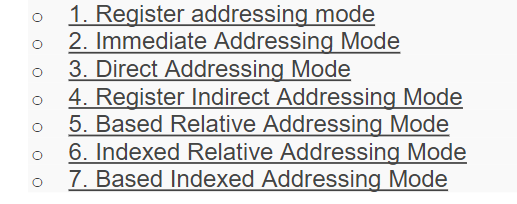
8086 Microprocessor Addressing Modes

What is Addressing Mode? Addressing modes are different ways by which CPU can access data or operands. They determine how to access a specific memory address. To load any data from and to memory/registers, MOV instruction is used.

The syntax of MOV instruction is:

MOV Destination, Source

It copies the data of 2nd operand (source) into the 1st operand (destination). To access memory, segment registers are used along with general-purpose registers.



1. **Register addressing mode**

This mode involves the use of registers. These registers hold the operands. This mode is very fast as compared to others because CPU doesn’t need to access memory. CPU can directly perform an operation through registers.

For example:

MOV AX,BL

MOV AL,BL

The above two instructions copy the data of BL register to AX and AL.

1. **Immediate Addressing Mode**

In this mode, there are two operands. One is a register and the other is a constant value. The register comes quickly after the op code.

For example:

•The instruction

MOV AX, 30H

copies hexadecimal value 30H to register AX.

•The instructions

MOV BX, 255

copies decimal value 255 to register BX.

You cannot use the immediate addressing mode to load immediate value into segment registers. To move any value into segment registers, first load that value into a general-purpose register then add this value into segment register.

1. **Direct Addressing Mode**

It loads or stores the data from memory to register and vice versa. The instruction consists of a register and an offset address.

MOV CX,[481]

The hexadecimal value of 481 is 1E1. Assume DS=2162H then the logical address will be 2162:01E1.

1. **Register Indirect Addressing Mode**

The register indirect addressing mode uses the offset address which resides in one of these three registers i.e., BX, SI, DI. **The sum of offset address and the DS value shifted by one position generates a physical address.**

For example:

MOV AL, [SI]

This instruction will calculate the physical address by shifting DS to the left by one position and adding it to the offset address residing in SI. The brackets around SI indicates that the SI contain the offset address of memory location whose data needs to be accessed. If brackets are absent, then the instruction will copy the contents of SI register to AL. Therefore, brackets are necessary

1. **Based Relative Addressing Mode**

This addressing mode uses a base register either BX or BP and a displacement value to calculate physical address.

Physical Address= Segment Register (Shifted to left by 1) + Effective address

The effective address is the sum of offset register and displacement value.

The default segments for BX and BP are DS and SS.

For example:

MOV[BX+5],DX

In this example, the effective address is BX + 5 and the physical address is DS (shifted left) + BX+5.The instruction on execution will copy the value of DX to memory location of physical address= DS (shifted left) +BX+5.

1. **Indexed Relative Addressing Mode**

This addressing mode is same as the based relative addressing mode.

The only difference is it uses DI and SI registers instead of BX and BP registers.

For example:

Given that

DS=704, SI = 2B2,

DI= 145MOV[DI]+12,AL

MOV[DI]+12,AL

This instruction on execution will copy the content of AL at memory address 7197 (7040 + 145 + 12)

MOVBX,[SI]+10 This instruction will load the contents from memory address 7302 (7040 +2B2 +10) to register BX.

1. **Based Indexed Addressing Mode**

The based indexed addressing mode is actually a combination of based relative addressing mode and indexed relative addressing mode. It uses one base register (BX, BP) and one index register (SI, DI).

For example:

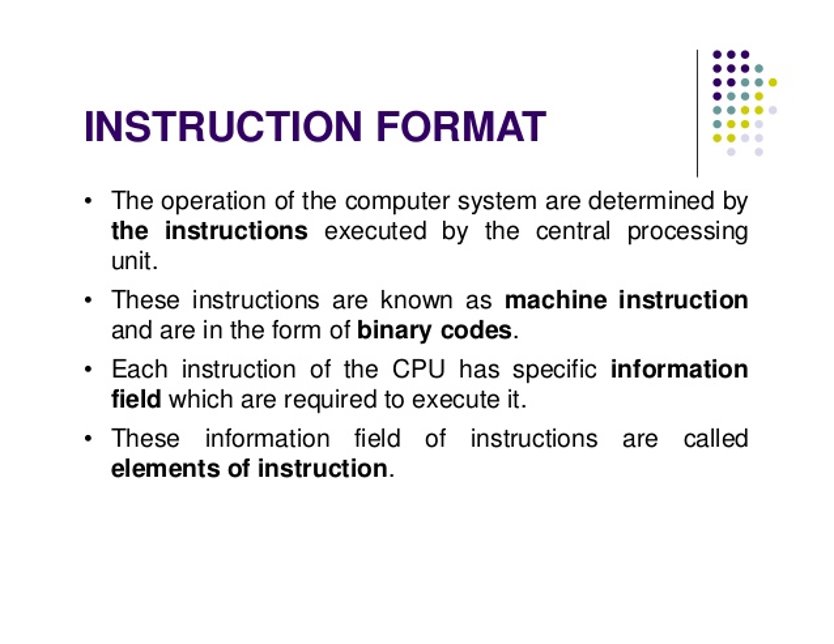
MOVAX,[BX+SI+20]

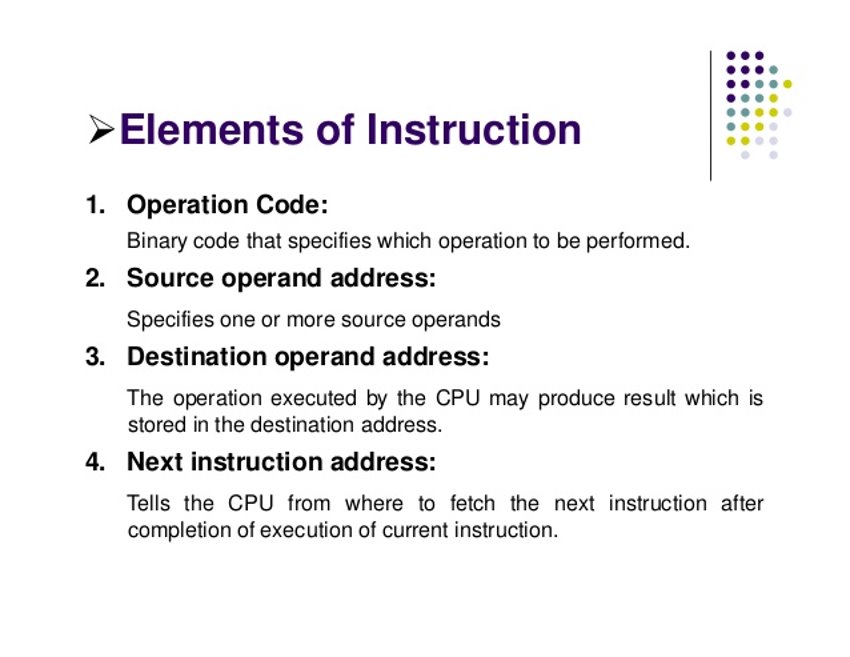
The above instruction can also be written as:

MOVAX,[SI+BX+20] Or MOVAX,[SI][BX]+20

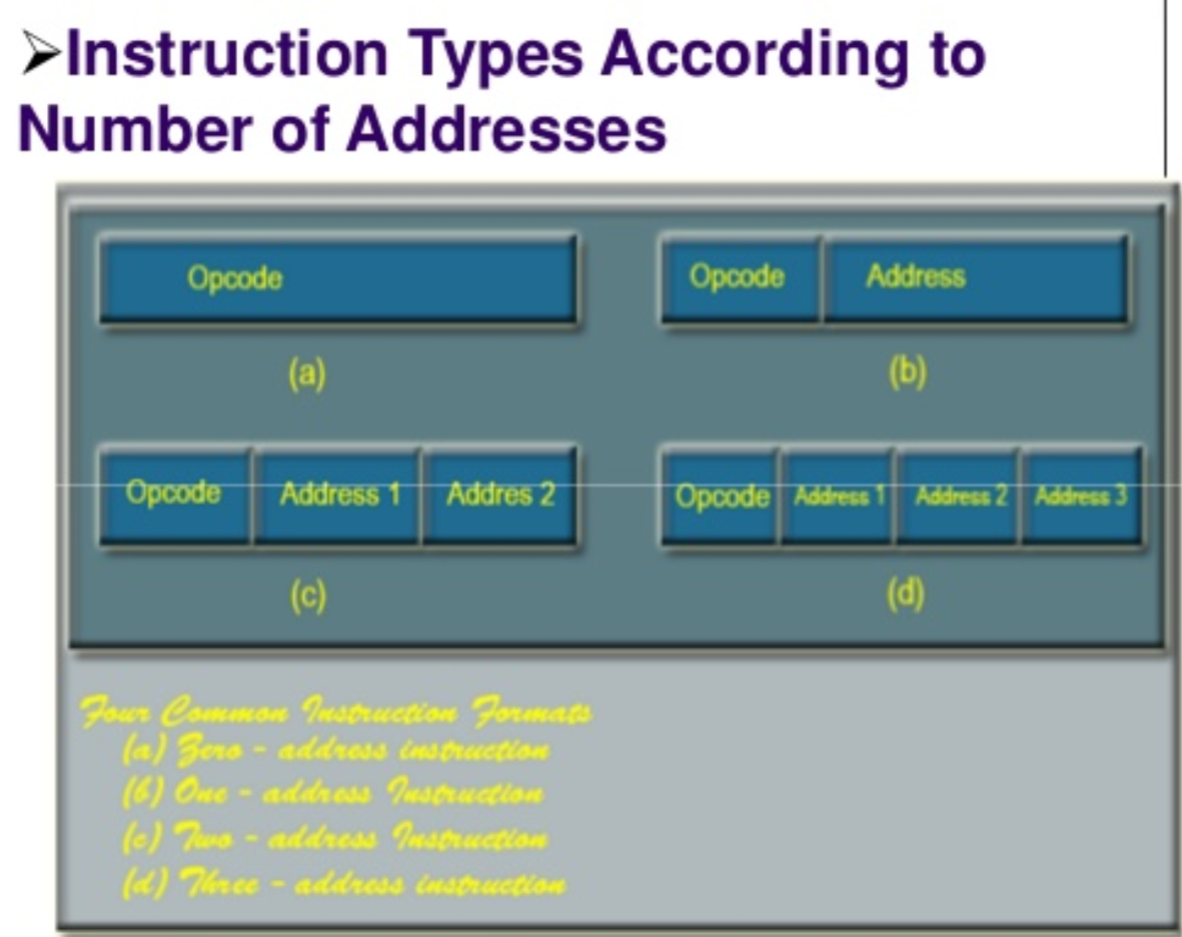
In this case, the physical address will be DS (Shifted left) + SI + BX + 20. Now, if we replace BX with BP then the physical address is equal to SS (Shifted left) + SI + BX + 20.

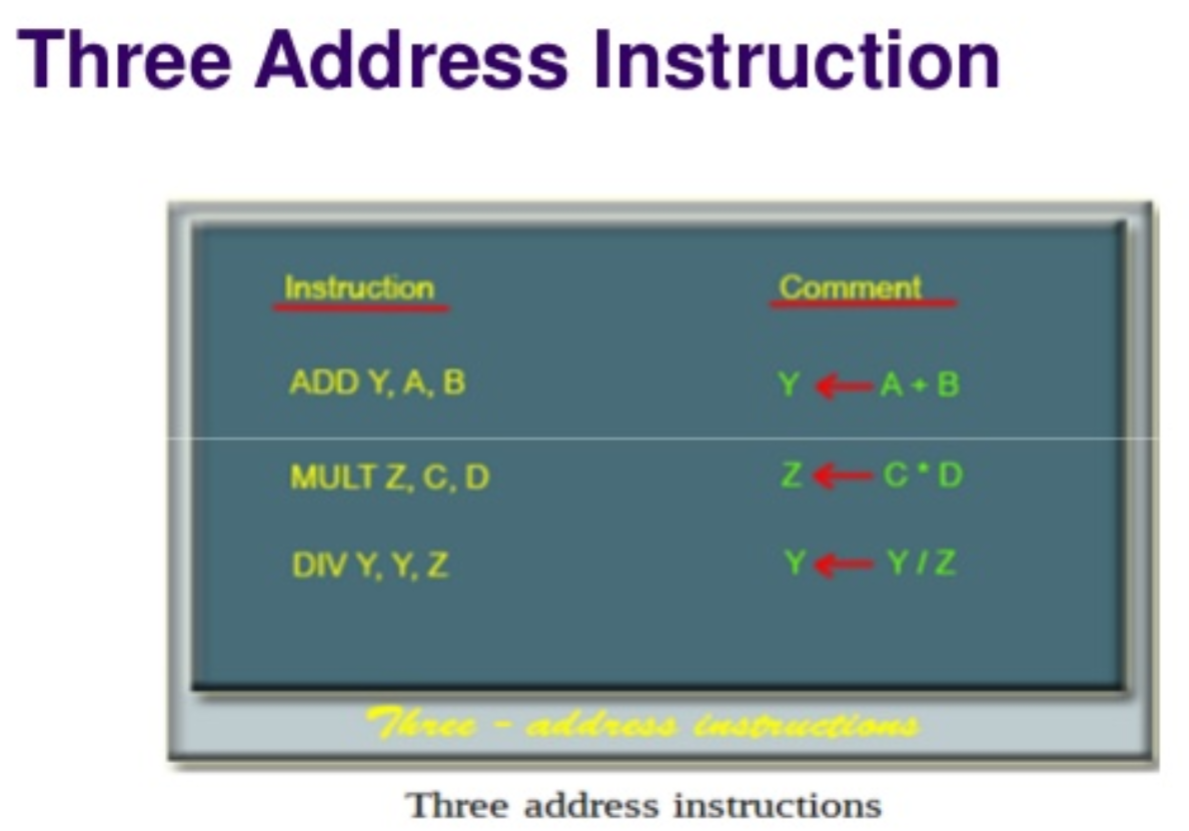
**INSTRUCTIONS IN 8086**

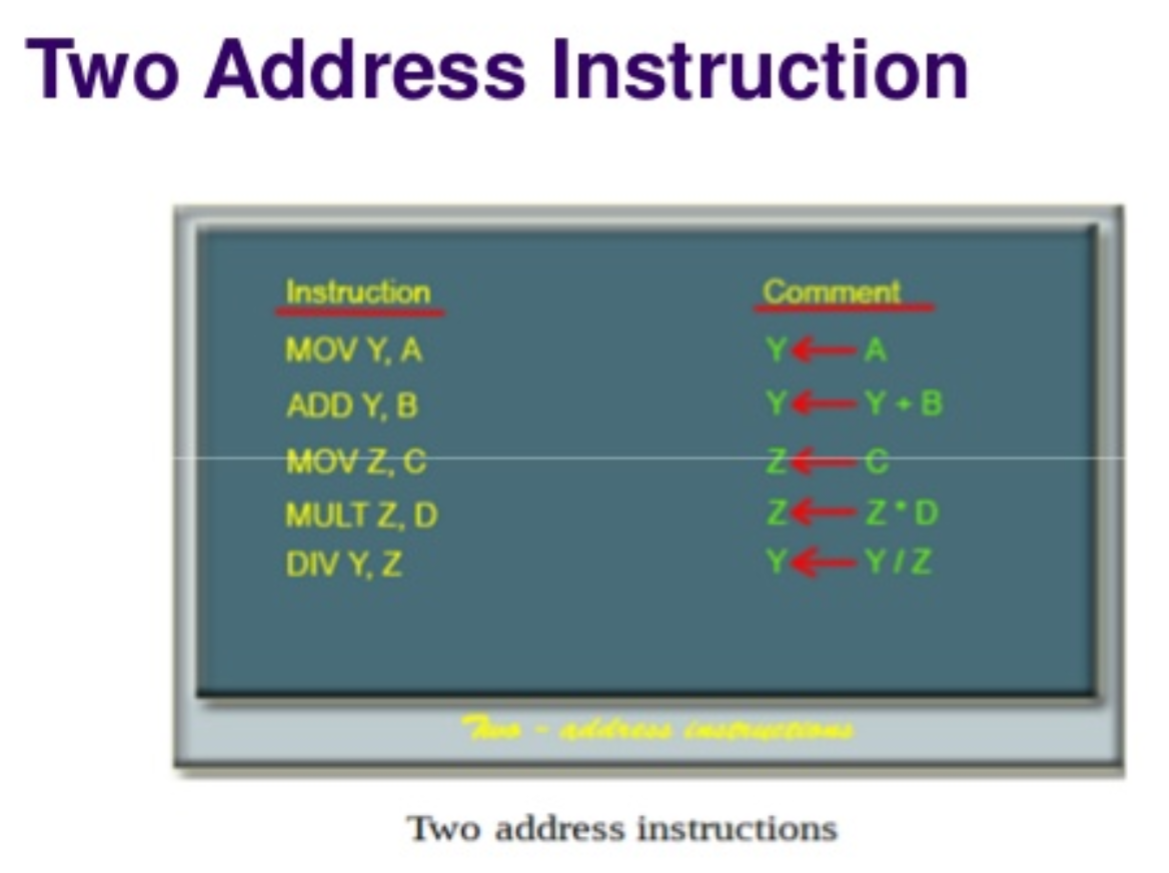
****

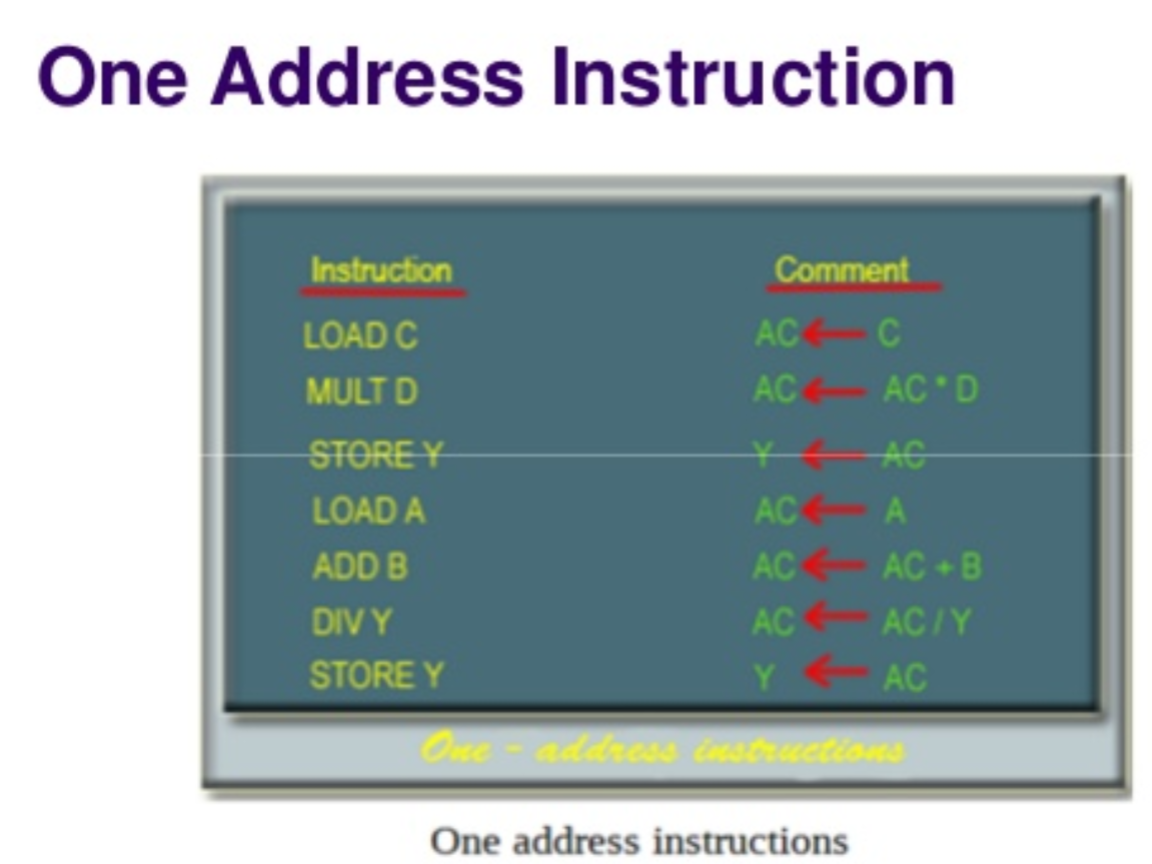
****

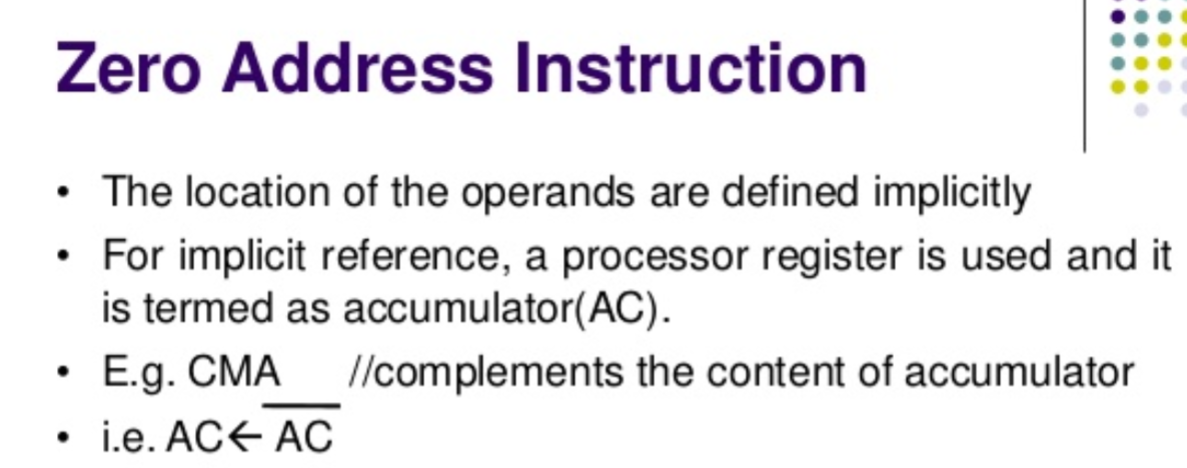




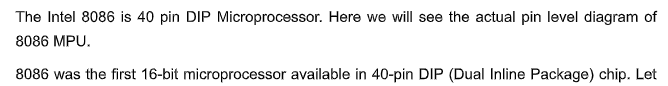


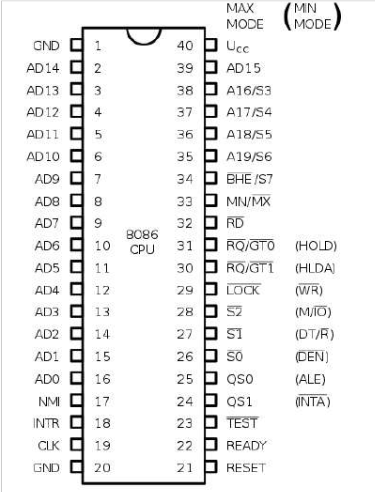


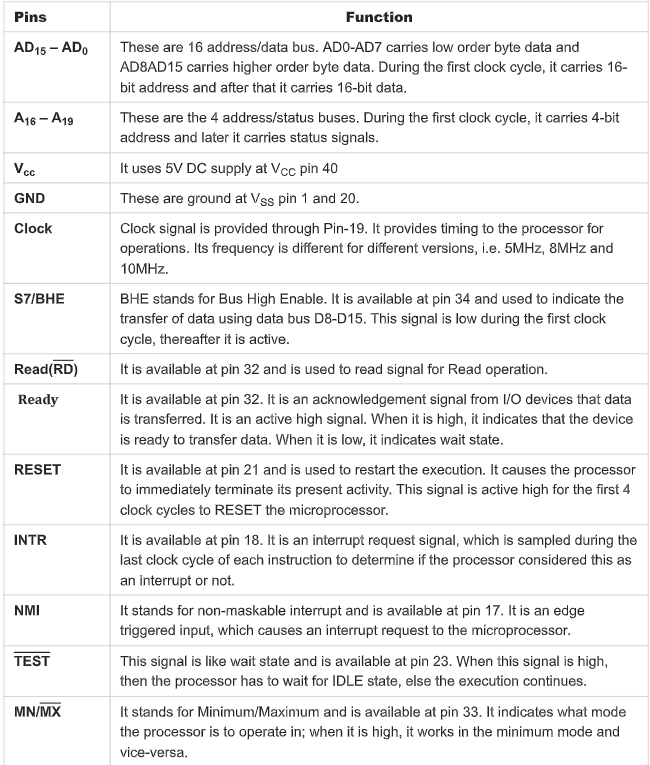


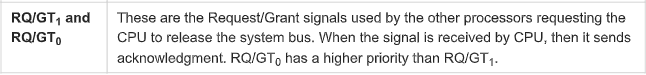
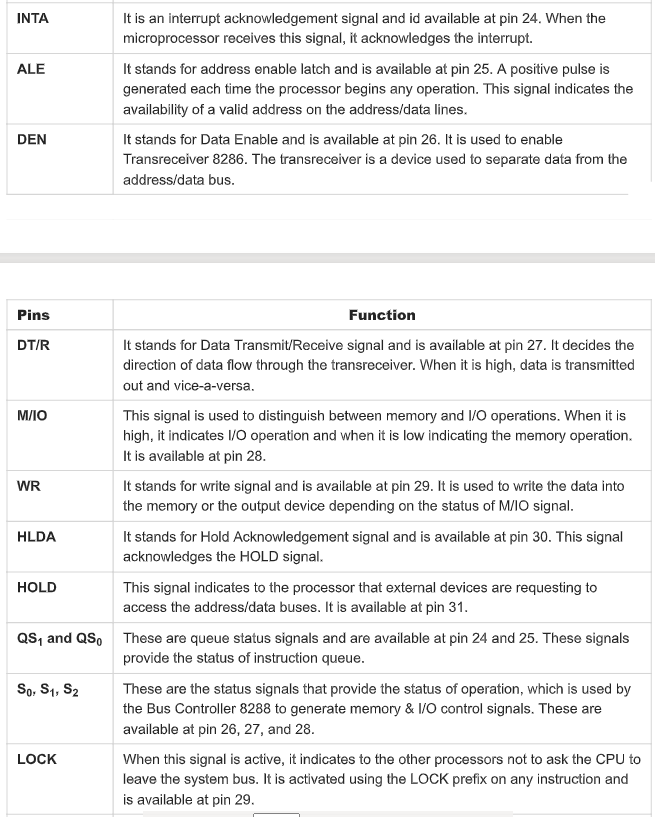


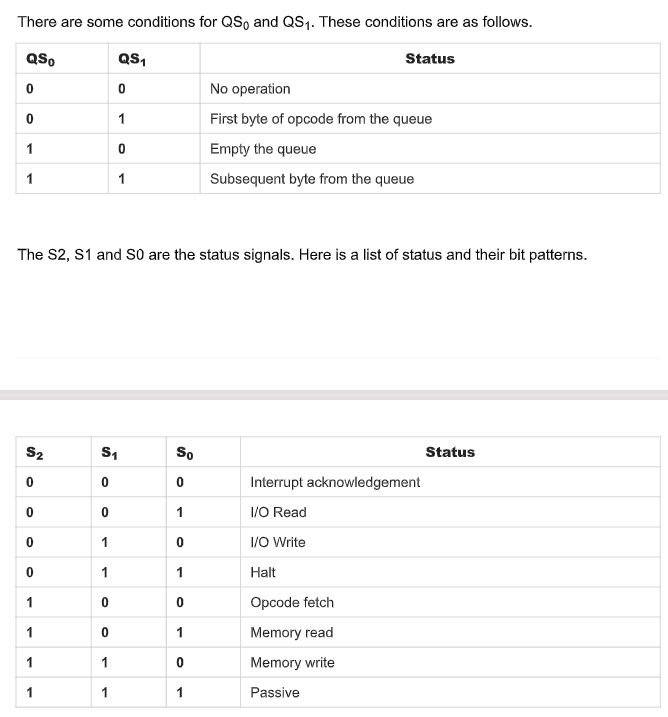
**PIN DIAGRAM 8086**

****

****

****

****

****

**Interrupts in 8086**

**8086 Interrupt Types:**

The8086 Interrupt Types are: Dedicated and Software interrupts

**Dedicated Interrupts:**

**Type 0 :** **Divide by Zero Interrupt**

When the quotient from either a DIV or IDIV instruction is too large to fit in the result register; 8086 Interrupt Types will automatically execute type 0 interrupt.

**Type 1 : Single Step Interrupt**

The type 1 interrupt is the single step trap. In the single step mode, system will **execute one instruction and wait for further direction from user**. **Then user can examine the contents of registers and memory locations and if they are correct**, user can tell the system to execute the next instruction. This feature is useful for **debugging** assembly language programs.

An 8086 Interrupt Types system is used in the single step mode by setting the trap flag. If the trap flag is set, the 8086 will automatically execute a type 1 interrupt after execution of each instruction. But the 8086 has no such instruction to directly set or reset the trap flag. These operations can be performed by taking the flag register contents into memory, changing the memory contents so to set or reset trap flag and save the memory contents into flag register.

**Type 2 : Non Maskable Interrupt**

As the name suggests, this interrupt cannot be disabled by any software instruction. This interrupt is **activated by low to high transition on 8086 NMI input pin**. In response, 8086 will do a type 2 interrupt.

**Type 3 : Breakpoint**

The type 3 interrupt is used to implement **break point** function in the system.

The type 3 interrupt is produced by execution of the INT 3 instruction.

Break point function is often used as a debugging aid in cases where single stepping provides more detail than wanted.

When you insert a breakpoint, the system executes the instructions upto the breakpoint, and then goes to the breakpoint procedure.

In the break point procedure you can write a program to display register contents, memory contents and other information that is required to debug your program.

You can insert as many breakpoints as you want in your program.

**Type 4 : Overflow Interrupt**

The type 4 interrupt is used to check overflow condition after any signed arithmetic operation in the system. The 8086 Interrupt Types overflow flag, OF, will be represented in the destination register or memory location.

For example, if you add the 8-bit signed number 0111 1000 (+ 120 decimal) and the 8 bit signed number 0110 1010 (t 106 decimal), result is 1110 0010 (-98 decimal).

In signed numbers, MSB (Most significant Bit) is reserved for sign and other bits represent magnitude of the number. In the previous example, after addition of two 8-bit signed numbers result is negative, since it is too large to fit in 7 bits. To detect this condition in the program, you can put interrupt on overflow instruction, INTO, immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will simply function as an NOP (no operation). However, if the overflow flag is set, indicating an overflow error, the 8086 will execute a type 4 interrupt after executing the INTO instruction.

**Another way to detect and respond to an overflow error in a program is to put the jump if overflow instruction (JO) immediately after the arithmetic instruction**. If the overflow flag is set as a result of arithmetic operation, execution will jump to the address specified in the JO instruction. At this address, you can put an error routine which responds in the way you want to the overflow.

**Software Interrupts:**

**Type 0 –255** :

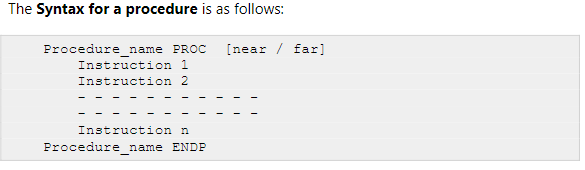
The 8086 INT instruction can be used to cause the 8086 to do one of the 256 possible interrupt types. The interrupt type is specified by the number as a part of the instruction. You can use an INT2 instruction to send execution to an NMI interrupt service routine. This allows you to test the NMI routine without needing to apply an external signal to the NMI input of the 8086 Interrupt Types.

**Macros and Procedures in 8086**

**Procedure**:

In a program, we very frequently face situations where there is a **need to perform the same set of task again and again**. So, for that instead of writing the same sequence of instructions, again and again, they are written separately in a subprogram. This subprogram is called a procedure. With the help of procedures, we can very well implement the concept of modular programming in our code. Also, whenever we need to execute the instructions mentioned in the procedure, we can **simply make a CALL to it**. Therefore, with the help of procedures, theduplicity in the instructions can be avoided.

The Syntax for a procedure is as follows:



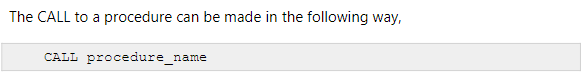
•Here, the PROC is a keyword to define that the set of instructions enclosed by the given name is a procedure.

•The ENDP keyword defines that the body of the procedure has been ended.

•All the instructions lying between these two keywords are the instructions that belong to the procedure and will be executed whenever a CALL to the procedure is made.

•The keyword near or far defines the range of code within which the procedure is defined. If it is defined in the same segment as the rest code, then near is used.

•If it is defined in some other segment, then the keyword far is used for it.



At the end of the procedure, the **RET** instruction is used. This instruction will cause the execution to be transferred to the program from which the call to the procedure was made.

**Macro**:

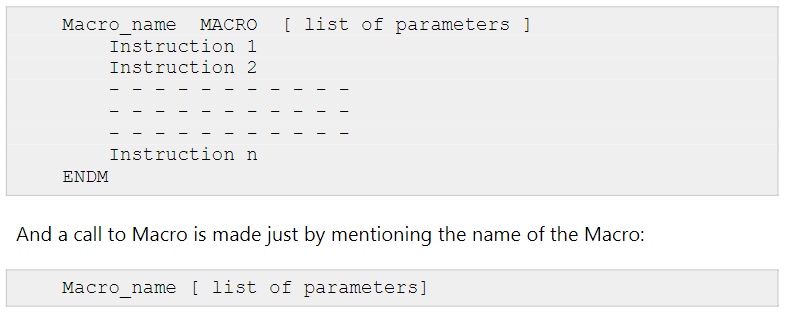
A Macro is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures).

The Macro is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the Macro is made.

A Macro can be defined in a program using the following assembler directives: **MACRO**(used after the name of Macro before starting the body of the Macro) and **ENDM**(at the end of the Macro).

All the instructions that belong to the Macro lie within these two assembler directives.

The following is the syntax for defining a Macro in the 8086 Microprocessor:



It is optional to pass the parameters in the Macro. If you want to pass them to your macros, you can simply mention them all in the very first statement of the Macro just after the directive: MACRO.

The advantage of using Macro is that it avoids the overhead time involved in calling and returning (as in the procedures). Therefore, **the execution of Macros is faster as compared to procedures.** Another advantage is that there is no need for accessing stack or providing any separate memory to it for storing and returning the address locations while shifting the processor controls in the program.

But it should be noted that every time you call a macro, the assembler of the microprocessor places the entire set of Macro instructions in the mainline program from where the call to Macro is being made. This is known as Macro expansion. Due to this, the program code (which uses Macros) takes more memory space than the code which uses procedures for implementing the same task using the same set of instructions. Hence, it is better to use Macros where we have small instruction sets containing less number of instructions to execute.

**Addressing modes of 8051**

In 8051 There are six types of addressing modes.

•Immediate Addressing Mode

•Register Addressing Mode

•Direct Addressing Mode

•Register Indirect Addressing Mode

•Indexed Addressing Mode

•Implied Addressing Mode

**1.Immediate addressing mode**

In this Immediate Addressing Mode, the data is provided in the instruction itself. The data is provided immediately after the opcode.

These are some examples of Immediate Addressing Mode.

MOVA,#0AFH;

MOVDPTR,#FE00H;

In these instructions, the # symbol is used for immediate data. In the last instruction, there is DPTR. The DPTR stands for Data Pointer. Using this, it points the external data memory location. In the first instruction, the immediate data is AFH, but one 0 is added at the beginning. So when the data is starting with A to F, the data should be preceded by 0.

**2. Direct Addressing** **Mode**

In the Direct Addressing Mode, the source or destination address is specified by using 8-bit data in the instruction. Only the internal data memory can be used in this mode.

Here some of the examples of direct Addressing Mode.

MOV80H,R6;

MOVR2,45H;

MOVR0,05H;

The first instruction will send the content of registerR6 to port P0 (Address of Port 0 is 80H).

The second one is for getting content from 45H to R2.

The third one is used to get data from Register R5 (When register bank RB0 is selected) to registerR5.

**4.Register indirect addressing Mode**

In this mode, the source or destination address is given in the register. By using register indirect addressing mode, the internal or external addresses can be accessed. The R0 and R1 are used for 8-bit addresses, and DPTR is used for 16-bit addresses, no other registers can be used for addressing purposes.

Let us see some examples of this mode.

MOV0E5H,@R0;

In the instructions, the @ symbol is used for register indirect addressing. In the instruction, it is showing that the R0 register is used. If the content of R0 is 40H, then that instruction will take the data which is located at location 40H of the internal RAM.

**5.Indexed addressing mode**

In the indexed addressing mode, the source memory can only be accessed from program memory. The destination operand is always the register A.

These are some examples of Indexed addressing mode.

MOVA,@A+PC;

MOVA,@A+DPTR;

For the first instruction, let us consider A holds 30H. And the PC value is 1125H. The contents of program memory location 1155H (30H + 1125H) are moved to register A.

**6.Implied Addressing Mode**

In the implied addressing mode, there will be a single operand. These types of instruction can work on specific registers only. These types of instructions are also known as register specific instruction.

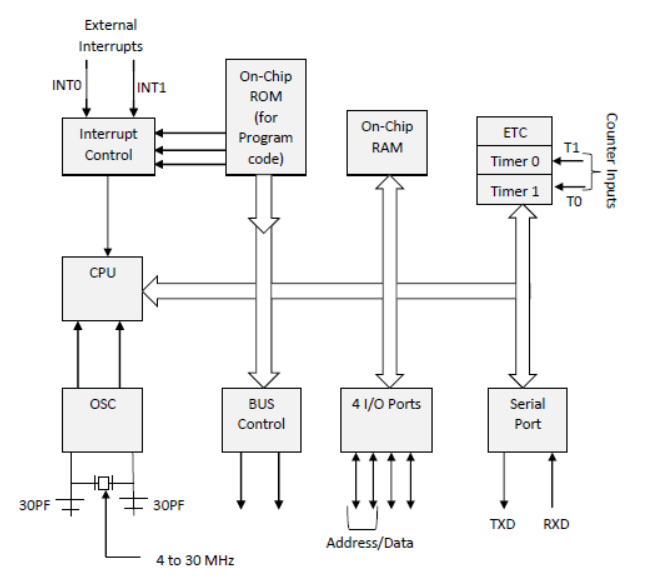
Here are some examples of Implied Addressing Mode.

RLA;

SWAPA;

These are 1-byte instruction. The first one is used to rotate the A register content to the Left. The second one is used to swap the nibbles in A.

8051 ARCHITECTURE



8051 microcontroller is designed by Intel in 1981. It is an 8-bit microcontroller. It is built with 40 pins DIP (dual inline package), 4kb of ROM storage and 128bytes of RAM storage, 2 16-bit timers. It consists of four parallel 8-bit ports, which are programmable as well as addressable as per the requirement. An on-chip crystal oscillator is integrated in the microcontroller having crystal frequency of 12 MHz. Let us now discuss the architecture of 8051 Microcontroller. In the following diagram, the system bus connects all the support devices to the CPU. The system bus consists of an 8-bit data bus, a 16-bit address bus and bus control signals. All other devices like program memory, ports, data memory, serial interface, interrupt control, timers, and the CPU are all interfaced together through the system bus.

**8051 Instruction Set**

the 8051 Microcontroller Instruction Set is optimized for 8-bit control applications.

8051 Microcontroller instruction set can have up to 28 = 256 Instructions.

There are 49 Instruction Mnemonics in the 8051 Microcontroller Instruction Set and these 49 Mnemonics are divided **into five groups:**

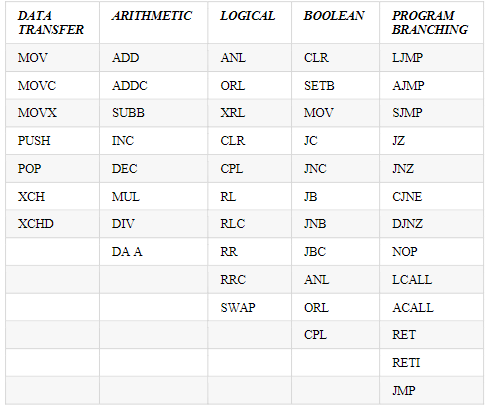
DATA TRANSFER

ARITHMETIC

LOGICAL

BOOLEAN

PROGRAM BRANCHING



**DATA TRANSFER:**

**MOV A, Rn** - Moves the Rn register to the accumulator

The instruction moves the Rn register to the accumulator.

The Rn register is not affected.

**Syntax**: MOV A,Rn

Before execution:

R3=58h

After execution:

R3 = 58h A=58h

**MOV A, #data** - Moves the immediate data to the accumulator

Instruction moves the immediate data to the accumulator.

**Syntax**: MOV A, #28

After execution: A = 28h

**MOV A, @Ri** - Moves the indirect RAM to the accumulator

Instruction moves the indirectly addressed register of RAM to the accumulator.

The register address is stored in the Ri register (R0 or R1).

The result is stored in the accumulator.

The register is not affected.

**Syntax**: MOV A,@Ri

Register Address SUM=F2h R0=F2h

Before execution: SUM=58h

After execution: A=58h SUM=58h

**Arithmetic**

1. **ADD A, Rn**; Adds the register Rn to the accumulator

Description: Instruction adds the register Rn (R0-R7) to the accumulator.

After addition, the result is stored in the accumulator

Before execution:

A=2Eh R4=12h

After execution:

A=40h R4=12h

1. **ADD A, #DATA**

Data: constant within 0-255 (0-FFh)

Description: Instruction adds data (0-255) to the accumulator. After addition, the result is stored in the accumulator.

ADD A, #33h

Before execution: A= 16h

After execution: A= 49h

1. **INC A** - Increments the accumulator by1

Description: This instruction increments the value in the accumulator by 1.

If the accumulator includes the number 255, the result of the operation will be 0.

Before execution: A=E4h

After execution: A=E5h 6.

1. **DEC A** -Decrements the accumulator by 1

Description: Instruction decrements the value in the accumulator by 1.

If there is a 0 in the accumulator, the result of the operation is FFh. (255 dec.)

Syntax: DEC A;

Byte: 1 (instruction code);

1. **DIV AB**  -Divides the accumulator by the register B

Description: Instruction divides the value in the accumulator by the value in the B register.

After division the integer part of result is stored in the accumulator while the register contains the remainder.

In case of dividing by 1, the flag OV is set and the result of division is unpredictable.

**The 8-bit quotient is stored in the accumulator and the 8-bit remainder is stored in the B register.**

Before execution:

A=FBh (251dec.) B=12h (18 dec.)

After execution:

A=0Dh (13dec.) B=11h (17dec.)

13·18 + 17 =251

1. **MUL A B** -Multiplies A and B

Description: Instruction multiplies the value in the accumulator with the value in the B register.

The low-order byte of the 16-bit result is stored in the accumulator, while the high byte remains in the B register.

If the result is larger than 255, the overflow flag is set. The carry flag is not affected.

Before execution:

A=80 (50h)B=160 (A0h)

After execution:

A=0 B=32h A·B=80·160=12800 (3200h)

**LOGICAL**

**SWAP A** - Swaps nibbles within the accumulator

A: accumulator

A nibble refers to a group of 4 bits within one register (bit0-bit3 and bit4-bit7). This instruction interchanges high and low nibbles of the accumulator.

Syntax: SWAPA

Before execution: A=E1h (11100001)bin.

After execution: A=1Eh (00011110)bin.

**ANL** **A,Rn**  - AND register to the accumulator

A: accumulator

Rn: any R register(R0-R7)

Instruction performs logic AND operation between the accumulator and Rn register.

The result is stored in the accumulator.

Syntax: ANL A,Rn

Before execution: A= C3h (11000011 Bin.) R5= 55h (01010101 Bin.)

After execution: A= 41h (01000001 Bin.)

**ORL A,Rn** - OR register to the accumulator

Rn: any R register (R0-R7)

Instruction performs logic OR operation between the accumulator and Rn register.

The result is stored in the accumulator.

Syntax: ORLA,Rn

Before execution: A= C3h (11000011 Bin.) R5= 55h(01010101 Bin.)

After execution: A= D7h (11010111 Bin.)

**CLR A** - Clears the accumulator

A: accumulator

Instruction clears the accumulator.

Syntax: CLR A

After execution: A=0

**Features and Architecture of Pentium Processor**

**Following are the features that Pentium processor offers:**

Some of the features of Pentium architecture are:

• Complex Instruction Set Computer (CISC) architecture with Reduced Instruction Set Computer (RISC) performance.

• 64-Bit Bus

• Upward code compatibility.

• Pentium processor uses Superscalar architecture and hence can issue multiple instructions per cycle.

• Multiple Instruction Issue (MII) capability.

• Pentium processor executes instructions in five stages. This staging, or pipelining, allows the processor to overlap multiple instructions so that it takes less time to execute two instructions in a row.

• The Pentium processor fetches the branch target instruction before it executes the branch instruction.

• The Pentium processor has two separate 8-kilobyte (KB) caches on chip, one for instructions and one for data. It allows the Pentium processor to fetch data and instructions from the cache simultaneously.

• When data is modified, only the data in the cache is changed. Memory data is changed only when the Pentium processor replaces the modified data in the cache with a different set of data

• The Pentium processor has been optimized to run critical instructions in fewer clock cycles than the 80486 processor.

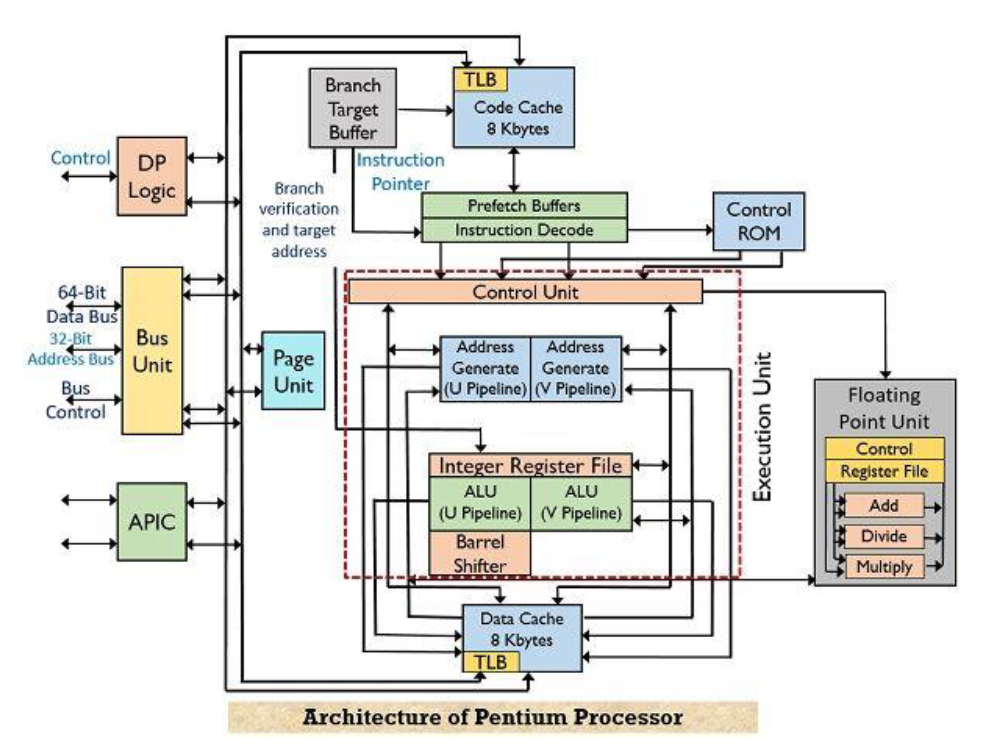
We have already mentioned in the beginning that Pentium is a superscalar processor. So, let us first understand,

**What are Superscalar Processors?**

A special category of microprocessors that involves a parallel approach for instruction execution called instruction-level parallelism through which more than one instruction gets executed in one clock cycle is called superscalar processors. It is famous as a second-generation RISC processor because RISC is the ones that operate in a faster manner with reduced instruction sets. Unlike scalar processors that have the ability to execute maximal one instruction per clock cycle, the superscalar processor uses the approach of simultaneously executing two instructions in one clock cycle. The superscalar processors perform this task by sending multiple instructions to various execution units at the same time. Hence this provides high throughput. It is to be noted here that superscalar processors are generally pipelined. However, pipelining is different from super scaling in a way that

superscalars allow execution of multiple instructions parallelly using multiple execution units while pipelining uses a single execution unit which is divided into multiple phases in order to execute multiple instructions

**Architecture of Pentium Microprocessor**



The various functional units are as follows:

1.Bus unit

2.Paging unit

3.Control ROM

4.Prefetch buffer

5.Execution unit with two integer pipeline (U-pipe and V-pipe)

6.Code cache

7.Data cache

8.Instruction decode

9.Branch target buffer

10.Dual processing logic

11.Advanced programmable interrupt controller

Let us now understand, how the architectural operation takes place. The bus unit of the architecture sends the control signal and fetches cade and data from external memory and IO devices. The size of the external data bus is 64-bit through which burst read and burst write-back cycles can be achieved. The paging unit in the architecture provides optional extensions of around 2 to 4 Mb page sizes. In order to load the instructions into the execution unit, code cache, branch target buffer and prefetch buffers operate together. The code cache or the external memory holds the instructions from where these are fetched. While the branch target buffer holds the address of the respective branch and the TLB (translational lookaside buffer) within the code cache converts the linear address into the physical address that is used by the code cache. This processor contains pairs of prefetch buffers having a size of 32-byte that combinedly operate with branch target buffer. Both the buffers operate independently but not at the same time. One of the prefetch buffers starts fetching the instructions in a sequential manner till the time branch instruction has not occurred. However, as soon as the branch instruction is fetched by the prefetch buffer then BTB will check for the branch but once it is checked by BTB that branch has not occurred then linear fetching of instruction will continue. On the contrary, while checking if BTB gets to know about the occurrence of the branch then the other prefetch buffer in pair gets enabled and starts fetching the instructions from the branch target address. By doing so, the branching instructions get simultaneously fetched and are ready for decoding and execution. The execution unit within the Pentium microprocessor contains two integer pipelines namely U-pipe and V-pipe and each one has its separate ALU. There are five stages in which these pipelines operate, namely, prefetch, decode-1, decode-2, execute, writeback. The U-pipe is responsible for executing all integer as well as floating-point instructions while V-pipe executes simple integer and some floating-point instructions. Here, the instruction fetch reads the instruction one at a time and stores them in the instruction queue. During the execution of an instruction, the processor does not sit idle and checks for the next two instructions in the queue. If the two instructions are independent of each other then U-pipe

and V-pipe are assigned instructions individually so that execution can occur simultaneously. However, in the case, the queued instructions are dependent on each other then both the instructions are assigned to U-pipe for execution one after the other and V-pipe remains idle. The controlling of the operations of the Pentium processor is provided by the control ROM that has a microcode within it. The control ROM directly controls U-pipe and V-pipe. Both data and code cache within the processor is organized in the 2-way associated set cache. Each cache has 128 sets and each set has 2 lines which are 32 bytes wide. The LRU (Least Recently Used) mechanism handles the cache replacement. As we can see clearly in the above figure that the code cache forms a connection with the prefetch buffer by a bus of size 256 bit, thus 256/8 i.e., 32 bytes of opcode can be buffered in one clock cycle. The data cache has two ports that are used to simultaneously deal with two data references. There is an on-chip Advanced Programmable Interrupt Controller that manages interrupt and offers 8259A compatibility.