1.1 Machine language

Every type of CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own unique numeric code called its operation code or opcode for short. The 80x86 processor's instructions vary in size. The opcode is always at the beginning of the instruction. Many instructions also include data (e.g., constants or addresses) used by the instruction. Machine language is very difficult to program in directly. Deciphering the meanings of the numerical-coded instructions is tedious for humans. For example, the instruction that says to add the EAX and EBX registers together and store the result back into EAX is encoded by the following hex codes:

03 C3

This is hardly obvious. Fortunately, a program called an assembler can do this tedious work for the programmer.

1.2 Assembly language

An assembly language program is stored as text (just as a higher level language program). Each assembly instruction represents exactly one machine instruction. For example, the addition instruction described above would be represented in assembly language as:

add eax, ebx

Here the meaning of the instruction is much clearer than in machine code. The word add is a mnemonic for the addition instruction. The general form of an assembly instruction is:

mnemonic operand(s)

An assembler is a program that reads a text file with assembly instructions and converts the assembly into machine code. Compilers are programs that do similar conversions for high-level programming languages. An assembler is much simpler than a compiler. Every assembly language statement directly represents a single machine instruction. High-level language statements are much more complex and may require many machine instructions. Another important difference between assembly and high-level languages is that since every different type of CPU has its own machine language, it also has its own assembly language. Porting assembly programs between different computer architectures is much more difficult than in a high-level language.

1.3 Data Representation

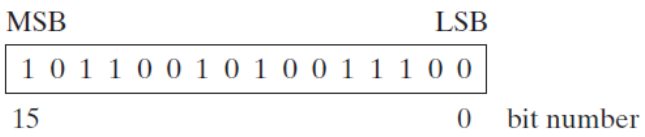
Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers. Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used. You must develop a certain fluency with number formats, so you can quickly translate numbers from one format to another. Each numbering format, or system, has a *base*, or maximum number of symbols that can be assigned to a single digit. Table (1) shows the possible digits for the numbering systems used most commonly in hardware and software manuals. In the last row of the table, hexadecimal numbers use the digits 0 through 9 and continue with the letters A through F to represent decimal values 10 through 15. It is quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.

Table (1) Binary, Octal, Decimal, and Hexadecimal Digits

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

- Binary Numbers

A computer stores instructions and data in memory as collections of electronic charges. Representing these entities with numbers requires a system geared to the concepts of *on* and *off* or *true* and *false*. *Binary numbers* are base 2 numbers, in which each binary digit (called a *bit*) is either 0 or 1. ***Bits*** are numbered sequentially starting at zero on the right side and increasing toward the left. The bit on the left is called the *most significant bit* (MSB), and the bit on the right is the *least significant bit* (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:



- Hexadecimal Numbers

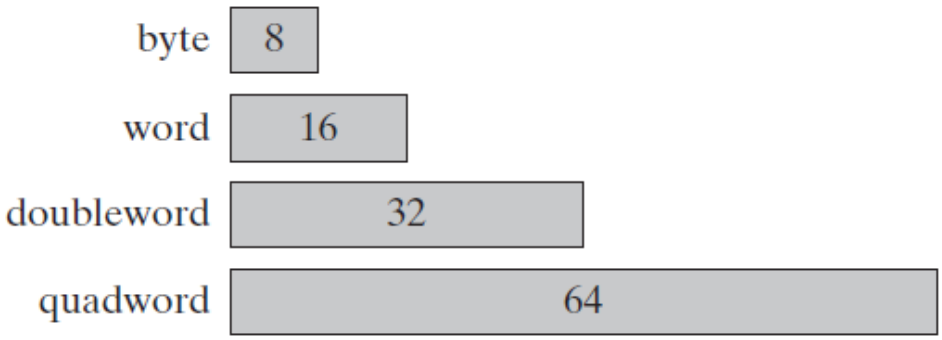
Large binary numbers are cumbersome to read, so hexadecimal digits offer a convenient way to represent binary data. Each digit in a hexadecimal integer represents four binary bits, and two hexadecimal digits together represent a byte. A single hexadecimal digit represents decimal 0 to 15, so letters A to F represent decimal values in the range 10 through 15. Table (2) shows how each sequence of four binary bits translates into a decimal or hexadecimal value.

Table (2) Binary, Decimal, and Hexadecimal Equivalents

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

- Integer Storage Sizes

The basic storage unit for all data in an x86 computer is a *byte*, containing 8 bits. Other storage sizes are *word* (2 bytes), *doubleword* (4 bytes), and *quadword* (8 bytes). In figure (1), the number of bits is shown for each size:



1.4 How Programs Run?

When you tell the computer's operating system (OS) to load and run a program, the following things happen (in sequence):

• The user issues a command to run a certain program. This might be done by typing the program's filename at a command prompt (as in MS-DOS or Linux), or by clicking on an icon or shortcut that identifies the program (as in MS-Windows or Mac OS).

• The OS searches for the program's filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called *paths)* for the filename. If the as fails to find the program filename , it issues an error message.

• If the program's filename is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive. (This process might involve several steps, but they are transparent to the user.)

• The OS determines the next available location in memory, and loads the program file into memory. It allocates a certain block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table)* . Additionally, the OS may adjust the values of pointers within the program so they contain the correct addresses of program data .

• The OS executes a branching instruction that causes the CPU to begin execution of the program's first machine instruction. As soon as the program begins running, it is called a ***process*** . The OS gives the process an identification number *(process ID)* that makes it possible to keep track of the process while it is running .

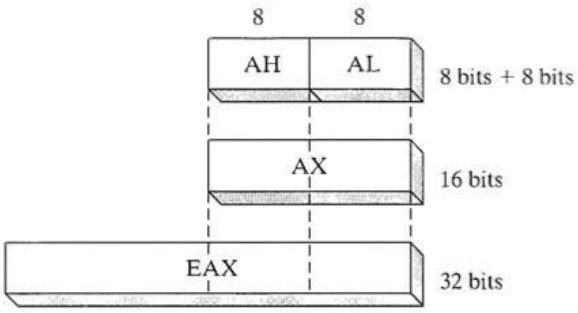
• The process runs by itself. It is the OS's job to track the execution of the process and to respond to its requests for system resources. Examples of resources are memory, disk files, and input-output devices .

• When the process ends , its handle is removed and the memory it used is released so it can be used by other programs.

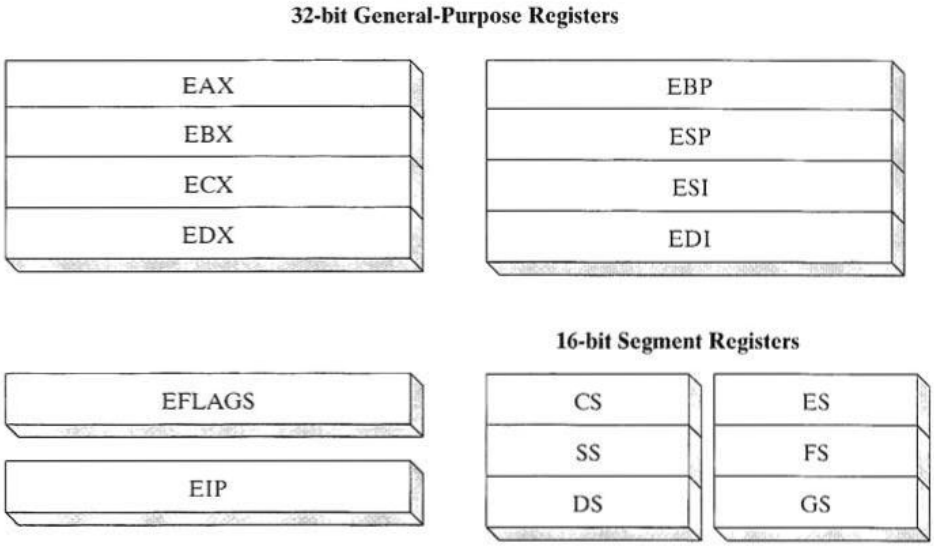
1.5 Basic Program Execution Registers

Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, registers are used inside the loop rather than variables. Figure (2) shows the *basic program execution registers* (as Intel calls them) . There are eight general-purpose registers, six segment registers, a register that holds processor status flags (EFLAGS). and an instruction pointer (EIP).

General-Purpose Registers*:* The *general-purpose registers* are primarily used for arithmetic and data movement. As shown in the following figure, each register can be addressed as either a single 32-bit value or a 16-bit value:



Some l6-bit registers can be addressed as two separate 8-bit values. For example, the EAX register is 32 bits. Its lower 16 bits are also named AX. The upper 8 bits of AX are named AH, and the lower 8 bits are named AL.



IA-32 Basic Program Execution Registers

This overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers have separate names for their lower 16 bits, but cannot be divided further. The l6-bit registers shown here are usually used only when writing programs that run in Real-address mode:

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Specialized Uses: Some general-purpose registers have specialized uses:

• EAX is automatically used by multiplication and division instructions . It is often called the *extended accumulator* register.

• The CPU automatically uses ECX as a loop counter.

• ESP addresses data on the stack (a system memory structure). It should never be used for ordinary arithmetic or data transfer. It is often called the *extended stack pointer* register.

• ESI and EDI are used by high-speed memory transfer instructions. They are sometimes called the *extended source index* and *extended destination index* registers .

• EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

***Segment Registers**:* The *segment registers* are used as base locations for preassigned memory areas called *segments*. Some segments hold program instructions (code), others hold variables (data), and another segment called the *stack segment* holds local function variables and function parameters.

***Instruction Pointer**:* The EIP, or *instruction pointer* register contains the address of the next instruction to be executed. Certain machine instructions manipulate this address, causing the program to branch to a new location.

EFLAGS Register: The EFLAGS (or just *Flags)* register consists of individual binary bits that either control the operation of the CPU or reflect the outcome of some CPU operation. There are machine instructions that can test and manipulate the processor flags.

A flag is set when it equals 1: it is clear (or reset) when it equals 0