An Introduction
to the
Structure
of
Computers and Programs

Jerud J. Mead

Computer Science Department
Bucknell University
Lewisburg, PA 17387

September, 1997

Contents

Part I

The Organization of a Computer

Chapter 1

Computer Architecture

Just as with buildings, each computer has a "visible" structure referred to as its ARCHITECTURE. The architecture of a building can be examined at various levels of detail: the number of stories and the size of rooms, the details of door and window placement, the layout of electrical wiring or heating conduit. We can look at a computer's architecture at similar levels of detail.

At the highest level a computer can be seen as a collection of components connected by a "device" called a BUS (See Figure 1.1). The bus is a medium for communication amongst the components, with each component having an address on the bus (the BUS ADDRESS). If one component must transfer a packet of data to another device, it will create a message (containing the data) and put it on the bus along with the bus address of the target device; each device "listens" to the bus and removes those messages with the appropriate bus address.

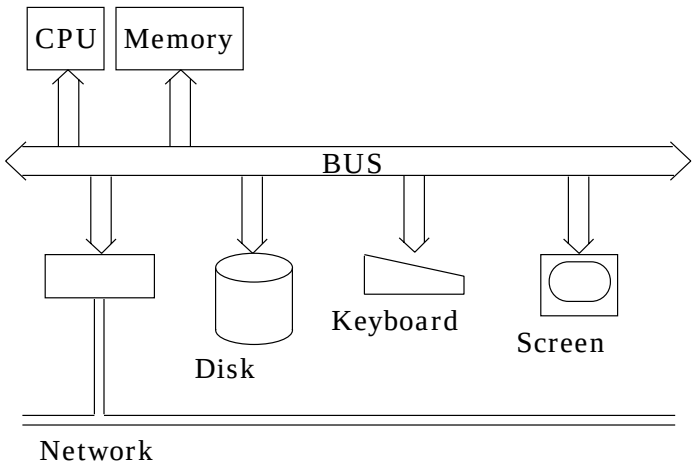


Figure 1.1: Computer Architecture - Bus Level

The components connected by the bus are those with which you may already be familiar. The two central components of a computer are its CENTRAL PROCESSING UNIT (CPU) and its MEMORY. Notice that these are separate components on the bus. The CPU is (generally) the active component in a computer, with the others being passive and responding to requests sent by the CPU. So, for example, the CPU sends a message to the memory unit requesting that a data item be transferred to the CPU; or the CPU sends to the screen a message containing the code for a character to be displayed. In this way the CPU can make use of a disk drive, a keyboard and a video screen (this is a typical configuration).

The next level of computer architectural detail is the structure of the CPU and memory as well as the mechanism which allows the combination to do work. The most common architecture for CPU's is called the

von Neumann architecture, named for the mathematician and early computer scientist John von Neumann (1903-1957). A computer does its work by manipulating data according to a sequence of instructions (a program). The main idea of the von Neumann architecture is that program instructions and data are kept in memory together; a program is EXECUTED or RUN on a set of data by means of a process called the FETCH/EXECUTE CYCLE, the details of which are described in the next section. To aid in understanding the process, we will describe a model computer, consisting of a CPU, memory, keyboard and screen. The model is a greatly simplified version of a real computer, but has all the important architectural features.

Chapter 2

The Architecture of the Itty Bitty Machine

The high-level architecture of the Itty Bitty Machine in Figure 2.1 shows four components: the CPU, a memory unit, a keyboard and a video screen, which is "connected" to part of the memory. Associated with the CPU is a set of instructions, referred to as the INSTRUCTION SET or MACHINE LANGUAGE, which can be carried out by the CPU. As described above, the job of the CPU is to sequence through a list of instructions, carrying out each instruction in turn. In this section we will describe in detail the structure of the memory unit and the CPU, including the instruction set and fetch/execute cycle.

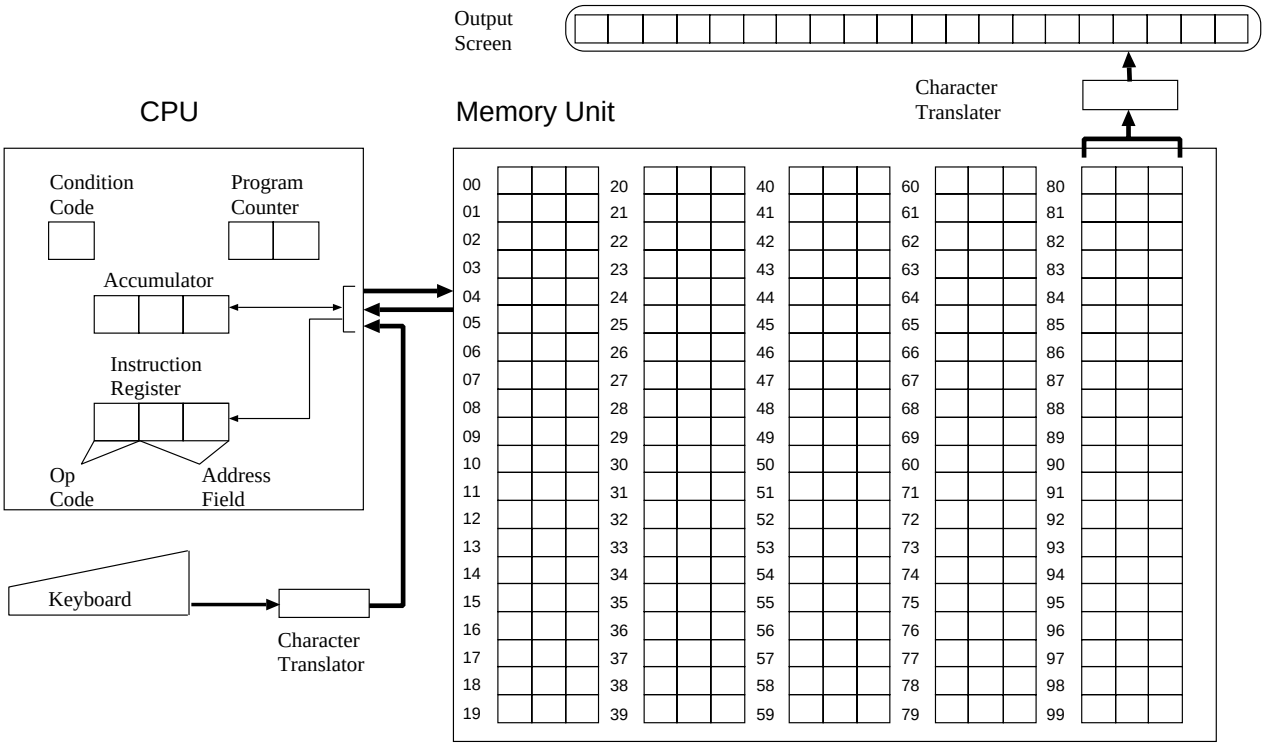


Figure 2.1: Itty Bitty Machine Architecture

2.1 Memory Unit

The memory unit of the Itty Bitty Machine consists of 100 cells called WORDs, with each word consisting of 3 DIGITs - each digit stores a decimal digit value, i.e., 0, 1, ..., 9. Each memory word has a 2 digit address (00 to 99). [Most modern computers use a binary (base-2) representation where each word consists of (usually) 32 binary digits or bits, each storing either 0 or 1.]

In fact, as is seen in Figure 2.1, memory is in two pieces. The first 80 memory locations are used as normal memory. The last 20 memory locations, though they can be used as normal memory, are also connected to the video screen in a technique known as *memory mapped I/O*. When a data value is stored in a location between address 80 and 99, a character also appears on the screen at a predetermined location (address 80 corresponds to the left-most screen location, 81 to the next, etc.). So a character is displayed on the screen by storing its numeric code in the appropriate memory location. Since only numbers can be stored in memory, there is a translation which associates with each 3-digit number a corresponding character. This translation takes place when a character is printed on the screen and also when a character is input from the keyboard.

2.2 CPU

The CPU contains a small collection of special purpose memory locations, called REGISTERS, which facilitate data manipulation and instruction sequencing. The nature of these registers, the Itty Bitty Machine instruction set and the fetch/execute cycle are described below.

Registers

The Itty Bitty Machine's CPU has four registers, each with a specific use as indicated below. The registers store data in the same way (base-10 digits) as the main memory, but the size of a register need not be the same size as a memory word.

1. ACCUMULATOR(ACC) This register acts as a scratch pad for the CPU. The accumulator size matches that of the memory word because data is transferred from memory to the accumulator and from the accumulator to memory (the process is described below). The CPU has the capability to add and subtract values from the value stored in the accumulator.

2. PROGRAM COUNTER(PC) This register holds the memory address of the next instruction to be executed by the processor. The PC is 2 digits wide in order to exactly match the size of a memory address.

3. INSTRUCTION REGISTER (IR) The instruction register holds the instruction currently being executed by the processor. The IR is broken into two parts: the left-most digit is an instruction code, or OP CODE, (described below) while the right 2 digits are (usually) interpreted as a memory address.

4. CONDITION CODE REGISTER (CC) This register holds one digit (actually either a 0 or a 1). This register is used to store a value based on the comparison of two numeric values: the CC has value 0 (zero) if the compared values are NOT equal and value 1 if the values are equal.

These registers are manipulated each time the control unit executes an instruction.

Fetch/Execute Cycle

Remember that a CPU does its work by executing a machine language program which is stored in memory. When a program is put into memory its instructions are placed in consecutive memory locations. The memory address of the first instruction is placed in the PC and then the fetch/execute cycle begins. The fetch/execute cycle consists of the following three steps (notice the references to the registers described above).

1. FETCH the value stored in memory at the address found in the PC and place the value in the IR.
2. INCREMENT the PC (add 1 to it).
3. EXECUTE the instruction in the IR.

This cycle begins when the computer is turned on and continues until the machine is turned off or a HALT instruction is executed.

Instruction Set

The CPU interprets the value in the IR according to the following table. This table is a complete description of the semantics of the Itty Bitty Machine's machine language. Assume that M is the value in the address field of the IR and that $v(M)$ denotes the value stored at address M in memory. We use a similar notation to refer to the value stored in a register (e.g., $v(ACC)$ for the value in the ACC register).

op code	explanation	action
0	**Load** value at M into the ACC	$ACC \leftarrow v(M)$
1	**Store** value in the ACC at M	$M \leftarrow v(ACC)$
2	**Add** value at M to value in the ACC and store result in ACC	$ACC \leftarrow v(ACC) + v(M)$
3	**Subtract** value at M from value in the ACC and store result in ACC [negative results are stored as 0 (zero)]	if $v(ACC) - v(M) < 0$ then $ACC \leftarrow 0$ otherwise $ACC \leftarrow v(ACC) - v(M)$
4	**Increment** value at M	$M \leftarrow v(M) + 1$
5	**Compare** value in ACC with value at M and set value in CC to 1(0) if they are equal(not equal)	if $v(ACC) = v(M)$ then $CC \leftarrow 1$ otherwise $CC \leftarrow 0$
6	**BranchE** to M if $v(CC) = 1$	if $v(CC) = 1$ then $PC \leftarrow M$
7	**BranchNE** to M if $v(CC) = 0$	if $v(CC) = 0$ then $PC \leftarrow M$
8	**Branch** to M	$PC \leftarrow M$
9	if $M = 10$ then **Clear** ACC	$ACC \leftarrow 0$
	if $M = 20$ then **Halt** - fetch/execute cycle	
	if $M = 30$ then **Blank** the screen	$v(i) = 40$ for each i from 80 to 99
	if $M = 40$ then **Shift left**	$ACC \leftarrow (v(ACC) * 10)\ mod\ 1000$
	if $M = 50$ then **Shift right**	$ACC \leftarrow v(ACC)\ div\ 10$
	if $M = 60$ then **Input** keyboard character	$ACC \leftarrow Keyboard$

2.3 Two Program Examples

The list of machine instructions which follows is a machine language program which, when executed, will input two digits and store their sum at memory address 41. Notice how memory location 40 is used to temporarily store the value of the first digit. You should consider what would happen if two letters were entered rather than two digits.

address	instruction	mnemonic	explanation
10	960	Input	input a digit from the keyboard to the ACC
11	140	Store 40	store the value at address 40
12	960	Input	input another digit from the keyboard to the ACC
13	240	Add 40	add what's at address 40 to ACC
14	141	Store 41	store result at address 41
15	920	Halt	stop fetch/execute cycle

That program was fairly straight forward. The second example is a bit trickier and uses more of the instructions in the instruction set. The following program inputs two digits and stores the larger at address 80. Notice that by storing the larger digit at address 80, the character form of the digit will appear in the first cell of the video screen. The program is written with the assumption that memory location 42 has '000' as its content and that the program will be stored in memory starting at address 10.

address	**instruction**	**mnemonic**	**explanation**
10	960	Input	input a digit
11	140	Store 40	store digit at address 40
12	960	Input	input a digit
13	141	Store 41	store digit at address 41
14	340	Subtract 40	subtract value at 40 from ACC
15	542	Compare 42	compare ACC with value at 42 (0)
16	619	BranchE 19	if ACC is 0, set PC to 19
17	041	Load 41	load value at 41
18	820	Branch 20	branch to display digit
19	040	Load 40	load value at 40
20	180	Store 80	display digit in ACC at first video cell
21	920	Halt	halt the fetch/execute cycle

Chapter 3

Representing Data

Theoretically, the IBM architecture provides all the power needed to carry out any computation. But real computers must not only be able to carry out these computations, they must be able to do them quickly and efficiently. One inefficiency inherent in the IBM architecture results from the fact that the IBM can only represent and manipulate integer values. In real computers there are several different data representations, typically for integers, real numbers, characters, and logical values (*true* and *false*). In this chapter we look at typical techniques for representing these various kinds of data values. Before discussing how these data values are represented in a computer, we will review how numbers can be represented in various number bases – a computer, of course, uses a base-2 (binary) coding for all values.

3.1 Representing Numbers on Paper

The notation we are used to for denoting numbers is based on an algebraic form. We understand the notation '437' to represent the polynomial

$$4 * 10^2 + 3 * 10^1 + 7.$$

We say that the '4' occupies the *hundreds* space, the '3' the *tens* space, and the '7' the *ones* space. The number represented by this notation can be understood as that number arrived at by counting to one hundred four times then to ten three times and finally to 7. Since this notation is based on powers of ten we refer to it as *base-10* notation. It is crucial to notice that in this notation there are ten (the base) basic symbols or *digits* used for representing numbers: $0, 1, ..., 9$. It is also important to realize that the number being represented is independent of the notation used to represent it; seven, 7, and VII all denote the same number.

Our culture is very used to using the base-10 notation for numbers, but there is, of course, nothing special about base-10 notation other than the fact that we are used to it. We could just as easily us base-60 notation, as did the Babylonians, or base-6 notation. In base-6 there would be six "digits", $0, 1, ..., 5$, used in the representation of numbers. In base-16 notation there are sixteen digits: $0, 1, ..., 9, A, B, C, D, E, F$. This probably looks weird, but it is necessary that each digit be denoted by a single symbol; letters are customarily used in denoting digits beyond 9.

The important thing to remember about different bases is that they are just different ways of representing the same numbers. The number fourteen can be represented by 14_{10} or 22_6 or 112_3 or 1110_2 (notice how the subscript is used to indicate the number base). The table in Figure 3.1 shows various representations for the numbers from zero to fourteen. The numbers zero and one have the same representation for the different bases.

In studying computer architecture we are most interested in representing numbers in base-2. But since we are most comfortable with base-10 notation it is good to also be familiar with the methods for converting from one notation to the other. The following two sections look at these conversions.

Numeric Representations					
alphabetic	Roman	base-10	base-6	base-3	base-2
zero	†	0_{10}	0_6	0_3	0_2
one	I	1_{10}	1_6	1_3	1_2
two	II	2_{10}	2_6	2_3	10_2
three	III	3_{10}	3_6	10_3	11_2
four	IV	4_{10}	4_6	11_3	100_2
five	V	5_{10}	5_6	12_3	101_2
six	VI	6_{10}	10_6	20_3	110_2
seven	VII	7_{10}	11_6	21_3	111_2
eight	$VIII$	8_{10}	12_6	22_3	1000_2
nine	IX	9_{10}	13_6	100_3	1001_2
ten	X	10_{10}	14_6	101_3	1010_2
eleven	XI	11_{10}	15_6	102_3	1011_2
twelve	XII	12_{10}	20_6	110_3	1100_2
thirteen	$XIII$	13_{10}	21_6	111_3	1101_2
fourteen	XIV	14_{10}	22_6	112_3	1110_2

† There is no Roman numeral for zero!

Figure 3.1: Examples of Numeric Representations

Base-$x \Longrightarrow$ Base-10

Converting from some other number base to base 10 is relatively easy. In a sense what we have to remember is that a number just represents the result of counting. If I write 24_7 this just means that I have counted up to the base seven twice and then up to four; or that I have counted to ten once and then to eight, i.e., 18_{10}. Similarly, the number $1B_{16}$ means counting to sixteen once and then to eleven. So, to convert 514_7 to base-10 notation we make use of the algebraic form:

$$514_7 = 5 * 7^2 + 1 * 7^1 + 4.$$

If we interpret these values in base-10 then we have

$$514_7 = 256_{10}.$$

What do we do about something like a base-16 notation? We follow the same principle:

$$3C6_{16} = 3 * 16^2 + 12 * 16 + 6 = 966.$$

Notice that '12' is used rather than 'C' because '12' is the base-10 equivalent of C_{16}.

So the algorithm for converting a number represented in base-x to an equivalent representation in base-10 can be stated as follows:

1. Write the number as a polynomial in x, where x is in its base-10 equivalent.
2. In the polynomial convert each coefficient to base-10 notation.
3. Compute the value of the polynomial, carrying out all calculations in base-10; this is the base-10 equivalent of the base-x number.

The algorithm is illustrated with the following example. To convert the number $1AC_{14}$:

1. $1 * 14^2 + A * 14^1 + C$
2. $1 * 14^2 + 10 * 14^1 + 12$
3. $196 + 140 + 12 = 348$

So $1AC_{14}$ and 348_{10} denote the same number.

Base-10 $\Longrightarrow$ Base-x

How about conversion in the other direction? The algorithm for doing this conversion is just the reverse of the algorithm above. Suppose we want to convert the number 543_{10} to an equivalent representation in base-x. We must be able to fashion a polynomial in x so that we can determine the polynomial coefficients and thus fabricate the base-x representation. As an example we will determine an equivalent representation for 543_{10} in base-8.

Step 1 Write down powers of 8 until the largest power less than or equal to 543_{10} is written:

$$1 = 8^0, 8 = 8^1, 64 = 8^2, 512 = 8^3.$$

The desired polynomial will be

$$a * 8^3 + b * 8^2 + c * 8^1 + d.$$

Step 2 The coefficient a is the integer quotient resulting from dividing 543 by $8^3 = 512$: $a = 1$.

Step 3 Subtract $a * 512$ from 543, leaving 31. Now find the largest power of 8 less than or equal to 31. Since this is 8^1, in this step we will be determining the coefficient c, since it is the coefficient of 8^1. Divide $8^1 = 8$ into 31 and make c the integer part of this quotient; $3 * 8 < 31$, so make $c = 3$.

Step 4 Subtract $c * 8$ from 31, leaving 7. Since $7 < 8$, we make $d = 7$, and the process is almost complete.

Step 5 In the above steps we have determined values for the coefficients a, c, and d. All undetermined coefficients are set to zero, i.e., we set $b = 0$.

Following these steps we have derived the polynomial

$$1 * 8^3 + 0 * 8^2 + 3 * 8^1 + 7$$

and have determined that $1037_8 = 543_{10}$.

PROBLEMS

Determine y in each of the following (show your work!).

1. $25_6 = y_{10}$
2. $1032_3 = y_{10}$
3. $AC05_{13} = y_{10}$
4. $1508_{10} = y_8$
5. $202_{10} = y_2$
6. $495_{10} = y_4$

3.2 Representing Rational Numbers

The numbers we have dealt with so far have been integers. In fact the same methods just discussed can be applied to representing rational numbers in varying bases. The reason, of course, is because the polynomial-based representation used for integers is extended to represent rational numbers. For example,

$$125.36 = 1 * 10^2 + 2 * 10^1 + 5 + 3 * 10^{-1} + 6 * 10^{-2}.$$

n	2^{-n}	a_n	sum
1	.5	$a_1 = 0$	0
2	.25	$a_2 = 1$	.25
3	.125	$a_3 = 1$	.375

Figure 3.2: Table for Converting 0.25_{10}

Base-$x \Longrightarrow$ Base-10

Conversion of representations from base-x to base-10 is really the same as described above. To convert 325.42_7 to base-10 we write out the polynomial in base-10 notation and compute the value:

$$\begin{aligned} 325.42_8 &= 3 * 8^2 + 2 * 8^1 + 5 + 4 * 8^{-1} + 2 * 8^{-2} \\ &= 3 * 64 + 2 * 8 + 5 + 4 * (1/8) + 2 * (1/8^2) \\ &= 213 + 4 * (.125) + 4 * (.125) + 2 * (.015625) \\ &= 213 + .5 + .03125 \\ &= 213.53125_{10} \end{aligned}$$

PROBLEMS

Try converting each of the following to base-10 notation.

1. 101.0101_2
2. 29.41_{16}
3. 43.0_5
4. 0.001_8

Base-10 $\Longrightarrow$ Base-2

Converting a base-10 decimal to another base takes a bit more. The problem is that, as we will see, a rational number with a finite expansion in one base can have an infinite expansion in another base. A simple example of this is 1/3; this rational number expands as follows: 0.1_3 and $0.333\cdots_{10}$. This means that when applying the technique described above for integer conversions, there will be times when an approximate representation will be all that is practical.

The conversion from base-10 to another base can be done in two steps. First convert the integer part of the number using the technique described earlier. Then do the conversion for the fractional part of the number as follows. First determine the number of "decimal" places needed in the new representation. Then write out a polynomial form for the converted value (as an example we will convert 0.375_{10} to base-2).

$$a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + \cdots$$

At this point it is convenient to set up two columns with the first containing the (base-10) decimal expansions for the first few negative powers of the base. The second column will be filled out from top to bottom with the values for the coefficients of the polynomial; Figure 3.2 shows this table. Of course when converting to base-2 the only coefficients possible are 0 and 1, so the conversion process is a matter of starting with the top row and creating a sum, where at each row we add 2^{-n} if adding the value would not exceed the base-10 value being converted. In the example of 0.375_{10} the coefficient of 0.5 must be 0 and the coefficient of 0.25 must be 1. Since 0.25 is less than the original value (0.375), we repeat the process with the next row. In this case adding 0.125 to the current sum just makes 0.375, so the process stops. For each row where we made an addition the corresponding coefficient is set to 1, while all other rows are set to 0. We have determined that

$$0.375_{10} = 0.011_2.$$

We will examine one more example, which will appear to be easier (because the initial representation is shorter) but will not be. We will convert 0.3_{10} to base-2, where the base-2 representation will have six "decimal" places. Figure 3.3 shows the conversion. The conversion to six places shows that

$$0.3_{10} \approx 0.010011_2,$$

where this approximation is off by $0.3 - .296875 = .003125$.

n	2^{-n}	a_n	sum
1	.5	$a_1 = 0$	0
2	.25	$a_2 = 1$	.25
3	.125	$a_3 = 0$	.25
4	.0625	$a_4 = 0$	.25
5	.03125	$a_5 = 1$	.28125
6	.015625	$a_6 = 1$	.296875

Figure 3.3: Table for Converting 0.3_{10}

One thing to notice about converting fractional parts from base-10 to base-2 is that, since every negative power of two end in a '5', that a base-10 number not ending in '5' will have an infinite expansion in base-2. Try this method out on the following numbers - converting to base-2 representation.

PROBLEMS

Try converting each of the following to base-2 notation.

1. 0.5625_{10}
2. 0.015625_{10}
3. 0.12_{10} (to 8 positions)
4. 28.575_{10} (to 8 positions)

3.3 Representing Data in the Computer

In this section we will look at the way data are represented in a real computer memory. Where the Itty Bitty Machine uses base-10 representation for data, real machines use base-2 representation. In the last section you should have gained an understanding of the relationship between base-10 and base-2 numeric representations. In this section we will look at the ways in which base-2 numeric representation is used to store character data and both real and integer numeric data in computer memory.

Bits, Bytes and Words

A computer's main memory is constructed as a sequence of storage units called *words*. Each word is constructed as a sequence of *bits*, where each bit stores a value of 0 or 1 (electronicly we think of this as a bit being on (= 1) or off (= 0)). It is important to know that every bit always has one or the other of these two values — never no value. As in the case of the Itty Bitty Machine, every word in a computer memory has a numeric address and it must be possible to store any address in one word of memory. So the size of a memory word defines a maximum size for the computer memory. We will usually represent memory as in Figure 3.4. Notice that each row represents a word and each word has an address.

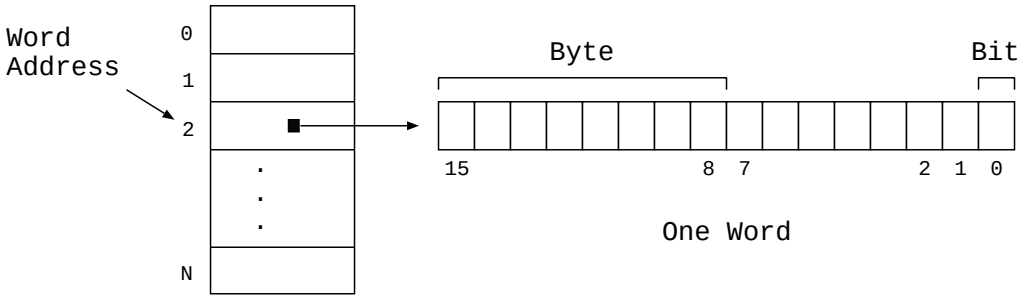


Figure 3.4: Structure of Memory

It is typical, though not universal, that the number of bits in a word is a power of two. So the SPARCstation memory word size is 32 bits; older PC's had a word size of 16 bits (really old ones had 8 bits per word). Every word in memory has the same size. For the purposes of this discussion we will assume a word size of 16 bits - 32 is a *bit* inconvenient (ha, ha!). Another division of a word which has become common is the *byte*. A byte is 8 bits, so a SPARCstation has 4 bytes per word.

Character Data

Character data, meaning the letters and other symbols which you can type at a keyboard, or that you see on a computer screen, are easy to represent in a computer. In fact we have seen the technique in the Itty Bitty Machine. The idea is to assign each symbol to be stored a numeric code. There are several such codes which have been proposed over the years, but the one which seems to be most common is the ASCII code - see the table on page 550 in the Riley text. There are 128 characters to be coded so the numbers from 0-127 are the ASCII code values (the associated characters are listed in the table mentioned above). Since $127_{10} = 1111111_2$, we allow a one bit slop, and always store one character code in a byte. In the case of the SPARCstation four characters can be stored in one word. Figure 3.5 shows several characters, their (base-10) ASCII codes and how they would be stored in one byte.

Integer Representation

The method for storing integer values in a computer memory is also straight forward. Each integer stored in a computer requires a whole word. For small integers, as we will see, this means that some space is wasted, but this is an accepted part of computer design.

To store an integer there are two things which must be represented: the magnitude of the number and the sign. In fact, the method of representation is referred to as *sign/magnitude*. Figure 3.6 shows the layout

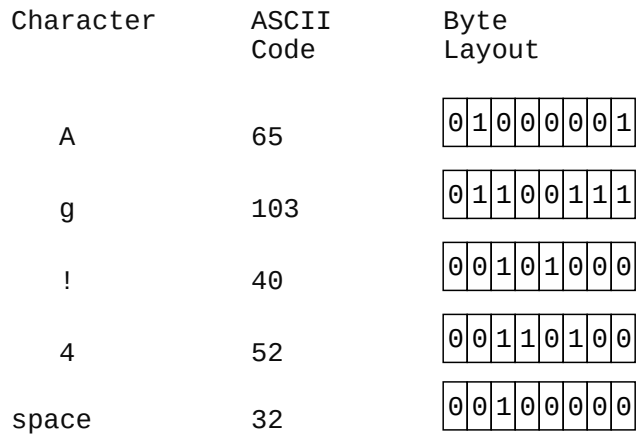


Figure 3.5: Examples of Character Storage

for integer representation in a 16-bit word. The idea is to set the *sign bit* (the left-most bit) to zero for positive numbers and one for negative numbers. The *magnitude* part holds the base-2 representation of the magnitude of the number, with the magnitude shifted as far to the right as possible. Figure 3.7 shows a few examples of how an integer is represented in a 16-bit word.

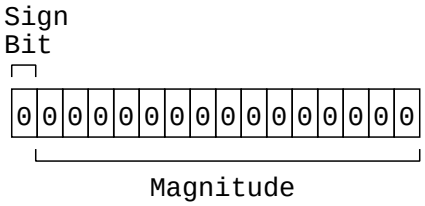


Figure 3.6: Word Format for Integers

Using this representation there is obviously a largest integer which can be stored; that is the number whose representation is all ones in the magnitude part; if the word size is 16 bits this maximum integer is $2^{15} - 1$. Thus, the integers which can be represented by a word of size n bits using sign/magnitude fall in the interval $[-2^{n-1} - 1, 2^{n-1} - 1]$. For example, with 32-word (as with the SPARCstations) the integer range is $[-2147483647, 2147483647]$.

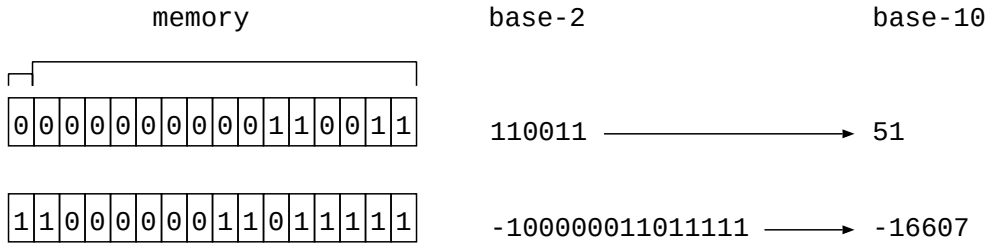


Figure 3.7: Examples of Integer Storage

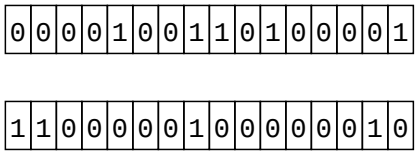
PROBLEMS

Draw the memory representation for the following integers. Assume a word size of 16 bits and sign/magnitude format.

1. 25_{10}
2. 1032_{10}
3. -1508_{10}
4. -202_{10}

PROBLEMS

What number is represented by the following bit patterns, assuming sign/magnitude format.



Real Number Representation

Representing real numbers are more of a problem. We can see this if we think about numbers like 1/3 and π, which have infinite decimal expansions. With a fixed number of bits to work with, it will be impossible to get all that accuracy. So some method of representation must be adopted which stores an approximate value. The common representation method is inspired by scientific notation. In this notation the number -13.465 will be written as $-0.13465 * 10^2$ while 0.00041037 is written as $0.41037 * 10^{-3}$. In general, we locate the leftmost non-zero digit and place the decimal point to its immediate left; this is the *mantissa*. Then we multiply the mantissa by the power of ten which will restore the original value. The same notation also works for base-2 representation. To represent this scientific notation in a word requires representing is the sign of the number, the mantissa and the exponent, both its sign and its magnitude.

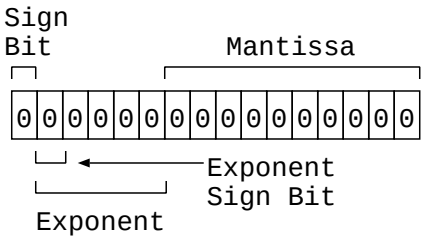
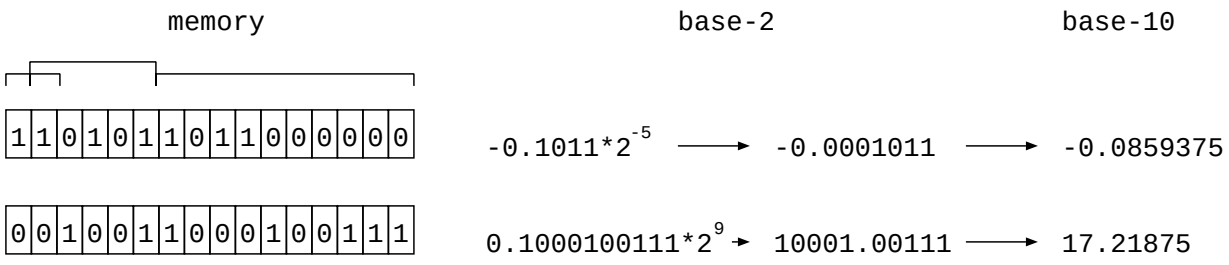


Figure 3.8: Word Format for Real Numbers

Figure 3.8 shows one way to do this. The sign of the real number is put in the left-most bit, just as for integers. The next five bits are used for the exponent. Since the exponent is an integer we use the five bits with the integer representation described above, so the left-most of the five bits is the sign of the exponent and the other four bits are the magnitude. The remaining ten bits are for the mantissa. The one tricky bit is that the mantissa is *left* justified in the 10-bit field. Here are two examples showing bit patterns and the real numbers represented.



To see how a number in base-10 notation will store in a word, consider storing the number -13.465_{10}. The following process is followed. First we convert the number to base-2 notation (using the process described above) making sure we generate enough "significant" binary digits to fill up the ten mantissa bits in the word. The fractional part of the number can be converted to 0.011101_2 as follows (an approximation, of course).

n	2^{-n}	coefficients	sum
1	.5	0	0
2	.25	1	.25
3	.125	1	.375
4	.0625	1	.4375
5	.03125	0	.4375
6	.015625	1	.453125

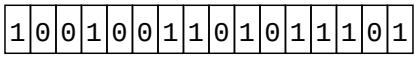
Since $-13_{10} = -1101_2$, when we combine the two expansions we get

$$-1101.011101_2$$

which has ten significant (binary) digits. The scientific notation for this number is

$$-0.1101011101 * 2^4.$$

Here is how this number would be stored in one word.



Part II

Specifying and Understanding Programs

Chapter 4

Making Programming Easier

In the previous chapter you studied the basic architectural characteristics of a computer's hardware. The CPU, memory, I/O devices and the bus provide a platform on which machine language programs are executed. When the hardware is "turned on" the fetch/execute cycle is activated and machine language instructions are fetched from the memory and executed by the CPU. The fetch/execute cycle is set up so that the CPU automatically fetches the "next" instruction each time, unless directed differently by the execution of a branch instruction. The obvious thing which has not yet been mentioned explicitly is that the whole hardware package is absolutely useless without a machine language program in the memory.

Your experience with machine language programs, though rather brief, was probably enough for you to conclude that "If that's how you have to program a computer, let me outa here!" Well, you're not alone. The early programmers, who had to write directly in machine language, had the same reaction and put some effort into developing methods for *describing* programs and then automatically translating these descriptions into the machine language. The language they developed was called an assembler language. Following naturally from the machine language, these assembler languages are referred to as *second generation* programming languages.

4.1 IBM Assembler Language

An assembler language is the simplest type of program description language. The idea is to make it possible for the programmer to generate a program without having to remember the opcodes for machine instructions or to deal with actual memory addresses. If you look back at the examples of machine language programs earlier in this chapter you will see that most are annotated with comments to help the readers, understand what is going on in the program. The assembler language described below is similar in flavor to these annotations.

Description Form - Syntax

The assembler language for the IBM is quite simple. Any language, computer or human, is defined in terms of 'words' and 'statements', where the statement is a sequence of words in some specific form (syntax). In our IBM assembler a program description is a sequence of statements, with each statement appearing on a separate line. There are two different categories of words in IBM assembler: *mnemonics* and *names*. There are 15 mnemonics, one corresponding to each different type of machine instruction. The mnemonics are summarized in the following table.

Opcode	Mnemonic
0	Load
1	Store
2	Add
3	Subtract
4	Increment
5	Compare
6	BranchE
7	BranchNE
8	Branch
910	ClearACC
920	Halt
930	BlankScreen
940	ShiftLeft
950	ShiftRight
960	Input

Names in IBM assembler are just sequences of letters and digits and are used to represent memory addresses. The only restriction is that a name cannot start with a digit nor can it be the same as one of the mnemonics. Finally, the IBM assembler has 20 predefined names: SCREEN1, SCREEN2, ..., SCREEN20 which represent the memory addresses 80...99, i.e., they are names for the 20 positions on the screen.

So we have defined the building blocks, words, of the IBM assembler, and now we can specify how these words can be combined into statements and the statements into program descriptions. Rather than writing down the details first, we will look at an example of an IBM assembler language program, one which would translate to the program at the end of Section 2.

```
+datadefs
digit1
digit2
zero            0
+instructions
                Input
                Store       digit1
                Input
                Store       digit2
                Subtract    digit1
                Compare     zero
                BranchE     its2
                Load        digit1
                Branch      show
its2            Load        digit2
show            Store       screen1
                Halt
```

In looking at this example you should first notice that much of it is easy to understand, based on your knowledge of the IBM machine language. Next, we see that the program is in two parts, with the start of each part signaled by a special name. '+DATADEFS'[1] starts every program and indicates that the next lines of the program will contain name definitions (more on that in a minute). The definition part extends up to but not including the line with '+INSTRUCTIONS'. This line marks the beginning of the program's statement section.

A statement in the +INSTRUCTIONS part has a specific form: a label (name), which is optional, followed by a mnemonic, followed by operand (name), which is required for some mnemonics and not allowed for others. A label names the memory address where the instruction is stored. Notice that the label SHOW also

[1]The use of the + as the first character of the two words is to distinguish them from words in the assembler language.

appears as the operand in the BRANCH statement. The only reason to have a label in a description is to name an address which will be referenced in one of the three branch instructions.

The operands in the other statements refer to memory locations where data is stored. Locations in memory where data is stored are referred to as *variables* and the names which reference them are called, obviously, *variable names*. The purpose of the +DATADEFS part of the program description is to define which names are variables and, as in the case of ZERO, to assign an initial value; because it is not intended that the value of ZERO change, we refer to such variables as *constants*. The following definitions sum up the terminology just presented.

Definitions

- **Variable**: A named location in memory for storing data.
- **Constant**: A Variable whose value is determined when a program description is written and doesn't change from that value during program execution.
- **Label**: A named location in memory for storing a program instruction.

4.2 Description Translation - Semantics

Remember that we use the assembler language to write program *descriptions*. These descriptions cannot be run on the IBM because they are not in a form the machine can "understand". There must be a mechanism for translating a description to an **equivalent** machine language program. When this machine language program is executed we give meaning to the description we started with. The meaning or semantics of a description language is defined by the translation process. For our assembler language the translation process involves two basic steps and results in a machine language program consisting of a sequence of address/value pairs, where the value is either a machine instruction or an initial variable value and the address is the memory location where the value is to be stored.

Assign Addresses	Assign consecutive memory addresses to each statement in the +INSTRUCTIONS part and for each variable in the +DATADEFS part. A side effect of this process will be to associate with each label an actual memory address. This step generates a list of name/address pairs.
Translate and Resolve	Create a machine language program by converting each description statement in the +DATADEFS part into an address/value pair, where the address is that associated with the particular variable name and the value is either the one specified in the definition or 0 (as a default value). Similarly convert each statement in the +INSTRUCTIONS part to an address/value pair where the address is the one already assigned the particular statement and the value is the opcode for the statement mnemonic or, if appropriate, the opcode along with the address associated with the statement's operand.

We can follow this process on the program description at the beginning of this section. Notice, that the program we generate may not be exactly like the program at the end of the previous chapter because we may not choose the same starting address for the program or the same addresses for the variables.

			Assign Addresses ⟹			Translate ⟹	
+datadefs							
digit1			12				
digit2			13				
zero	0		14	0			
+instructions							
	Input		0	Input		0	960
	Store	digit1	1	Store	digit1	1	1-12
	Input		2	Input		2	960
	Store	digit2	3	Store	digit2	3	1-13
	Subtract	digit1	4	Subtract	digit1	4	3-12
	Compare	zero	5	Compare	zero	5	5-14
	BranchE	its2	6	BranchE	its2	6	6-09
	Load	digit1	7	Load	digit1	7	0-12
	Branch	show	8	Branch	show	8	8-10
its2	Load	digit2	9	Load	digit2	9	0-13
show	Store	screen1	10	Store	screen1	10	1-80
	Halt		11	Halt		11	920
						12	0
			its2	9		13	0
			show	10		14	0
			digit1	12			
			digit2	13			
			zero	14			

This translation process is simple and, in fact, can be easily automated. Before the end of this semester you will have learned all the techniques necessary to write a C^{++} description for a program which carries out the translation process just described. You would then be able to write an assembler language description, translate it with your program and then run the resulting IBM machine language program on the IBM simulator.

4.3 Virtual Machine Levels

In this section we have developed a new language which can be used to write machine language programs. Well, not quite. We can use the assembler language to write a program and then translate it to an equivalent machine language program. This process frees us from the problems associated with memory addresses, specifically, where the data and program are located in memory.

What the assembler language allows us to do is to use mnemonics in the place of machine codes and variable and label names in the place of memory addresses. In a way, the assembler language is a "machine" language for a new kind of machine which has a less well defined notion of memory. This machine has the same organization as the IBM, but in place of main memory's address/value pairs the new machine has identifier/value pairs. We have abstracted the notions of address and op-code to a more convenient level where they are represented by identifiers (names).

The history of programming can be seen as a continuing attempt to abstract a real machine to a level where it is really easy to program. When you learn a high-level language such as C^{++} you see that addresses continue to be represented by identifiers. The kind of data that the virtual machine can handle is much more extensive (integer, reals, character, boolean) with the added capability for the user to define new data types. The real machine's op-codes are abstracted, not to mnemonics as in the assembler language, but to new instructions not obviously related to those of the underlying machine.

Chapter 5

A Formal Approach to Programming

Of course you are not in this course to learn about programming in assembler language. But this description of the IBM assembler language illustrates several basic aspects of programming languages. First, a programming language is used to write descriptions which can then be translated (in an automated way) into machine language programs. The descriptions themselves cannot be run on a computer. Though programming languages vary considerably in form, they all must in some way describe two separate aspects of a program description, namely the variables and the instructions. When we look at Pascal you will see that every Pascal description has these two basic parts: data description and statement description.

5.1 Syntax

In the previous section it became clear that an IBM assembler description has a very specific form. This form is determined by the *syntax* of the language. Every programming language has a very specific syntactic structure, just as English, German and Chinese have specific syntactic structure. This structure starts, at the most detailed level, with a listing of the legal characters which can appear in the language, usually alphabetic and punctuation characters along with the ten digits. The syntax goes on to specify how these characters can be legally combined into words, how words can be legally combined, perhaps with a smattering of punctuation symbols, into statements, and how the statements can be combined into program descriptions.

5.2 Semantics

Of course, form is of little use without a corresponding meaning. The notion of syntax is easily recognized in programming languages[1], but the meaning of a particular program description...what does *that* even mean? What we mean by the *semantics* or meaning of a program description is the intended effect on the machine of executing the corresponding machine language program. Since a machine executes one machine instruction at a time, presumably we can talk about the semantics of not just a complete description, but even of a particular statement.

In order to talk about the effect on a machine of executing an instruction, we must understand what in a machine can be affected by executing an instruction. When put this way the answer is clear - the only things which change when an instruction is executed are register and memory values (and, indirectly, the screen and keyboard). We will refer those elements of the machine which can be changed by executing a program as the program's *state*. The state should include all the registers, since they can change, and any memory addresses which can be modified by the program. If we reflect on the previous section we arrive at the following preliminary definition.

[1]Both art and music are forms of communication in which the language has a structure which is not necessarily obvious to the viewer or listener. We know that color and tones are in there, but their combinations don't always have an apparent structure.

Program State

The state of a program is the set of register/value pairs for the CPU along with the set of address/value pairs for those addresses modified by the program.

Using this we can give a definition of what we mean by the semantics of a sequence of statements.

Semantics

The semantics of a sequence of statements is the change in the program state resulting from the execution of the statements.

Now, if you refer back to the section on the IBM instruction set in the first chapter, you will see that the table there, in fact, describes in a very precise way the semantics of each instruction in the machine language.

Though this discussion has been oriented around the IBM machine and its assembler language, it is important that these concerns of syntax and semantics will pervade our studies throughout the semester. The basic notions of words, statements, and program state will be discussed in the context of the language you study.