, more commmonly called *disks* (though magnetic tape may also be used); therefore, Secondary Memory is often referred to as **the disk**. The physical devices making up Secondary Memory, the *disk drives*, are also known as **mass storage devices** because relatively large amounts of data and many programs may be stored on them.

The disk drives making up Secondary Memory are one form of *Input/Output* (I/O) device since they provide a means for information to be brought into (input) and taken out of (output) the CPU and its memory. Other forms of I/O devices which transfer information between humans and the computer are represented by the *Peripherals* box in Figure 1.1. These Peripherals include of devices such as terminals – a keyboard (and optional mouse) for input and a video screen for output, high-speed printers, and possibly floppy disk drives and tape drives for permanent, removable storage of data and programs. Other I/O devices may include high-speed optical scanners, plotters, multiuser and graphics terminals, networking hardware, etc. In general, these devices provide the *physical interface* between the computer and its environment by allowing humans or even other machines to communicate with the computer.

1.1.2 Computer Software – The Operating System

Hardware is called "hard" because, once it is built, it is relatively difficult to change. However, the hardware of a computer system, by itself, is useless. It must be given directions as to what to do, i.e. a program. These programs are called **software**; "soft" because it is relatively easy to change both the instructions in a particular program as well as which program is being executed by the hardware at any given time. When a computer system is purchased, the hardware comes with a certain amount of software which facilitates the use of the system. Other software to run on the system may be purchased and/or written by the user. Some major vendors of computer systems include: IBM, DEC, HP, AT&T, Sun, Compaq, and Apple.

The remaining blocks in Figure 1.1 are typical software layers provided on most computing systems. This software may be thought of as having a hierarchical, layered structure, where each layer uses the facilities of layers below it. The four major blocks shown in the figure are the *Operating System*, *Utilities*, *User Programs* and *Applications*.

The primary responsibility of the *Operating System* (OS) is to "manage" the "resources" provided by the hardware. Such management includes assigning areas of memory to different programs which are to be run, assigning one particular program to run on the CPU at a time, and controlling the peripheral devices. When a program is called upon to be **executed** (its operations

performed), it must be **loaded**, i.e. moved from disk to an assigned area of memory. The OS may then direct the CPU to begin fetching instructions from this area. Other typical responsibilities of the OS include Secondary Storage management (assignment of space on the disk), a piece of software called the file system, and Security (protecting the programs and data of one user from activities of other users that may be on the same system).

Many mainframe machines normally use proprietary operating systems, such as VM and CMS (IBM) and VAX VMS and TOPS 20 (DEC). More recently, there is a move towards a standardized operating system and most workstations and desktops typically use Unix (AT&T and other versions). A widely used operating system for IBM PC and compatible personal computers is DOS (Microsoft). Apple Macintosh machines are distinguished by an easy to use proprietary operating system with graphical icons.

1.1.3 Utility Programs

The layer above the OS is labeled *Utilities* and consists of several programs which are primarily responsible for the *logical interface* with the user, i.e. the "view" the user has when interacting with the computer. (Sometimes this layer and the OS layer below are considered together as the operating system). Typical utilities include such programs as *shells*, *text editors*, *compilers*, and (sometimes) the *file system*.

A **shell** is a program which serves as the primary interface between the user and the operating system. The shell is a "command interpreter", i.e. is prompts the user to enter commands for tasks which the user wants done, reads and interprets what the user enters, and directs the OS to perform the requested task. Such commands may call for the execution of another utility (such as a text editor or compiler) or a user program or application, the manipulation of the file system, or some system operation such as logging in or out. There are many variations on the types of shells available, from relatively simple command line interpreters (DOS) or more powerful command line interpreters (the Bourne Shell, *sh*, or C Shell, *csh* in the Unix environment), to more complex, but easy to use graphical user interfaces (the Macintosh or Windows). You should become familiar with the particular shell(s) available on the computer you are using, as it will be your primary means of access to the facilities of the machine.

A **text editor** (as opposed to a word processor) is a program for entering programs and data and storing them in the computer. This information is organized as a unit called a **file** similar to a file in an office filing cabinet, only in this case it is stored on the disk. (Word processors are more complex than text editors in that they may automatically format the text, and are more properly considered applications than utilities). There are many text editors available (for example *vi* and *emacs* on Unix systems) and you should familiarize yourself with those available on your system.

As was mentioned earlier, in today's computing environment, most programming is done in high level languages (HLL) such as C. However, as we shall see in Section 1.2.3, the computer hardware cannot understand these languages directly. Instead, the CPU executes programs coded in a lower level language called the **machine language**. A utility called a **compiler** is program which translates the HLL program into a form understandable to the hardware. Again, there are

many variations in compilers provided (for different languages, for example) as well as facilities provided with the compilers (some may have built-in text editors or debugging features). Your system manuals can describe the features available on your system.

Finally, another important utility (or task of the operating system) is to manage the *file system* for users. A file system is a collection of files in which a user keeps programs, data, text material, graphical images, etc. The file system provides a means for the user to organize files, giving them names and gathering them into *directories* (or folders) and to manage their file storage. Typical operations which may be done with files include creating new files, destroying, renaming, and copying files.

1.1.4 User Programs and Applications

Above the utilities in Figure 1.1 is the block labeled *User Programs*. It is at this level where a computer becomes specialized to perform a task to solve a user's problem. Given a task that needs to be performed, a programmer can design and code a program to perform that task using the text editors, compilers, debuggers, etc. The program so written may make use of operating system facilities, for example to do I/O to interact with the program user. It is at this level that the examples, exercises and problems in this text will be written.

However, not everyone who uses a computer is a programmer or desires to be a programmer. As well, if every time a new task was presented to be programmed, one had to start from scratch with a new program, the utility and ease of using the computers would be reduced. These days packages of predefined software, or *Applications*, are available from many vendors in the industry. Highly functional *word processors*, *desktop publishing* packages, *spread sheet* and *data base* programs and, yes, *games* are readily available for computer users as well as programmers. In fact, perhaps most computer users these days access their machines exclusively through these application programs.

A computer system is typically purchased with an operating system, a variety of utilities (such as compilers for high level languages and text editors) and application programs. Without the layers of software in modern computers, computer systems would not be as useful and popular as they are today. While the complexity of these underlying layers has increased greatly in recent years, the net effect has been to make computers easier for people to use.

In the remainder of this Chapter we will take a more detailed look at how data and programs are represented within the machine. We finally discuss the design of programs and their coding in the C language before beginning a detailed description in Chapter 2.

1.2 Representing Data and Program Internally

In a computer, it is the hardware discussed in the previous section that stores data items and programs and that performs operations on these items. This hardware is implemented using electronic circuits called **gates** which, because we are talking about digital computers, represent

information using only two values: *True* and *False*. In most machines, these two values are represented by two different voltages with in the circuit; for example 0 Volts representing a False value, and +5 Volts representing a True value. One such value is called a **binary digit** or **bit** and each such bit can be considered to be a symbol for a piece of information. However, in computer applications we need to represent information that can have more than just two values, i.e. we have more than 2 symbols. So bits are grouped together and the pattern of values on the group is used to represent a symbol. For example, a group of 8 bits, called a **byte** can have 256 different patterns (we will see how below) and therefore represent 256 different symbols. In modern computers, groupings of bytes (usually 2 or 4), called **words** can represent larger "chunks" of information.

Simply representing symbols in a computer, however, is not sufficient. We also need to *process* the information, i.e. perform operations on it. The designers of the hardware make use of an algebra, called **Boolean Algebra**, which uses two values, 0 and 1, and logical operations (AND, OR and NOT) to design the circuits that perform more complex operations on bytes and words of data. These complex operations are the **instruction set** of the computer and are the basic tools the programmer has to write software for the computer. All executable programs must be sequences of instructions from this set which includes basic arithmetic, logical, store and retrieve, and program control instructions. The instructions themselves can also be represented in the machine as patterns of bits.

This section first discusses how different types of data are represented using patterns of bits, then describes how data, as well as instructions, are stored in memory, and finally gives a short example of how instructions are represented.

1.2.1 Representing Data

Standard methods for representing commonly used numeric and non-numeric data have been developed and are widely used. While a knowledge of internal binary representation is not required for programming in C, an understanding of internal data representation is certainly helpful.

Binary Representation of Integers

As mentioned above, all data, including programs, in a computer system is represented in terms of groups of binary digits. A single bit can represent one of two values, 0 or 1. A group of two bits can be used to represent one of four values:

```
00 --- 0
01 --- 1
10 --- 2
11 --- 3
```

If we have only four symbols to represent, we can make a one-to-one correspondence between the patterns and the symbols, i.e., one and only one symbol is associated with each binary pattern.

For example, the numbers 0, 1, 2, and 3 are mapped to the patterns above.

Such a correspondence is called a **code** and the process of representing a symbol by the corresponding binary pattern is called **coding** or **encoding**. Three binary digits can be used to represent eight possible distinct values using the patterns:

```
000                 100
001                 101
010                 110
011                 111
```

A group of k binary digits (bits) can be used to represent 2^k symbols. Thus, 8 bits are used to represent $2^8 = 256$ values, 10 bits to represent $2^{10} = 1024$ values, and so on. It should be clear by now that powers of two play an important role because of the binary representation of all data. The number 1024 is close to one thousand, and it is called $1K$, where K stands for Kilo; nK equals $n * 1024$, and if $n = 2^m$, then nK is $2^{(10+m)}$.

We will first present a standard code for natural numbers, i.e., unsigned integers 0, 1, 2, 3, 4, etc. There are several ways to represent these numbers as groups of bits, the most natural way is analogous to the method we use to represent decimal numbers. Recall, a decimal (or base 10) representation uses exactly ten digit symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Any decimal number is represented using a weighted positional notation.

For example, a single digit number, say 9, represents just nine, because the weight of the rightmost position is 1. A two digit number, say 39, represents thirty plus nine. The rightmost digit has a weight 1, and the next digit to the left has a weight of 10. So, we multiply 3 by 10, and add 9 multiplied by 1. Thus, for decimal notation the weights for the digits starting from the rightmost digit and moving to the left are 1, 10, 100, and so on, as shown below.

5	4	3	2	1	0	digit position
100000	10000	1000	100	10	1	position weight

Thus,

$$3456 = (6 * 1) + (5 * 10) + (4 * 100) + (3 * 1000)$$

The positional weights are the powers of the base value 10, with the rightmost position having the weight of 10^0, the next positions to the left having in succession the weight of 10^1, 10^2, 10^3, and so on. Such an expression is commonly written as a sum of the contribution of each digit, starting with the lowest order digit and working toward the largest weight; that is, as sums of contributions of digits starting from the rightmost position and working toward the left.

Thus, if i is an integer written in decimal form with digits d_k:

$$i = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$$

then i represents the sum:

$$i = \sum_{k=0}^{n-1} d_k * 10^k$$

where n is the total number of digits, and d_k is the k^{th} digit from the rightmost position in the decimal number.

Binary representation of numbers is written in exactly the same manner. The base is 2, and a number is written using just two digits symbols, 0 and 1. The positional weights, starting from the right are now powers of the base 2. The weight for the rightmost digit is $2^0 = 1$, the next digit has the weight of $2^1 = 2$, the next digit has the weight of $2^2 = 4$, and so on. Thus, the weights for the first ten positions from the right are as follows:

10	9	8	7	6	5	4	3	2	1	0	position
1024	512	256	128	64	32	16	8	4	2	1	pos. weights

A natural binary number is written using these weights. For example, the binary number

1 0 0 1 0

represents the number whose decimal equivalent is

$$2^1 + 2^4 = 2 + 16 = 18$$

and the binary number

1 0 1 0 1 0 0 0

represents the number whose decimal equivalent is

$$2^3 + 2^5 + 2^7 = 8 + 32 + 128 = 168$$

When a binary number is stored in a computer word with a fixed number of bits, unused bits to the left (leading bits) are set to 0. For example, with a 16 bit word, the binary equivalent of 168 is

0000 0000 1010 1000

We have shown the bits in groups of four to make it easier to read.

In general, if i is an integer written in binary form with digits b_k:

$$i = b_{n-1}b_{n-2}\ldots b_2 b_1 b_0$$

then its decimal equivalent is:

$$i = \sum_{k=0}^{n-1} b_k * 2^k$$

As we said, a word size of k bits can represent 2^k distinct patterns. We use these patterns to represent the unsigned integers from 0 to $2^k - 1$. For example, 4 bits have 16 distinct patterns representing the equivalent decimal unsigned integers 0 to 15, 8 bits for decimal numbers 0 through 255, and so forth.

Given this representation, we can perform operations on these unsigned integers. Addition of two binary numbers is straightforward. The following examples illustrate additions of two single bit binary numbers.

```
 0        0        1        1
+0       +1       +0       +1
---      ---      ---      ---
 0        1        1       10
```

The last addition, $1 + 1$, results in a sum digit of 0 and a carry forward of 1. Similarly, we can add two arbitrary binary numbers, b1 and b2:

```
          011100      (carry forward    )

 b1       101110      (base 10 value: 46)
+b2      +001011      (base 10 value: 11)
---      --------
sum       111001      (base 10 value: 57)
```

Decimal to Binary Conversion

We have seen how, given a binary representation of a number, we can determine the decimal equivalent. We would also like to go the other way; given a decimal number, find the corresponding binary pit pattern representing this number. In general, there are two approaches; one generates the bits from the most significant (the leftmost bit) to the least significant; the other begins with the rightmost bit and proceeds to the leftmost.

In the first case, to convert a decimal number, n, to a binary number, determine the highest power, k, of 2 that can be subtracted from n:

$$r = n - 2^k$$

and place a 1 in the k^{th} binary digit position. The process is repeated for the remainder r, and so forth until the remainder is zero. All other binary digit positions have a zero. For example, consider a decimal number 103. The largest power of 2 less than 103 is 64 (2^6):

$$\begin{array}{rcrcrcrcr} 103 & - & 2^6 & = & 103 & - & 64 & = & 39 \\ 39 & - & 2^5 & = & 39 & - & 32 & = & 7 \\ 7 & - & 2^2 & = & 7 & - & 4 & = & 3 \\ 3 & - & 2^1 & = & 3 & - & 2 & = & 1 \\ 1 & - & 2^0 & = & 1 & - & 1 & = & 0 \end{array}$$

So we get:

```
weights        128  64   32   16   8    4    2    1
                    1    1                 1    1    1
```

which, for an 8 bit representation give:

0110 0111

In the alternate method, we divide n by 2, using integer division (discarding any fractional part), and the remainder is the next binary digit moving from least significant to most. In the example below, the % operation is called **mod** and is the remainder from integer division.

$$\begin{array}{rcrcr \quad rcrcr} 103 & \% & 2 & = & 1 & 103 & / & 2 & = & 51 \\ 51 & \% & 2 & = & 1 & 51 & / & 2 & = & 25 \\ 25 & \% & 2 & = & 1 & 25 & / & 2 & = & 12 \\ 12 & \% & 2 & = & 0 & 12 & / & 2 & = & 6 \\ 6 & \% & 2 & = & 0 & 6 & / & 2 & = & 3 \\ 3 & \% & 2 & = & 1 & 3 & / & 2 & = & 1 \\ 1 & \% & 2 & = & 1 & 1 & / & 2 & = & 0 \end{array}$$

Reading the bits top to bottom filling right to left, the number is

0110 0111

Representing Signed Integers

The binary representation discussed above is a standard code for storing unsigned integer numbers. However, most computer applications use signed integers as well; i.e. integers that may be either positive or negative. There are several methods used for representing signed numbers.

The first, and most obvious, is to represent signed numbers as we do in decimal, with an indicator for the sign followed by the magnitude of the number as an unsigned quantity. For example, we write:

$+100$

-100

In binary we can use one bit within a representation (usually the most significant or leading bit) to indicate either positive (0) or negative (1), and store the unsigned binary representation of the magnitude in the remaining bits. So for an 16 bit word, we can represent the above numbers as:

$$+100: \quad 0000\ 0000\ 0110\ 0100$$

$$-100: \quad 1000\ 0000\ 0110\ 0100$$

However; for reasons of ease of design of circuits to do arithmetic on signed binary numbers (e.g. addition and subtraction), a more common representation scheme is used called **two's complement**. In this scheme, positive numbers are represented in binary, the same as for unsigned numbers. On the other hand, a negative number is represented by taking the binary representation of the magnitude, complementing all bits (changing 0's to 1's and 1's to 0's), and adding 1 to the result.

Let us examine the 2's complement representation of $+100$ and -100 using 16 bits. For $+100$, the result is the same as for unsigned numbers:

$$+100: \quad 0000\ 0000\ 0110\ 0100$$

For -100, we begin with the unsigned representation of 100:

$$0000\ 0000\ 0110\ 0100$$

complement each bit:

$$1111\ 1111\ 1001\ 1011$$

and add 1 to the above to get 2's complement:

$$-100: \quad 1111\ 1111\ 1001\ 1100$$

This operation is reversible, that is, the magnitude (or absolute value) of a two's complement representation of a negative number can be obtained with the same procedure; complement all bits:

$$0000\ 0000\ 0110\ 0011$$

and add 1:

$$0000\ 0000\ 0110\ 0100$$

In a two's complement representation, we can still use the most significant bit to determine the sign of the number; 0 for positive, and 1 for negative. Let us determine the decimal value of a negative 2's complement number:

$$1111\ 1111\ 1101\ 0110$$

This is a negative integer since the leading bit is 1, so to find its magnitude we complement all bits:.

$$0000\ 0000\ 0010\ 1001$$

and add 1:

0000 0000 0010 1010

The decimal magnitude is 42, and the sign is negative, so, the original integer represents decimal -42.

In determining the range of integers that can be represented by k bits, we must allow for the sign bit. Only $k-1$ bits are available for positive integers, and the range for them is 0 through $2^{(k-1)}-1$. The range of negative integers representable by k bits is -1 through $-2^{(k-1)}$. Thus, the range of integers representable by k bits is $-2^{(k-1)}$ through $2^{(k-1)}-1$. For example, for 8 bits, the range of signed integers is $-2^{(8-1)}$ through $2^{(8-1)}-1$, or -128 to $+127$.

It can be seen from the above analysis that, due to a finite number of bits used to represent numbers, there are limits to the largest and/or smallest numbers that can be represented in the computer. We will discuss this further in Chapter 5.

Octal and Hexadecimal Representations

One important thing to keep in mind at this point is that we have been discussing different *representations* for numbers. Whether a number is expressed in binary, e.g. 010011, or decimal, 19, it is still the same number, namely nineteen. It is simply more convenient for people to think in decimal and for the computer to use binary. However, converting the computer binary representation to the human decimal notation is somewhat tedious, but at the same time writing long strings of bits is also inconvenient and error prone. So two other representation schemes are commonly used in working with binary representations. These schemes are call **octal** and **hexadecimal** (sometimes called **hex**) representations and are simply positional number systems using base 8 and 16, respectively.

In general, an unsigned integer, i, consisting of n digits d_i written as:

$$i = d_{n-1}d_{n-2}\ldots d_3d_2d_1d_0$$

in any base is interpreted as the sum:

$$i = \sum_{k=0}^{n-1} d_k * \mathrm{b}ase^k$$

If the base is 2 (binary), the symbols which may be used for the digits d_i are [0, 1]. If the base is 10 (decimal) the digit symbols are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Likewise, for base 8 (octal) the digit symbols are [0, 1, 2, 3, 4, 5, 6, 7]; and for hexadecimal (base 16) they are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f]. The letter symbols, [a, b, c, d, e, f] (upper case [A, B, C, D, E, F] may also be used) give us the required 16 symbols and correspond to decimal values [10, 11, 12, 13, 14, 15] respectively. Using the above sum, it should be clear that the following are representations for the same number:

Base 10: 423
Base 2: 0000 0001 1010 0111
Base 8: 000647
Base 16: 01A7

For hexadecimal numbers, the positional weights are, starting from the right, 1, 16, 256, etc. Here are a few examples of converting hex to decimal:

Hexadecimal	Decimal		
30	0 * 1 + 3 * 16	=	48
1E	14 * 1 + 1 * 16	=	30
1C2	2 * 1 + 12 * 16 + 1 * 256	=	450

Similarly, octal numbers are base 8 numbers with weights 1, 8, 64, etc. The following are some examples of converting octal to decimal:

Octal	Decimal		
11	1 * 1 + 1 * 8	=	9
20	0 * 1 + 2 * 8	=	16
257	7 * 1 + 5 * 8 + 2 * 64	=	175

The reason octal and hex are so common in programming is the ease of converting between these representations and binary, and vice versa. For hexadecimal numbers, exactly four bits are needed to represent the symbols 0 through F. Thus, segmenting any binary number into 4 bit groups starting from the right, and converting each group to its hexadecimal equivalent gives the hexadecimal representation.

Binary:	1010		1000
Hex:	A		8
	10 * 16	+	8 * 1
Decimal:	168		

As a side effect, conversion from binary to decimal is much easier by first converting to hex and then to decimal, as shown above.

Similarly, segmenting a binary number into three bit groups starting from the right gives us the octal representation. Thus, the same number can be expressed in octal as:

Binary:	10		101		000
Octal:	2		5		0
	2 * 64	+	5 * 8	+	0 * 1
Decimal:	168				

Conversion of base 8 or base 16 numbers to binary is very simple; for each digit, its binary representation is written down. Conversion between octal and hex is easiest done by converting to binary first:

Decimal	122	199	21	63
Binary	01111010	11000111	010101	111111
Hexadec.	0111 1010 0X7A	1100 0111 0XC7	0001 0101 0X15	0011 1111 0X3F
Octal	01 111 010 0172	11 000 111 0307	00 010 101 025	00 111 111 077

Table 1.1: Number Representations

Hex:	2	f	3	
Binary:	0010	1111	0011	
Binary:	001	011	110	011
Octal:	1	3	6	3

Some additional examples of equivalent hexadecimal, octal, binary, and decimal numbers are shown in Table 1.1 In a programming language we need a way to distinguish numbers written in different bases (base 8, 16, 10, or 2). In C source programs, a simple convention is used to write constants in different bases. Decimal numbers are written without leading zeros. Octal numbers are written with a leading zero, e.g. 0250 is octal 250. Hexadecimal numbers are written with a leading zero followed by an x or X, followed by the hexadecimal digits. Thus, 0xA8 will mean hexadecimal A8. (Binary numbers are not used in source programs).

Representing Other Types of Data

So far we have discussed representations of integers, signed and unsigned; however, many applications make use of other types of data in their processing. In addition, some applications using integers require numbers larger than can be stored in the available number of bits. To address these problems, another representation scheme, called **floating point** is used. This scheme allows representation of numbers with fractional parts (real numbers) as well as numbers that may be very large or very small.

Representation of floating point numbers is analogous to decimal *scientific notation.* For example:

$$1.234 * 10 + 2$$

$$.1234 * 10 + 3$$

By adjusting the decimal place, as in the last case above, a number of this form consists of just three parts: a fractional part called the **mantissa**, a base, and an exponent. Over the years, several schemes have been devised for representing each of these parts and storing them as bits in a computer. However, in recent years a standard has been defined by the Institute for Electrical and Electronics Engineers (IEEE Standard 754) which is gaining in acceptance and use in many computers. A detailed description of these schemes, and their relative tradeoffs, is beyond the scope of this text; however, as programmers, it is sufficient that we realize that we can express floating point numbers either in decimal with a fractional part:

$$325.54927$$

or using exponential form:

$$3.2554927E + 2$$

$$325549.27E - 3$$

where E or e refers to exponent of the base (10 in this case). As with integers, due to the fixed number of bits used in the representation, there are limits to the range (largest and smallest numbers) and the precision (number of digits of accuracy) for the numbers represented.

Another widely used data type is character data which is non-numeric and includes all the symbols normally used to represent textual material such as letters, digits and punctuation. Chapter 4 discusses the representation of character data in detail, however, the principle is the same; some pattern of bits is selected to represent each symbol to be stored.

These are the principle data types provided by programming languages, but as we will see in future Chapters, languages also provide a means for the programmer to define their own data types and storage schemes.

1.2.2 Main Memory

Now that we have seen that information (data) can be represented in a computer using binary patterns, we can look at how this information is stored within the machine. An electronic circuit that can be switched ON or OFF can represent one binary digit or one bit of information. A class of such devices (called **flip-flops**) which can retain the value of a bit, even after the input to them changes (though only as long as power is applied to them), can be used to store a bit. The Main Memory block of Figure 1.1 is constructed of many of these devices, organized so that data (and instructions) may be stored there and subsequently accessed. Memory in present day computers is usually organized as a sequence of bytes (a **byte** is a group of eight bits). Each byte in memory is given a unique unsigned integer *address*, which may be considered its "name". (See Figure 1.2). A row of houses on a street with street addresses or a row of numbered mailboxes are reasonable analogies to memory addresses. The CPU (or any other device wishing to access memory) may place an address on a set of wires connected to the memory (called the **address bus**) in order to either read (load) or write (store) information in memory. Once information

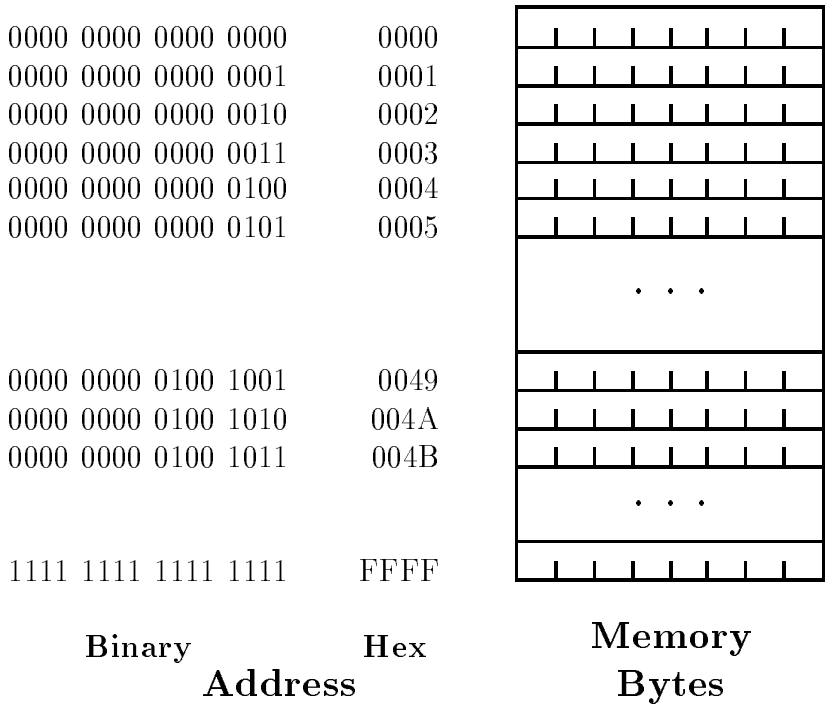

Figure 1.2: Memory and Addresses

has been written to a particular location (address) in memory, it will remain unchanged unless a subsequent write is performed to the same address. Multiple bytes may be accessed (either simultaneously or sequentially) for data items large than a single byte. Like other information in the computer, an address is represented internally in binary. In the figure, we have shown the addresses both in binary and in hexadecimal form.

Computers are often classified by how many bits may be accessed simultaneously, e.g. 16 bits or 32 bits. The maximum number of bytes directly addressable in a computer depends on the number of bits in the memory address. A 16 bit machine usually allows 16 bits for address and a 32 bit machine usually allows anywhere form 17 to 32 bits for address. Since n bits can represent 2^n values, 16 bit addresses can address 64 KBytes (i.e. 65,536 bytes from byte addresses 0 to 65535) and 32 bit addresses can address 4 GigaBytes (over 4,000,000,000 bytes) directly.

1.2.3 Representing Programs

As has been mentioned, in addition to data being stored in memory, the program to be executed is also stored there in the form of a sequence of instructions. It is the CPU shown in Figure 1.1 that is responsible for fetching instructions, one at a time, from memory and performing the specified operation on data. A more detailed picture of the CPU with its memory is shown in Figure 1.3. Within the CPU are several key components; the ALU, a set of Registers, and a Control Unit.

The ALU (Arithmetic Logic Unit) is a digital circuit which is designed to perform arithmetic

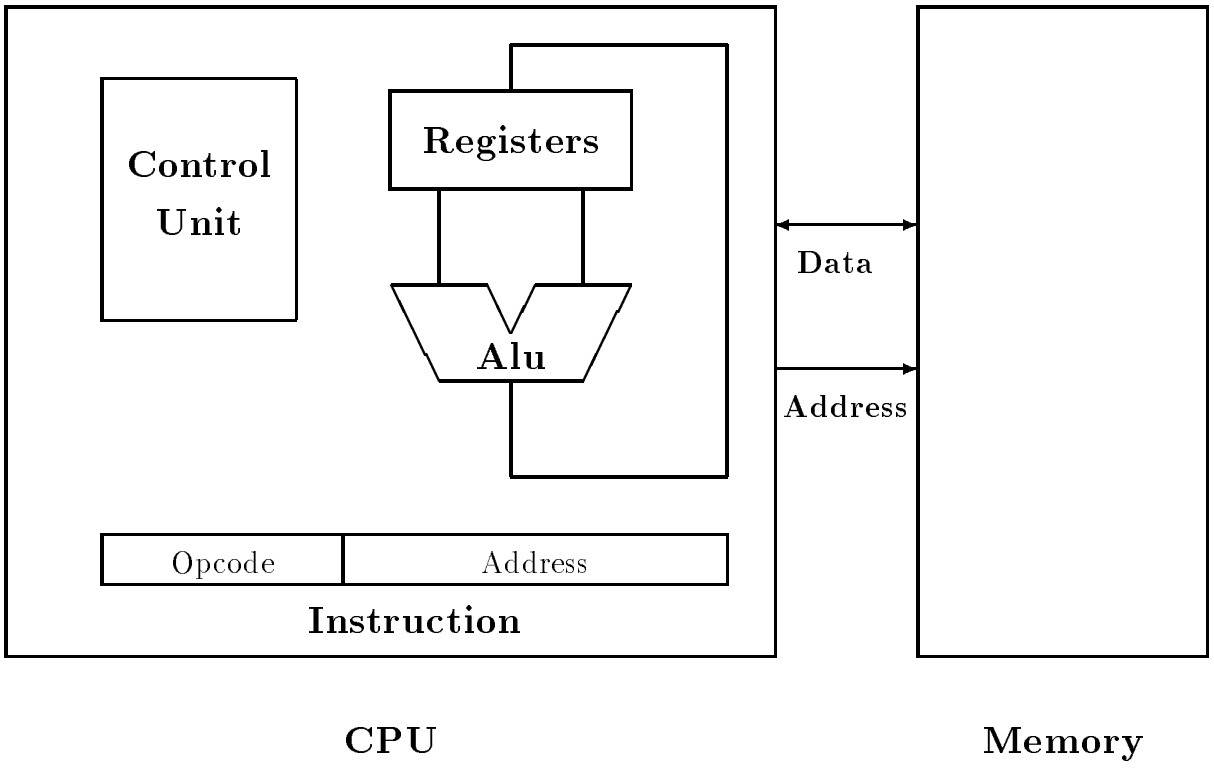

Figure 1.3: CPU and Memory Configuration

(add, subtract) operations as well as logic (AND, OR) operations on data. The registers in the CPU are a small scratchpad memory to temporarily store data while it is in use. The Control Unit is another circuit which determines what operation is being requested by an instruction and controls the other circuitry to carry out that operation; i.e. the Control Unit directs all operations within the machine.

Also shown in the figure are the connections between the CPU and Memory. They consist of an address bus, as mentioned in the previous Section, and a data bus, over which all information (data and program) passes between the CPU and Memory.

This Section describes how programs are stored in the machine as a sequence of instructions coded in binary. Such an encoding is called the **machine language** of the computer and is described below.

Machine Language

The basic operations that the CPU is capable of performing are usually quite simple and the set of these operations provided on a particular computer is called the **instruction set**. Within this set are instructions which can move data from one place to another, for example from memory to a CPU register; an operation called **load**. Similarly there are **store** instructions for moving data from the CPU to a location in memory. In addition there are instructions directing arithmetic

operations, such as add, on data values. There are also instructions which control the flow of the program; i.e. that determine from where in memory the next instruction should be fetched. Normally instructions are fetched sequentially – the next instruction is fetch from the next memory address; however, these control instructions may test a condition and direct that the next instruction be fetched from somewhere else in memory instead. Finally, there may also be instructions in the set for "housekeeping" operations within the machine, such as controlling external I/O devices.

To encode these instructions in binary form for storage in memory, some convention must be adopted to describe the meaning of the bits in the instruction. Most of the instructions described above require at least 2 pieces of information – a specification of what particular instruction this is, called the **opcode** or operation code, and the address of the data item on which to operate. These parts can be seen in Figure 1.3 in the block labeled *instruction*.

Instructions coded in binary form are called **machine language instructions** and the collection of these instructions that make up a program is called a **machine language program**. Such a program is very difficult for a person to understand or to write. Just imagine thinking in terms of binary codes for very low level instructions and in terms of binary memory addresses for data items. It is not practical to do so except for very trivial programs. Humans require a higher level of programming languages that are more adapted to our way of thinking and communicating. Therefore, at a level a little higher than machine language, is a programming language called **assembly language** which is very close to machine language. Each assembly instruction translates to one machine language instruction. The main advantage is that the instructions and memory cells are not in binary form; they have names. Assembly instructions include operational codes, (i.e., mnemonic or memory aiding names for instructions), and they may also include addresses of data. An example of a very simple program fragment for the machine described above is shown in Figure 1.4. The figure shows the machine language code and its corresponding assembly language code. Definitions of memory cells are shown below the program fragment.

The machine language code is shown in binary. It consists of 8 bits of opcode and 16 bits of address for each instruction. From the assembly language code it is a little easier to see what this program does. The first instruction loads the data stored in memory at a location known as "Y" into the CPU register (for CPU's with only one register, this is often called the **accumulator**). The second instruction adds the data stored in memory at location "X" to the data in the accumulator, and stores the sum back in the accumulator. Finally, the value in the accumulator is stored back to memory at location "Y". With the data values shown in memory in the figure, at the end of this program fragment, the location known as "Y" will contain the value 48.

A utility program is provided to translate the assembly language code (arguably) readable by people into the machine language code readable by the CPU. This program is called the **assembler**. The program in the assembly language or any other higher language is called the **source program**, whereas the program assembled into machine language is called the **object program**. The terms source code and object code are also used to refer to source and object programs.

Assembly language is a decided improvement over programming in machine language, however, we are still stuck with having to manipulate data in very simple steps such as load, store, add, etc., which can be a tedious, error prone process. Fortunately for us, programming languages at

```
Program Fragment:       Y = Y + X

Machine Language Code                          Assembly Language
(Binary Code)                                        Code

Opcode          Address
1100 0000       0010 0000 0000 0000                LOAD  Y
1011 0000       0001 0000 0000 0000                ADD   X
1001 0000       0010 0000 0000 0000                STORE Y


Memory Cell Definitions:

     Addr.      Name       Cell Contents

     1000       X          32
     2000       Y          16
```

Figure 1.4: Machine and Assembly Language Program Fragment

higher levels still, languages closer to the way we think about programming, have been developed along with translators (called **compilers**) for converting to object programs. One such language is C, which is the subject of this text and is introduced in the next Section.

1.3 Designing Programs and the C Language

We defined a program as an organized set of instructions stating the steps to be performed by a computer to accomplish a task. **Computer programming** is the process of planning, implementing, testing, and revising (if necessary) the sequences of instructions in order to develop successful programs. In writing computer programs we must specify with precise, unambiguous instructions exactly what we want done and the order in which it should be done. Before we can write the actual program, we must either know or develop a step-by-step procedure, or *algorithm*, that will accomplish the task. We can then implement the algorithm by coding it into a source language program.

1.3.1 Designing The Algorithm

An algorithm is a general solution of a problem which can be written as a verbal description of a precise, logical sequence of actions. Cooking recipes, assembly instructions for appliances and

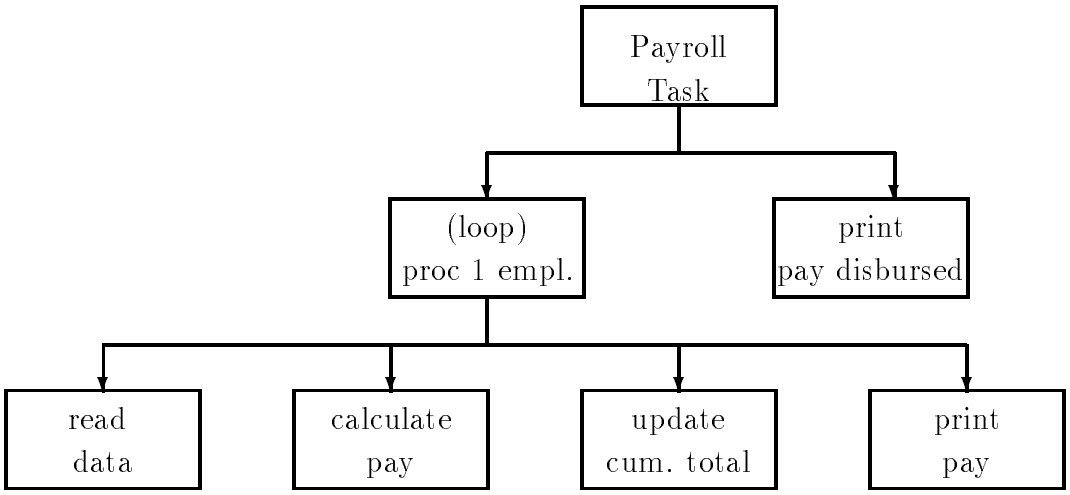

Figure 1.5: Structural Diagram for Payroll Task

toys, or precise directions to reach a friend's house, are all examples of algorithms. A computer program is an algorithm expressed in a specific programming language. An algorithm is the key to developing a successful program.

Suppose a business office requires a program for computing its payroll. There are several people employed. They work regular hours, and sometimes overtime. The task is to compute pay for each person as well as compute the total pay disbursed.

Given the problem, we may wish to express our recipe or *algorithm* for solving the payroll problem in terms of repeated computations of total pay for several people. The logical modules involved are easy to see.

```
Algorithm: PAYROLL

     Repeat the following while there is more data:
          get data for an individual,
          calculate the pay for the individual from the current data,
               and, update the cumulative pay disbursed so far,
          print the pay for the individual.
     After the data is exhausted, print the total pay disbursed.
```

Figure 1.5 shows a **structural diagram** for our task. This is a layered diagram showing the development of the steps to be performed to solve the task. Each box corresponds to some subtask which must be performed. On each layer, it is read from left to right to determine the performance order. Proceeding down one layer corresponds to breaking a task up into smaller component steps – a refinement of the algorithm. In our example, the payroll task is at the top and that box represents the entire solution to the problem. On the second layer, we have divided the problem into two subtasks; processing a single employee's pay in a loop (to be described below), and

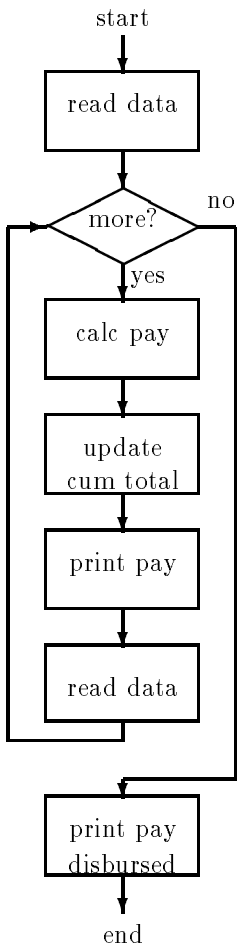

Figure 1.6: Flow Chart for Payroll Task

printing the total pay disbursed for all employees. The subtask of processing an individual pay record is then further refined in the next layer. It consists of, first reading data for the employee, then calculating the pay, updating a cumulative total of pay disbursed, and finally printing the pay for the employee being processed.

The structural diagram is useful in developing the steps involved in designing the algorithm. Boxes are refined until the steps within the box are "doable". Our diagram corresponds well with the algorithm developed above. However, this type of diagram is not very good at expressing the sequencing of steps in the algorithm. For example, the concept of looping over many employees is lost in the bottom layer of the diagram. Another diagram, called a **flow chart** is useful for showing the *control flow* of the algorithm, and can be seen in Figure 1.6. Here the actual flow of control for repetitions is shown explicitly. We first read data since the control flow requires us to test if there is more data. If the answer is "yes" we proceed to the calculation of pay for an individual, updating of total disbursed pay so far, and printing of the individual pay. We then read the next set of data and loop back to the test. If there is more data, repeat the process, otherwise control passes to the printing of total disbursed pay and the program ends.

From this diagram we can write our refined algorithm as shown below. However, one module may require further attention; the one that calculates pay. Each calculation of pay may involve arithmetic expressions such as multiplying hours worked by the rate of pay. It may also involve branching to alternate computations if the hours worked indicate overtime work. Incorporating these specifics, our algorithm may be written as follows:

```
Algorithm: PAYROLL

get (first) data, e.g., id, hours worked, rate of pay
while more data (repeat the following)
     if hours worked exceeds 40
          (then) calculate pay using overtime pay calculation
     otherwise calculate pay using regular pay calculation
     calculate cumulative pay disbursed so far
     print the pay statement for this set of data
     get (next) data

print cumulative pay disbursed
```

The algorithm is the most important part of solving difficult problems. Structural diagrams and flow charts are tools that make the job of writing the algorithm easier, especially in complex programs. The final refined algorithm should use the same type of constructs as most programming languages. Once an algorithm is developed, the job of writing a program in a computer language is relatively easy; a simple translation of the algorithm steps into the proper statements for the language. In this text, we will use algorithms to specify how tasks will be performed. Programs that follow the algorithmic logic will then be easy to implement. Readers may wish to draw structural diagrams and flow charts as visual aids in understanding complex algorithms.

There is a common set of programming constructs provided by most languages useful for algorithm construction, including:

- *Branching*: test a condition, and specify steps to perform for the case when the condition is satisfied (True), and (optionally) when the condition is not satisfied (False). This construct was used in our algorithm as:

```
if overtime hours exceed 40
     then calculate pay using overtime pay calculation
otherwise calculate pay using regular pay calculation
```

- *Looping*: repeat a set of steps as long as some condition is True, as seen in:

```
while new data repeat the following
     ...
```

- *Read* or *print* data from/to peripheral devices. Reading of data by programs is called data input and writing by programs is called data output. The following steps were used in our algorithm:

```
read data
write/print data, individual pay, disbursed pay
```

Languages that include the above types of constructions are called **algorithmic languages** and include such languages as C, Pascal, and FORTRAN.

A program written in an algorithmic language must, of course, be translated into machine language. A Utility program, called a **compiler**, translates source programs in algorithmic languages to object programs in machine language. One instruction in an algorithmic language, called a **statement**, usually translates to several machine level instructions. The work of the compiler, the translation process, is called **compilation**.

To summarize, program writing requires first formulating the underlying algorithm that will solve a particular problem. The algorithm is then coded into an algorithmic language by the programmer, compiled by the compiler, and loaded into memory by the operating system. Finally, the program is executed by the hardware.

1.3.2 The C Language

In this text, our language of choice for implementing algorithms is C. C was originally developed on a small machine (PDP-11) by Dennis Ritchie for implementing the UNIX operating system at Bell Laboratories in Murray Hill, New Jersey (1971-73). C is now used for a wide range of applications including UNIX implementations, systems programming, scientific and engineering computation, spreadsheets, and word processing. In fact, the popularity of C has encouraged the development of a C standard by the American National Institute of Standards (ANSI). This text adheres to *ANSI C*. Major differences between ANSI C and "old C" are pointed out in Appendix B. References at the end of this chapter include books by Kernighan and Ritchie [1, 2], which define both traditional C and ANSI C as well as a reference to the proposed ANSI C standard by Harbison and Steele[3].

In keeping with the original intent, C is a small language; however, it features modern control flow and data structures and a rich set of operators. C provides a wealth of constructs, or statements, which correspond to good algorithmic structures. C uses a standard library of functions to perform many routine tasks such as input and output and string operations. Since C is oriented towards the use of a library of functions, programs in C tend to be modular with numerous small functional modules. It is also possible for users to develop their own libraries of functions to improve program development.

C is fairly standard; programs written in C are easily moved from one machine to another. Such portability of programs is a major advantage in that applications developed on one computer can be

used elsewhere. This allows one to write clear and algorithmically well structured programs. Such a *structured programming* approach is very important in developing complex, error-free applications.

C provides low level logic operations, normally available only in machine language or assembly language. Low level operations are required for *systems programming*, such as writing operating systems and other programs at the system level. Today, many operating systems are written in C. C is also suitable for writing scientific and engineering programs, for example it provides double precision computations of real numbers, as well as long integer computation which can be useful in many applications where a large range of integers is required.

As a first programming language C has some weaknesses; however, they can be overcome by discipline in writing programs. In the text, we will indicate items that beginning programmers need to watch out for.

1.4 Summary

In this Chapter we have given a brief overview of modern computing systems, including both the hardware and software. We had described how information is represented in these machines, both data and programs. We have discussed the development of algorithms as the first, and probably most important step in writing a program. As we shall see, programming is a design process; an algorithm is written, coded, and tested followed by *iteration*. Programs are not written in one step – initial versions are developed and then refined and improved.

One brief note about the organization of chapters in the text. In this chapter (following the References) are two sections labeled *Exercises* and *Problems*. These are very important sections in learning to program, because the only way to learn and improve programming skills is to program. The exercises are designed to be done with pencil-and-paper. They test the key concepts and language constructs presented in the chapter. The problems are generally meant to be computer exercises. They present problems for which programs should be written. By writing these programs you will increase your experience in the methods and thought processes that go into developing ever more complex applications.

With the background of this Chapter, we are ready to begin looking at the specifics of the C language, so

E hoʻomaka kākou.
(Let's start).

1.5 References

[1] Kernighan, Brian W., and Ritchie, Dennis M. The C Programming Language, First Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1978.

[2] Kernighan, Brian W., and Ritchie, Dennis M. The C Programming Language, Second Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[3] Harbison, Samuel P., and Steele, Guy L. Jr., C: A Reference Manual, Second Edition, Englewood Cliffs, N.J.: Prentice-Hall, 1987.

1.6 Exercises

1. Convert the following binary numbers into decimal values:

```
0000 0100 0110 1001
0011 0001 0111 1111
0101 0101 0101 0101
```

2. Convert the following octal numbers into decimal:

```
000345
000111
000777
```

3. Convert the following hexadecimal numbers into decimal:

```
1A
FF
21
```

4. Convert the following decimal integer values into binary, octal, and hexadecimal:

```
101
324
129
```

5. Add the following binary numbers:

```
0000 0100 0110 1001
0011 0001 0111 1111
0101 0101 0101 0101
```

6. Add the following octal numbers:

```
000345
000111
000777
```

7. Add the following hexadecimal numbers:

```
1A
FF
21
```

8. How many distinct binary strings can be formed with n bits?

9. Find the negative of the following binary numbers in a two's complement representation:

```
0000 0100 0110 1001
0011 0001 0111 1111
0101 0101 0101 0101
```

10. Represent the following in two's complement form using 16 bits:

```
-29
 165
-100
```

11. What is the largest positive integer that can be stored in n bits, with one leading bit reserved for the sign bit? Explain. Negative integer? Assume two's complement representations.

1.7 Problems

1. Develop an algorithm for the calculation of the value of each stock and the total value of a portfolio of stocks. Draw a structural diagram and write the algorithm using constructions used in the text.

2. Develop an algorithm for calculating and printing the squares of a set of numbers. Draw a structural diagram, a flow chart, and write the algorithm.

3. Develop an algorithm for calculation of the grade point ratio for each student, i.e., (total grade points) / (total credit hours). Each student earns grades (0-4) in a set of courses, each course with different credit hours (1-3). Grade points in one course are given by the product of the grade and the credit hours for the course. Draw a structural diagram and a flow chart.

4. Assume that an "add" operator is available, but not a "multiply" operator in a programming language. Develop an algorithm that will multiply two positive integers using only the "add" operator.

5. Assume that you are only able to read the numeric value of each successive digits of a decimal integer one digit at a time. The objective is to find the overall numeric value of the number. As each new digit is read, the overall numeric equivalent must be updated to allow for the new digit. For example, if the digits read are 3,2, and 5, the result printed should be 325. Extend the algorithm for a number in any specified base.

6. Log in to the computer system available to you. Practice using the text editor available by entering the following simple program and storing it in a file:

```
main()
{
printf("hello world\n");
}
```

7. Compile the program you entered in Problem 6. Note which file have been created during compilation. Execute the compiled program.

8. Explore the computer you will be using. See what applications may be available to you such as electronic mail, and news.