Assembly Language – Chapter No. 2
x86 Processor Architecture

X86 PROCESSORS

Sheikh Muhammad Aamir

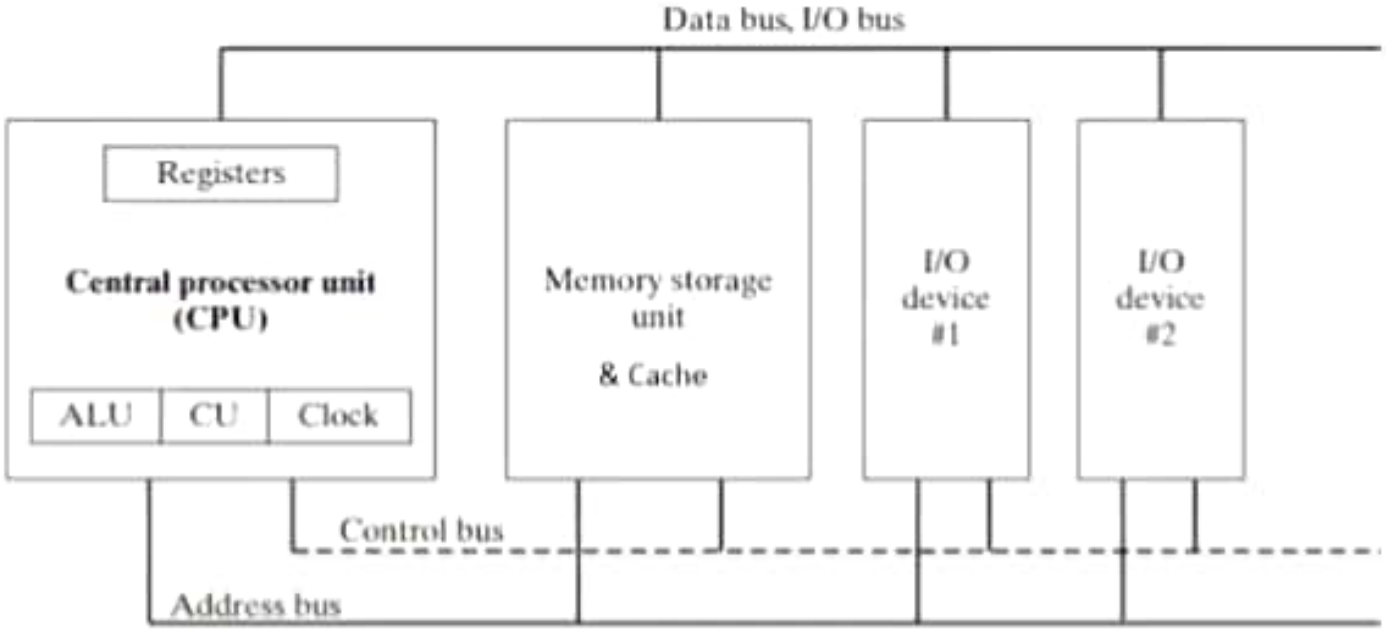
Department of Computer Science

GC University, Faisalabad

Topics to be covered

- Basic Microcomputer Design
- Memory Hierarchy
- Instruction Execution Cycle
- Reading from Memory
- Loading and Executing a program
- CPU Registers

Basic Microcomputer Design



2.1.1 Basic Microcomputer Design

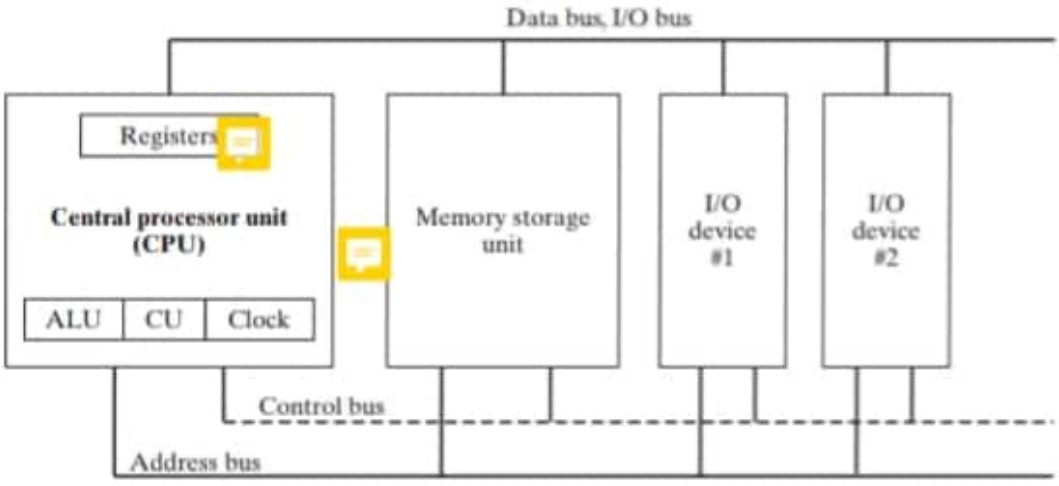
Figure 2-1 shows the basic design of a hypothetical microcomputer. The *central processor unit* (CPU), where calculations and logical operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The *clock* synchronizes the internal operations of the CPU with other system components.
- The *control unit* (CU) coordinates the sequencing of steps involved in executing machine instructions.
- The *arithmetic logic unit* (ALU) performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer's motherboard. Most pins connect to the data bus, the control bus, and the address bus. The *memory storage unit* is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory. All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute. Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together.

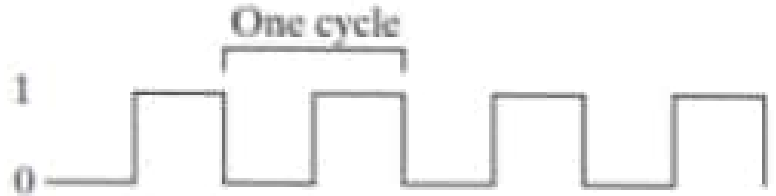
A *bus* is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address. The *data bus* transfers instructions and data between the CPU and memory. The I/O bus transfers data

FIGURE 2–1 Block diagram of a microcomputer.



between the CPU and the system input/output devices. The *control bus* uses binary signals to synchronize actions of all devices attached to the system bus. The *address bus* holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

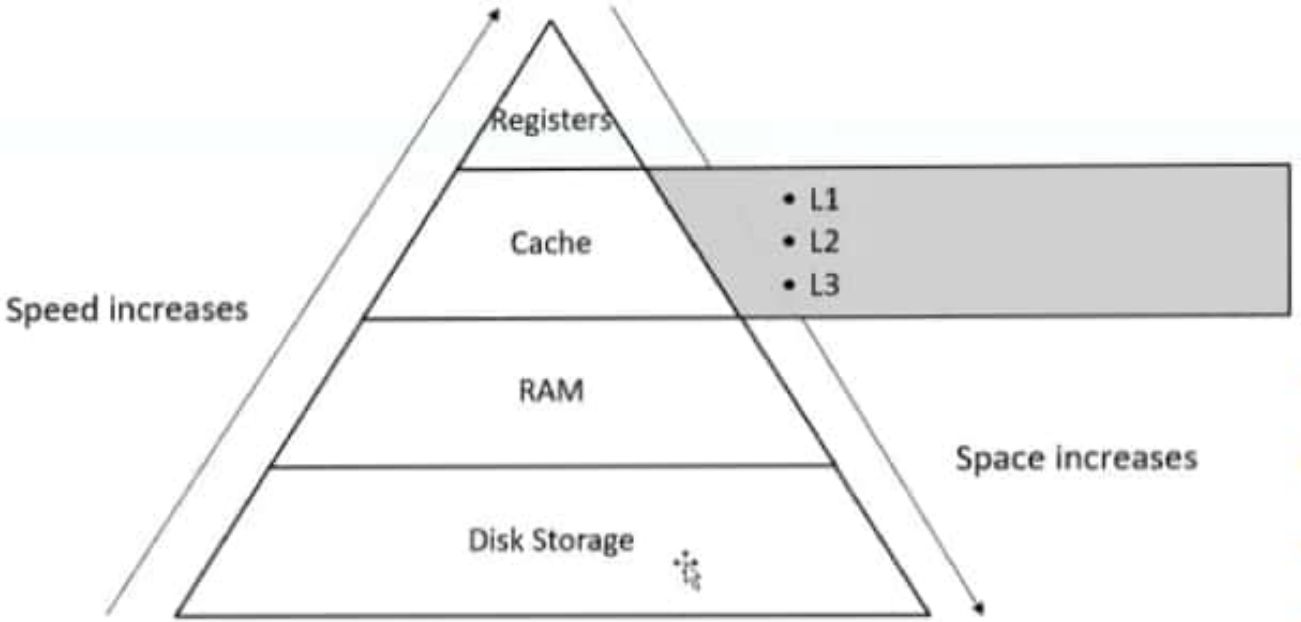
Clock Each operation involving the CPU and the system bus is synchronized by an internal clock pulsing at a constant rate. The basic unit of time for machine instructions is a *machine cycle* (or *clock cycle*). The length of a clock cycle is the time required for one complete clock pulse. In the following figure, a clock cycle is depicted as the time between one falling edge and the next:



The duration of a clock cycle is calculated as the reciprocal of the clock's speed, which in turn is measured in oscillations per second. A clock that oscillates 1 billion times per second (1 GHz), for example, produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

A machine instruction requires at least one clock cycle to execute, and a few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example). Instructions requiring memory access often have empty clock cycles called *wait states* because of the differences in the speeds of the CPU, the system bus, and memory circuits.

Memory Hierarchy



- Register capacity in BYTES
- Cache capacity in MB
- RAM capacity in GB
- HDD capacity in TB

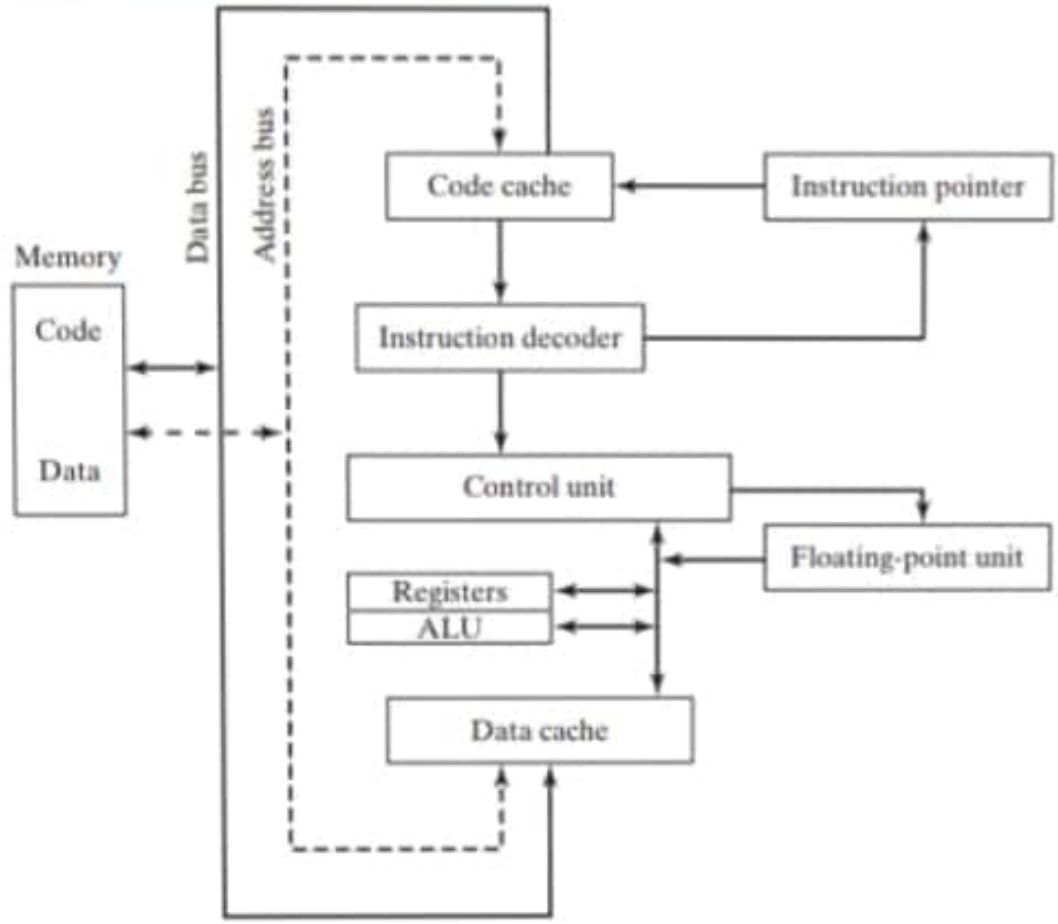
2.1.2 Instruction Execution Cycle

A single machine instruction does not just magically execute all at once. The CPU has to go through a predefined sequence of steps to execute a machine instruction, called the *instruction execution cycle*. Let's assume that the instruction pointer register holds the address of the instruction we want to execute. Here are the steps to execute it:

1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction queue*. Right after doing this, it increments the instruction pointer.
2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern. This bit pattern might reveal that the instruction has operands (input values).
3. If operands are involved, the CPU **fetches the operands** from registers and memory. Sometimes, this involves address calculations.
4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step. It also updates a few status flags, such as Zero, Carry, and Overflow.
5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its execution in the operand.

We usually simplify this complicated-sounding process to three basic steps: **Fetch**, **Decode**, and **Execute**. An *operand* is a value that is either an input or an output to an operation. For example, the expression Z = X + Y has two input operands (X and Y) and a single output operand (Z).

Figure 2–2 Simplified CPU block diagram.



digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit. Although the control bus is not shown in this figure, it carries signals that use the system clock to coordinate the transfer of data between the different CPU components.

2.1.3 Reading from Memory

As a rule, computers read memory much more slowly than they access internal registers. This is because reading a single value from memory involves four separate steps:

1. Place the address of the value you want to read on the address bus.
2. Assert (change the value of) the processor's RD *(read)* pin.
3. Wait one clock cycle for the memory chips to respond.
4. Copy the data from the data bus into the destination operand.

Each of these steps generally requires a single *clock cycle*, a measurement of time based on a clock that ticks inside the processor at a regular rate. Computer CPUs are often described in terms of their clock speeds. A speed of *1.2 GHz*, for example, means the clock ticks, or oscillates, 1.2 billion times per second. So, 4 clock cycles go by fairly fast, considering each one lasts for only 1/1,200,000,000th of a second. Still, that's much slower than the CPU registers, which are usually accessed in only one clock cycle.

Fortunately, CPU designers figured out a long time ago that computer memory creates a speed bottleneck because most programs have to access variables. They came up with a clever way to reduce the amount of time spent reading and writing memory—they store the most recently used instructions and data in high-speed memory called *cache*. The idea is that a program is more likely to want to access the same memory and instructions repeatedly, so cache keeps these values where they can be accessed quickly. Also, when the CPU begins to execute a program, it can look ahead and load the next thousand instructions (for example) into cache, on the assumption that these instructions will be needed fairly soon. If there happens to be a loop in that block of code, the same instructions will be in cache. When the processor is able to find its data in cache memory, we call that a *cache hit*. On the other hand, if the CPU tries to find something in cache and it's not there, we call that a *cache miss*.

Cache memory for the x86 family comes in two types. *Level-1 cache* (or *primary cache*) is stored right on the CPU. *Level-2 cache* (or *secondary cache*) is a little bit slower, and attached to the CPU by a high-speed data bus. The two types of cache work together in an optimal way.

There's a reason why cache memory is faster than conventional RAM—it's because cache memory is constructed from a special type of memory chip called *static RAM*. It's expensive, but it does not have to be constantly refreshed in order to keep its contents. On the other hand, conventional memory, known as *dynamic RAM*, must be refreshed constantly. It's much slower, but cheaper.

2.1.4 Loading and Executing a Program

Before a program can run, it must be loaded into memory by a utility known as a *program loader*. After loading, the operating system must point the CPU to the program's *entry point*, which is the address at which the program is to begin execution. The following steps break this process down in more detail:

- The operating system (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 OS fails to find the program filename, it issues an error message.
- If the program file 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.
- The OS determines the next available location in memory and loads the program file into memory. It allocates a 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 addresses of program data.
- The OS begins execution of the program's first machine instruction (its entry point). As soon as the program begins running, it is called a *process*. The OS assigns the process an identification number (*process ID*), which is used to keep track of it while running.
- The *process* runs by itself. It is the OS's job to track the execution of the process and to respond to requests for system resources. Examples of resources are memory, disk files, and input-output devices.
- When the process ends, it is removed from memory.

> ***Tip:*** If you're using any version of Microsoft Windows, press *Ctrl-Alt-Delete* and select the *Task Manager* item. The Task Manager window lets you view lists of Applications and Processes. Applications are the names of complete programs currently running, such as Windows Explorer or Microsoft Visual C++. When you click on the *Processes* tab, you see a long list of process names. Each of those processes is a small program running independently of all the others. You can continuously track the amount of CPU time and memory used by each process. In some cases, you can shut down a process by selecting its name and pressing the *Delete* key.

CPU Registers

- Named storage location inside the CPU or CPU local memory
- A register is faster than Cache, RAM and than other storages
- A register may hold :
  - An instruction
  - Storage address
  - Or any kind of data

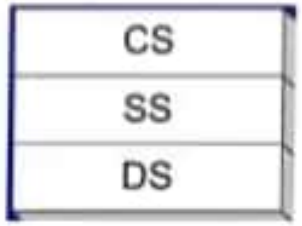
CPU Registers

- Named storage location inside the CPU or CPU local memory
- A register is faster than Cache, RAM and than other storages
- A register may hold :
  - An instruction
  - Storage address
  - Or any kind of data

Register Types

- 32-bits General Purpose Registers

- 16-bits Segment Registers



Other Registers

Register Types . . .

- 32-bits General Purpose Registers

Register	Name	Register	Name
EAX	Extended Accumulator	EBP	Extended Base Pointer
EBX	Extended Base Register	ESP	Extended Stack Pointer
ECX	Extended Counter Register	ESI	Extended Source Index
EDX	Extended Data Register	EDI	Extended Destination Index

- 16-bits Segment Registers

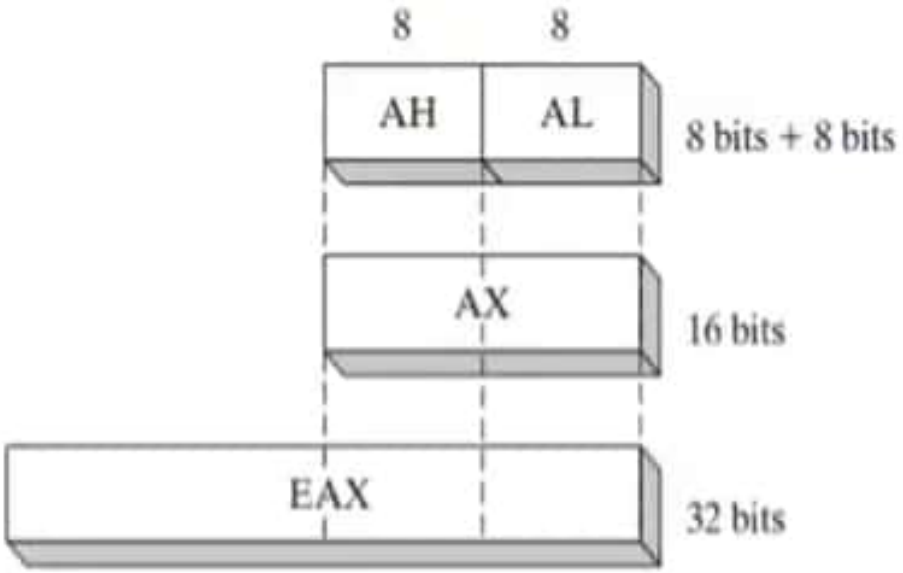
Register	Name
CS	Code Segment
DS	Data Segment
SS	Stack Segment

Other Registers

Register	Name
EFLAGS	Extended Flag Register
EIP	Extended Instruction Pointer

Register Types . . .

- 32-bits General Purpose 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

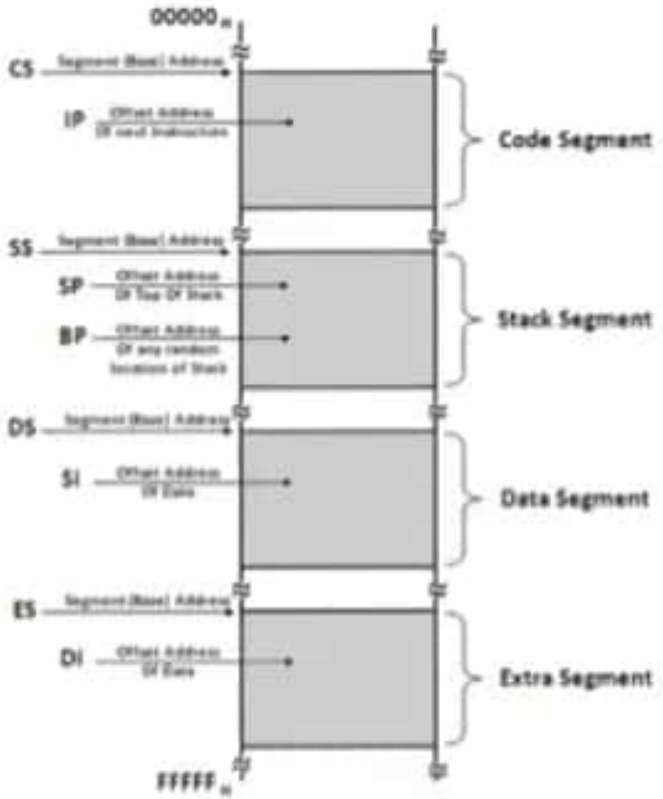
Register Types . . .

- **General Purpose Registers** (32-bits or 16-bits) - remaining

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

Register Types . . .

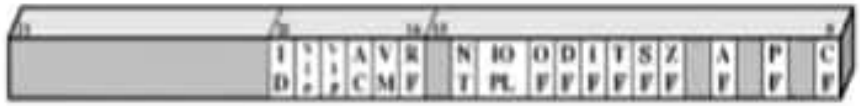
- **Segment Registers** (16-bits)
- Segments are specific areas defined in a program for containing data, code and stack.
  - DS – Data Segment
  - CS – Code Segment
  - SS – Stack Segment
- **EIP - Instruction Pointer** (32-bits)
  - The EIP, or *instruction pointer*, register contains the address of the very next instruction to be executed.



Register Types . . .

- **EFLAGS Register**
  - The EFLAGS (or just *Flags*) register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation
  - A flag is *set* when it equals 1; it is *clear* (or reset) when it equals 0
  - It contains:

Status Bits	Control Bits	Reserved Bits
• CF – Carry Flag • OF – Overflow Flag • SF – Sign Flag • ZF – Zero Flag • PF – Parity Flag • etc.	• DF – Direct Flag • IF – Interrupt Flag • etc.	



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Special Concepts

- EAX is used automatically in Multiplication and Division

For example:

```
MOV EAX, 10
MOV EBX, 5
MUL EBX          ; EAX = EAX * EBX
```

- ECX is also known as Loop Counter. It means to perform loop iteration, ECX must contains a non-negative value (more than 0)

- ESP addresses data on the stack (a system memory structure)