**Chapter One**

**Introductory Concepts**

1. **Introductory Concepts**
   1. **Introduction**

Before we go in to the detailed discussions entailing Computer Organization and Assembly Language Programming, let’s discuss an introductory concept that incorporates most of the ideas we will discuss in the subsequent sessions, but based on a fictitious microprocessor.

In general, the following points should be emphasized before learning an Assembly Language programming:

* Getting to know the architecture of a Microprocessor (the registers, the buses, the bus width etc.. and how it communicates with Memory)
* Getting to know the assembly instruction set of the Microprocessor and their syntaxes.
* Getting to know the assembler platform you are working on.

The subsequent discussions will brief you on these ideas based on the fictitious Microprocessor. **But please keep in mind that, by no means the processor and its instructions we discussed here will not be confused with the processor and instructions for our course – the Intel 8086 Microprocessor and its instructions.**

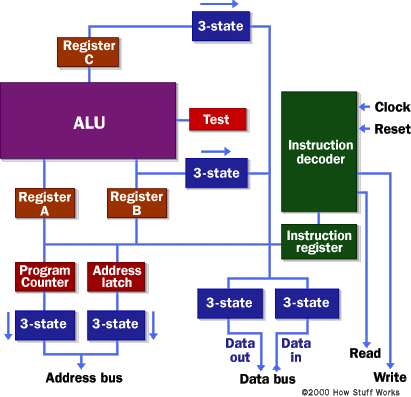
* 1. **Microprocessor Logic**

To understand how a microprocessor works, it is helpful to look inside and learn about the logic used to create one. In the process you can also learn about **assembly language** -- the native language of a microprocessor -- and many of the things that engineers can do to boost the speed of a processor.

A microprocessor executes a collection of machine instructions that tell the processor what to do. Based on the instructions, a microprocessor does three basic things:

* Using its ALU (Arithmetic/Logic Unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication and division. Modern microprocessors contain complete floating point processors that can perform extremely sophisticated operations on large floating point numbers.
* A microprocessor can move data from one memory location to another.
* A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

There may be very sophisticated things that a microprocessor does, but those are its three basic activities. The following diagram shows an **extremely simple fictitious microprocessor** capable of doing those three things:



This is about as simple as a microprocessor gets. This microprocessor has:

* An **address bus** (that may be 8, 16 or 32 bits wide) that sends an address to memory
* A **data bus** (that may be 8, 16 or 32 bits wide) that can send data to memory or receive data from memory
* An **RD** (read) and **WR** (write) line to tell the memory whether it wants to set or get the addressed location
* A **clock line** that lets a clock pulse sequence the processor
* A **reset line** that resets the program counter to zero (or whatever) and restarts execution

Let's assume that both the address and data buses are 8 bits wide in this example.

Here are the components of this simple microprocessor:

* Registers A, B and C are simply latches made out of flip-flops.
* The address latch is just like registers A, B and C.
* The program counter is a latch with the extra ability to increment by 1 when told to do so, and also to reset to zero when told to do so.
* The ALU could be as simple as an 8-bit adder or it might be able to add, subtract, multiply and divide 8-bit values. Let's assume the latter here.
* The test register is a special latch that can hold values from comparisons performed in the ALU. An ALU can normally compare two numbers and determine if they are equal, if one is greater than the other, etc. The test register can also normally hold a carry bit from the last stage of the adder. It stores these values in flip-flops and then the instruction decoder can use the values to make decisions. In general terms, this is called Flag register.
* There are six boxes marked "3-State" in the diagram. These are **tri-state buffers**. A tri-state buffer can pass a 1, a 0 or it can essentially disconnect its output (imagine a switch that totally disconnects the output line from the wire that the output is heading toward). A tri-state buffer allows multiple outputs to connect to a wire, but only one of them to actually drive a 1 or a 0 onto the line. (Like a multiplexer, but with better capabilities).
* The instruction register and instruction decoder are responsible for controlling all of the other components.

Although they are not shown in this diagram, there would be control lines from the instruction decoder that would:

* Tell the A register to latch the value currently on the data bus
* Tell the B register to latch the value currently on the data bus
* Tell the C register to latch the value currently output by the ALU
* Tell the program counter register to latch the value currently on the data bus
* Tell the address register to latch the value currently on the data bus
* Tell the instruction register to latch the value currently on the data bus
* Tell the program counter to increment
* Tell the program counter to reset to zero
* Activate any of the six tri-state buffers (six separate lines)
* Tell the ALU what operation to perform
* Tell the test register to latch the ALU's test bits
* Activate the RD line
* Activate the WR line

Coming into the instruction decoder are the bits from the test register and the clock line, as well as the bits from the instruction register.

* 1. **Microprocessor Memory**
     1. **ROM chip**

The previous section talked about the address and data buses, as well as the RD and WR lines. These buses and lines connect either to RAM or ROM -- generally both. In our sample microprocessor, we have an address bus 8 bits wide and a data bus 8 bits wide. That means that the microprocessor can address (28) 256 bytes of memory, and it can read or write 8 bits of the memory at a time. Let's assume that this simple microprocessor has 128 bytes of ROM starting at address 0 and 128 bytes of RAM starting at address 128.

ROM stands for read-only memory. A ROM chip is programmed with a permanent collection of pre-set bytes. The address bus tells the ROM chip which byte to get and place on the data bus. When the RD line changes state, the ROM chip presents the selected byte onto the data bus.

* + 1. **RAM chip**

RAM stands for random-access memory. RAM contains bytes of information, and the microprocessor can read or write to those bytes depending on whether the RD or WR line is signaled. One problem with RAM chips is that they forget everything once the power goes off. That is why the computer needs ROM.

By the way, nearly all computers contain some amount of ROM (it is possible to create a simple computer that contains no RAM -- many microcontrollers do this by placing a handful of RAM bytes on the processor chip itself -- but generally impossible to create one that contains no ROM). On a PC, the ROM holds the BIOS. When the microprocessor starts, it begins executing instructions it finds in the BIOS. The BIOS instructions do things like test the hardware in the machine, and then it goes to the hard disk to fetch the **boot sector**. This boot sector is another small program, and the BIOS stores it in RAM after reading it off the disk. The microprocessor then begins executing the boot sector's instructions from RAM. The boot sector program will tell the microprocessor to fetch something else from the hard disk into RAM, which the microprocessor then executes, and so on. This is how the microprocessor loads and executes the entire operating system.

* 1. **Microprocessor Instructions**

Even the incredibly simple microprocessor shown in the previous example will have a fairly large set of instructions that it can perform. The collection of instructions is implemented as **bit** **patterns**, each one of which has a different meaning when loaded into the instruction register. Humans are not particularly good at remembering bit patterns, so a set of short words are defined to represent the different bit patterns. This collection of words is called the **assembly language** of the processor. An **assembler** can translate the words into their bit patterns very easily, and then the output of the assembler is placed in memory for the microprocessor to execute.

Here's the set of assembly language instructions that the designer *might* create for the simple microprocessor in our example:

* **LOADA mem** - Load register A from memory address
* **LOADB mem** - Load register B from memory address
* **CONB con** - Load a constant value into register B
* **SAVEB mem** - Save register B to memory address
* **SAVEC mem** - Save register C to memory address
* **ADD** - Add A and B and store the result in C
* **SUB** - Subtract A and B and store the result in C
* **MUL** - Multiply A and B and store the result in C
* **DIV** - Divide A and B and store the result in C
* **COM** - Compare A and B and store the result in test
* **JUMP addr** - Jump to an address
* **JEQ addr** - Jump, if equal, to address
* **JNEQ** addr - Jump, if not equal, to address
* **JG addr** - Jump, if greater than, to address
* **JGE addr** - Jump, if greater than or equal, to address
* **JL addr** - Jump, if less than, to address
* **JLE addr** - Jump, if less than or equal, to address
* **STOP** - Stop execution

Now to understand how assembly programming functions, let’s try to find a solution to a problem of finding the factorial of 5 (5!).

Giving a solution to the problem with this simple piece of C code and calculate the factorial of 5 (where the factorial of 5 = 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120):

a=1;  
f=1;  
while (a <= 5)  
{  
 f = f \* a;  
 a = a + 1;  
}

At the end of the program's execution, the variable **f** contains the factorial of 5. As noticed, we only wrote seven lines of codes.

* + 1. **Assembly Language**

A **C compiler** translates this C code into assembly language. Assuming that RAM starts at address 128 in this processor, and ROM (which contains the assembler) starts at address 0, then for our simple microprocessor the assembly language might look like this:

// Assume a is at address 128  
// Assume F is at address 129  
0 CONB 1 // a=1;  
1 SAVEB 128  
2 CONB 1 // f=1;  
3 SAVEB 129  
4 LOADA 128 // if a > 5 the jump to 17  
5 CONB 5  
6 COM  
7 JG 17  
8 LOADA 129 // f=f\*a;  
9 LOADB 128  
10 MUL  
11 SAVEC 129  
12 LOADA 128 // a=a+1;  
13 CONB 1  
14 ADD  
15 SAVEC 128  
16 JUMP 4 // loop back to if  
17 STOP

* + 1. **Machine Language/ROM**

So now the question is, "How do all of these instructions look in ROM?" Each of these assembly language instructions must be represented by a **binary** number. For the sake of simplicity, let's assume each assembly language instruction is given ***a unique number***, like this:

* LOADA - 1
* LOADB - 2
* CONB - 3
* SAVEB - 4
* SAVEC mem - 5
* ADD - 6
* SUB - 7
* MUL - 8
* DIV - 9
* COM - 10
* JUMP addr - 11
* JEQ addr - 12
* JNEQ addr - 13
* JG addr - 14
* JGE addr - 15
* JL addr - 16
* JLE addr - 17
* STOP - 18

The numbers are known as **opcodes**. In ROM, our little program would look like this:

// Assume a is at address 128  
// Assume F is at address 129  
Addr opcode/value  
0 3 // CONB 1  
1 1  
2 4 // SAVEB 128  
3 128  
4 3 // CONB 1  
5 1  
6 4 // SAVEB 129  
7 129  
8 1 // LOADA 128  
9 128  
10 3 // CONB 5  
11 5  
12 10 // COM  
13 14 // JG 17  
14 31  
15 1 // LOADA 129  
16 129  
17 2 // LOADB 128  
18 128  
19 8 // MUL  
20 5 // SAVEC 129  
21 129  
22 1 // LOADA 128  
23 128  
24 3 // CONB 1  
25 1  
26 6 // ADD  
27 5 // SAVEC 128  
28 128  
29 11 // JUMP 4  
30 8  
31 18 // STOP

**You can see that seven lines of C code became 18 lines of assembly language, and that became 32 bytes in ROM.**

The subsequent chapters of the course will focus on how the core aspects of a given computing machine – the Microprocessor, Memory and I/O will function.

But right away one question should come to your mind – why learning Assembly Language Programming, while we still can do programming with the easier higher level programming language.

**What is Assembly Language?**

* **It is a machine (CPU) Specific programming language.**
* **There is a one-to-one correspondence between statements (assembly instruction) and machine language.**
* **Matches machine instructions and underlying architecture.**
* **More difficult to learn than High Level Programming language.**

**What is an Assembler?**

* **System level software that translates assembly language source code to machine language.**

**Why Learn Assembly?**

* **Learn how a microprocessor works.**
* **Understand basic computer Architecture.**
* **Explore the internal representation of data and instructions.**
* **Gain an insight into hardware concepts.**
* **Allows creation of small and efficient programs.**
* **Allow programmers to bypass high-level programming language restrictions.**
* **Might be necessary to accomplish certain tasks, for example, to program embedded processors inside different machines.**

**Computer Architecture:-is the study of blocks or components that make up a computer system and how they are interconnected. Two famous architectures are broadly known:**

**1) Von Neumann or Stored Program Architecture:**

**2) Harvard Architecture**

**Computer Organization:-is concerned with the implementation of computer architecture.**

**Computer Engineering:-is a field of study that concerned with the actual construction of the system. Examples include length of wires, size of circuits, cooling, electrical requirements etc.**

* 1. **Additional Note on Assembly language**

There is some debate over the usefulness of assembly language. It is often said that modern compilers can render higher-level languages into codes that run as fast as hand-written assembly, but counter-examples can be made, and there is no clear consensus on this topic. It is reasonably certain that, given the increase in complexity of modern processors, effective hand-optimization is increasingly difficult and requires a great deal of knowledge.

However, some discrete calculations can still be rendered into faster running code with assembly, and some low-level programming is simply easier to do with assembly. Some system-dependent tasks performed by operating systems simply cannot be expressed in high-level languages. In particular, assembly is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers. Many compilers also render high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes.

It's also common, especially in relatively low-level languages such as C, to be able to embed assembly language into the source code with special syntax.

Many embedded systems are also programmed in assembly to obtain the absolute maximum functionality out of what are often very limited computational resources, though this is gradually changing in some areas as more powerful chips become available for the same minimal cost.

Another common area of assembly language use is in the system BIOS of a computer. This low-level code is used to initialize and test the system hardware prior to booting the OS and is stored in ROM. Once a certain level of hardware initialization has taken place, code written in higher level languages can be used, but almost always the code running immediately after power is applied is written in assembly language. This is usually due to the fact that system RAM may not yet be initialized at power-up and assembly language can execute without explicit use of memory, especially in the form of a stack. This use becomes inherent in programming microprocessors embedded with machines with no viable operating system and high level compiler.

Computer systems vendors may charge high prices for compiler language runtime libraries, thereby virtually assuring not every installation supports applications that are written in a particular language, except assembly language. Under this premise, assembly language is forced on Independent Software Vendors to keep the prospective buyer's costs down. What is good from a software engineering viewpoint is bad for business.

###### Assembly language is also valuable in reverse engineering, since many programs are distributed only in machine code form, and machine code is usually easy to translate into assembly language and carefully examine in this form, but very difficult to translate into a higher-level language. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose.

* + 1. **Programming a Computer**
       1. **Machine Language**

A system of codes directly understandable by a computer's CPU is termed as this CPU's **native** or **machine language**. Although machine code may seem similar to *assembly language* they are in fact two different types of languages. *Assembly code* consists of both binary numbers and simple words whereas *machine code* is composed only of the two binary digits 0 and 1. Every CPU model has its own machine language, although there is considerable overlap between some. If CPU *A* understands the full language of CPU *B* it is said that *A* is compatible with *B*. CPU *B* may not be compatible with CPU *A*, as *A* may know a few codes that *B* does not.

The "words" of a machine language are called *instructions*; each of these causes an elementary action by the CPU, such as reading from a memory location. A program is just a long list of instructions that are *executed* by a CPU. Older processors executed instructions one after the other, but newer superscalar processors are capable of executing several instructions at once. Program flow may be influenced by special *jump* instructions that transfer execution to an instruction other than the following one. Conditional jumps are taken or not depending on some condition.

Instructions are simply a pattern of bits - different patterns correspond to different commands to the machine. Humans use mnemonic codes to refer to the useful bit-patterns: this more readable rendition of the machine language is called assembly language. For example, in one processor, the machine code 00000101 causes the CPU to decrement the B register. In assembly language we write this DEC B.

* + - 1. **Assembly Language**

**Assembly language** or simply **assembly** is a human-readable notation for the machine language that a ***specific computer architecture uses.*** Machine language, a pattern of bits encoding machine operations, is made readable by replacing the raw values with symbols called *mnemonics*.

For example, a computer with the appropriate processor will understand this x86 machine instruction:

10110000 01100001

For programmers, however, it is easier to remember the equivalent assembly language representation:

mov al, $61

which means to move the hexadecimal value 61 (97 decimal) into the processor register with the name "al". The mnemonic "mov" is short for "move", and a comma-separated list of arguments or parameters follows it; this is a typical assembly language statement.

Transforming assembly into machine language is accomplished by an assembler, and the reverse by a disassembler. Unlike in high-level languages, there is usually a 1-to-1 correspondence between simple assembly statements and machine language instructions. However, in some cases an assembler may provide *pseudoinstructions* which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)".

Every computer architecture has its own machine language, and therefore its own assembly language. Computers differ by the number and type of operations that they support. They may also have different sizes and numbers of registers and different representations of data types in storage. While all general-purpose computers are able to carry out essentially the same functionality, the way they do it differs, and the corresponding assembly language must reflect these differences.

In addition, multiple sets of mnemonics or assembly-language syntax may exist for a single instruction set. In these cases, the most popular one is usually that used by the manufacturer in their documentation.

* + - 1. **High Level Programming Language**

These are human understandable programming languages such as Pascal and Fortran. These statements will be translated into a machine readable executable program by Compilers. Each statement in a high level language corresponds to several machine language instructions. Using such languages a programmer can write programs more quickly and concentrates on the problem that the program is designed to solve, instead of how the computer will carry out the program. However, programming written with high level programming languages run slower that those written in a low-level programming language.

* + - 1. **Why Learn Assembly?**
* Learn how a microprocessor works
* Understand basic computer architecture
* Explore the internal representation of data and instructions
* Gain an insight into hardware concept
* Allows creation of small and efficient programs
* Allows programmers to pass high level programming language instructions
* For writing low level system programs such as firmware software and device drivers
* For programming microprocessor units embedded with factory machines, vehicles, airplanes etc.
  1. **Detail Concepts For This Chapter (Optional Part)**
     1. **Data Organization**

In pure mathematics a value may take an arbitrary number of bits. Computers, on the other hand, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called nibbles), groups of eight bits (called bytes), groups of 16 bits (called words), and more. The sizes are not arbitrary. There is a good reason for these particular values. This section will describe the bit groups commonly used on the Intel 80x86 chips.

* + 1. **Bits**

The smallest unit of data on a binary computer is a single bit. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit. With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are not limited to representing binary data types (that is, those objects which have only two distinct values). You could use a single bit to represent the numbers 723 and 1,245. or perhaps 6,254 and 5. You could also use a single bit to represent the colors red and blue. You could even represent two unrelated objects with a single bit,. For example, you could represent the color red and the number 3,256 with a single bit. You can represent any two different values with a single bit. However, you can represent only two different values with a single bit.

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can’t. But this illustrates the whole idea behind computer data structures: data is what you define it to be. If you use a bit to represent a Boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any true meaning, you must be consistent. That is, if you are using a bit to represent true or false at one point in your program, you shouldn’t use the true/false value stored in that bit to represent red or blue later.

Since most items you will be trying to model require more than two different values, single bit values are not the most popular data type you will use. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. However, you will soon see that individual bits are difficult to manipulate, so we will often use other data types to represent Boolean values.

* + 1. **Nibbles**

A nibble is a collection of four bits. It wouldn’t be a particularly interesting data structure except for two items: BCD (binary coded decimal) numbers and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits (see The Hexadecimal Numbering System on page 17). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits. In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

* + 1. **Bytes**

Without question, the most important data structure used by the 80x86 microprocessor is the byte. A byte consists of eight bits and is the smallest addressable datum (data item) on the 80x86 microprocessor. Main memory and I/O addresses on the 80x86 are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an eight-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven using the convention in Figure 1.1. Bit 0 is the low order bit or least significant bit, bit 7 is the high order bit or most significant bit of the byte. We will refer to all other bits by their number.

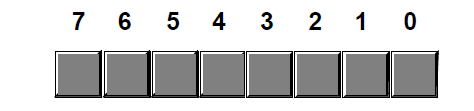


Figure 1.1: Bit Numbering in a Byte

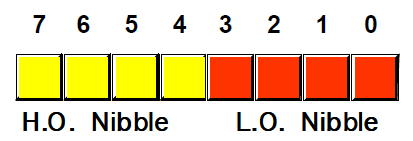


Figure 1.2: The Two Nibbles in a Byte

Note that a byte also contains exactly two nibbles (see Figure 1.2). Bits 0..3 comprise the low order nibble, bits 4..7 form the high order nibble. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits. Since a byte contains eight bits, it can represent 28, or 256, different values. Generally, we will use a byte to represent numeric values in the range 0..255, signed numbers in the range -128..+127 (see Signed and Unsigned Numbers on page 23), ASCII/IBM character codes, and other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the 80x86 is a byte addressable machine (see Memory Layout and Access on page 145), it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, well often represent the boolean values true and false by 000000012 and 000000002 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard; displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, the IBM PC uses a variant of the ASCII character set . There are 28 defined codes in the ASCII character set. IBM uses the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols. See Appendix A for the character/code assignments.

* + 1. **Words**

A word is a group of 16 bits. We will number the bits in a word starting from zero on up to fifteen. The bit numbering appears in Figure 1.3.

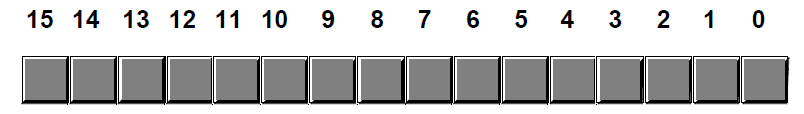


Figure 1.3: Bit Numbers in a Word

Like the byte, bit 0 is the low order bit and bit 15 is the high order bit. When referencing the other bits in a word use their bit position number.

Notice that a word contains exactly two bytes. Bits 0 through 7 forms the low order byte, bits 8 through 15 forms the high order byte (see Figure 1.4). Naturally, a word may be further broken down into four nibbles as shown in Figure 1.5. Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. The other two nibbles are nibble one or nibble two. With 16 bits, you can represent 216 (65,536) different values. These could be the values in the range 0..65,535 (or, as is usually the case, -32,768..+32,767) or any other data type with no more than 65,536 values. The three major uses for words are integer values, offsets, and segment values (next chapter description of segments and offsets). Words can represent integer values in the range 0..65,535 or -32,768..32,767. Unsigned numeric values are represented by the binary value corresponding to the bits in the word. Signed numeric values use the two’s complement form for numeric values. Segment values, which are always 16 bits long, constitute the paragraph address of a code, data, extra, or stack segment in memory.

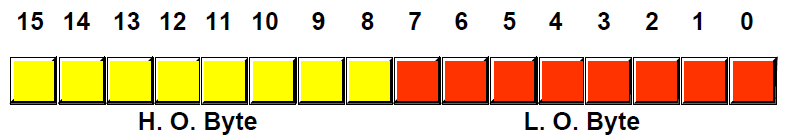
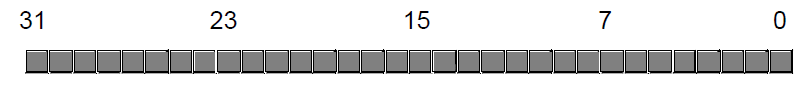


Figure 1.4: The Two Bytes in a Word

* + 1. **Double Words**

A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Figure 1.6.

 Figure 1.6: Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, or four different bytes, or eight different nibbles (see Figure 1.7). Double words can represent all kinds of different things. First and foremost on the list is a segmented address. Another common item represented with a double word is a 32-bit integer value (which allows unsigned numbers in the range 0..4,294,967,295 or signed numbers in the range -2,147,483,648..2,147,483,647). 32-bit floating point values also point into a double word. Most of the time, we will use double words to hold segmented addresses.

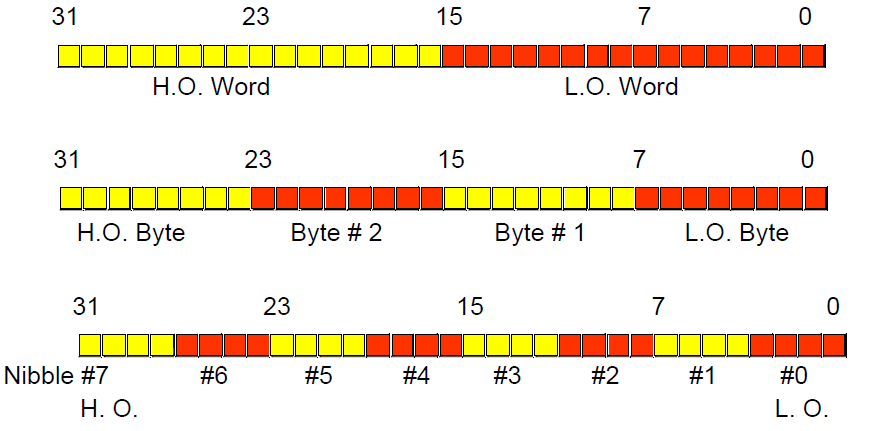


Figure 1.7: Nibbles, Bytes, and Words in a Double Word

.

* + 1. **The Hexadecimal Numbering System**

A big problem with the binary system is verbosity. To represent the value 20210 requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy. Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we are looking for: they are very compact, and it is simple to convert them to binary and vice versa. Because of this, most binary computer systems today use the hexadecimal numbering system2. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 123416 is equal to:

1 \* 163 + 2 \* 162 + 3 \* 161 + 4 \* 160

or

4096 + 512 + 48 + 4 = 466010.

Each hexadecimal digit can represent one of sixteen values between 0 and 1510. Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 1010 through 1510. Rather than create new symbols for these digits, we will use the letters A through F. The following are all examples of valid hexadecimal numbers: 2. Digital Equipment is the only major holdout. They still use octal numbers in most of their systems. A legacy of the days when they produced 12-bit machines.



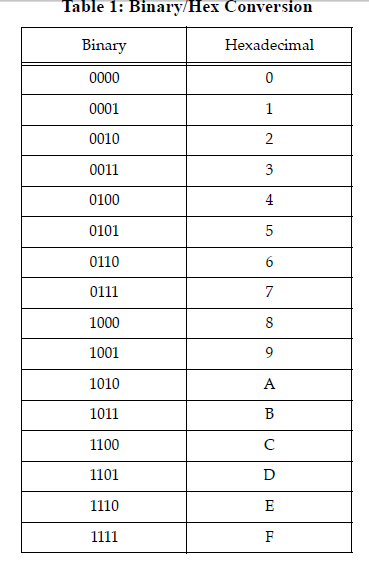
Since we will often need to enter hexadecimal numbers into the computer system, we will need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We will adopt the following conventions:

* All numeric values (regardless of their radix) begin with a decimal digit.
* All hexadecimal values end with the letter h, e.g., 123A4h3.
* All binary values end with the letter b.
* Decimal numbers may have a t or d suffix.

Examples of valid hexadecimal numbers:

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table: This table provides all the information you will ever need to convert any hexadecimal number into a binary number or vice versa.



To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert 0ABCDh into a binary value, simply convert each hexadecimal digit according to the table above:



To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010 1100 1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, e.g., 2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a calculator that will do the conversion for you, you’ll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

* + 1. **Arithmetic Operations on Binary and Hexadecimal Numbers**

There are several operations we can perform on binary and hexadecimal numbers. For example, we can add, subtract, multiply, divide, and perform other arithmetic operations. Although you needn’t become an expert at it, you should be able to, in a pinch, perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually, the correct way to perform such arithmetic operations is to have a calculator which does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Manufacturers of Hexadecimal Calculators:

* Casio
* Hewlett-Packard
* Sharp
* Texas Instruments

This list is, by no means, exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch. However, they are more expensive than the others. Sharp and Casio produce units which sell for well under $50. If you plan on doing any assembly language programming at all, owning one of these calculators is essential.

Another alternative to purchasing a hexadecimal calculator is to obtain a TSR (Terminate and Stay Resident) program such as SideKicktm which contains a built-in calculator. However, unless you already have one of these programs, or you need some of the other features they offer, such programs are not a particularly good value since they cost more than an actual calculator and are not as convenient to use. To understand why you should spend the money on a calculator, consider the following arithmetic problem:

9h

+ 1h

----

You’re probably tempted to write in the answer ‘10h’ as the solution to this problem. But that is not correct! The correct answer is ten, which is ‘0Ah’, not sixteen which is ‘10h’. A similar problem exists with the arithmetic problem:

10h

- 1h

----

You’re probably tempted to answer ‘9h’ even though the true answer is ‘0Fh’. Remember, this problem is asking ‘what is the difference between sixteen and one?’ The answer, of course, is fifteen which is ‘0Fh’.

Even if the two problems above don’t bother you, in a stressful situation your brain will switch back into decimal mode while you’re thinking about something else and you’ll produce the incorrect result. Moral of the story - if you must do an arithmetic computation using hexadecimal numbers by hand, take your time and be careful about it. Either that, or convert the numbers to decimal, perform the operation in decimal, and convert them back to hexadecimal.

You should never perform binary arithmetic computations. Since binary numbers usually contain long strings of bits, there is too much of an opportunity for you to make a mistake. Always convert binary numbers to hex, perform the operation in hex (preferably with a hex calculator) and convert the result back to binary, if necessary.

* + 1. **Shifts and Rotates**

Another set of logical operations which apply to bit strings are the shift and rotate operations. These two categories can be further broken down into left shifts, left rotates, right shifts, and right rotates. These operations turn out to be extremely useful to assembly language programmers. The left shift operation moves each bit in a bit string one position to the left (see Figure 1.8).

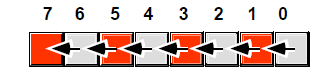


Figure 1.8: Shift Left Operation

Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, etc. There are, of course, two questions that naturally arise: ‘What goes into bit zero?’ and ‘Where does bit seven winds up?’ Well, that depends on the context. We’ll shift the value zero into the L.O. bit, and the previous value of bit seven will be the carry out of this operation.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a zero to the right of the number) effectively multiplies it by ten (the radix):

1234 SHL 1 = 12340 (SHL 1 = shift left one position)

Since the radix of a binary number is two, shifting it left multiplies it by two. If you shift a binary value to the left twice, you multiply it by two twice (i.e., you multiply it by four). If you shift a binary value to the left three times, you multiply it by eight (2\*2\*2). In general, if you shift a value to the left n times, you multiply that value by 2n.

A right shift operation works the same way, except we’re moving the data in the opposite direction. Bit seven moves into bit six, bit six moves into bit five, bit five moves into bit four, etc. During a right shift, we’ll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 1.9).

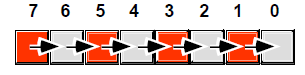


Figure 1.9: Shift Right Operation

Since a left shift is equivalent to a multiplication by two, it should come as no surprise that a right shift is roughly comparable to a division by two (or, in general, a division by the radix of the number). If you perform n right shifts, you will divide that number by 2n. There is one problem with shift rights with respect to division: as described above a shift right is only equivalent to an unsigned division by two.

For example, if you shift the unsigned representation of 254 (0FEh) one place to the right, you get 127 (07Fh), exactly what you would expect. However, if you shift the binary representation of -2 (0FEh) to the right one position, you get 127 (07Fh), which is not correct. This problem occurs because we’re shifting a zero into bit seven. If bit seven previously contained a one, we’re changing it from a negative to a positive number. Not a good thing when dividing by two.

To use the shift right as a division operator, we must define a third shift operation: arithmetic shift right7. An arithmetic shift right works just like the normal shift right operation (a logical shift right) with one exception: instead of shifting a zero into bit seven, an arithmetic shift right operation leaves bit seven alone, that is, during the shift operation it does not modify the value of bit seven as Figure 1.10 shows.

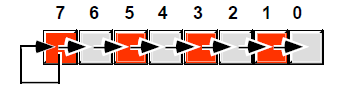


Figure 1.10: Arithmetic Shift Right Operation

This generally produces the result you expect. For example, if you perform the arithmetic shift right operation on -2 (0FEh) you get -1 (0FFh). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer which is less than or equal to the actual result. Based on experiences with high level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates towards zero. But this simply isn’t the case. For example, if you apply the arithmetic shift right operation on -1 (0FFh), the result is -1, not zero. -1 is less than zero so the arithmetic shift right operation rounds towards minus one. This is not a ‘bug’ in the arithmetic shift right operation. This is the way integer division typically gets defined. The 80x86 integer division instruction also produces this result.