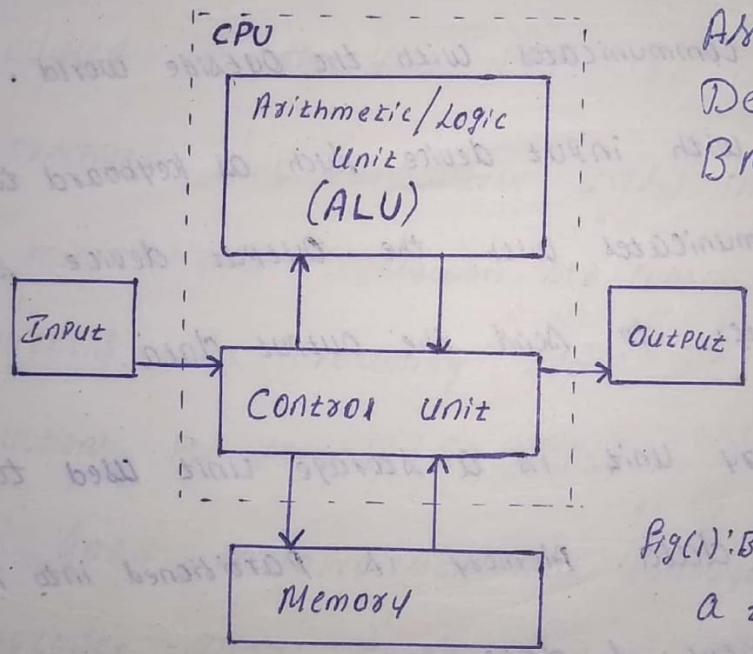
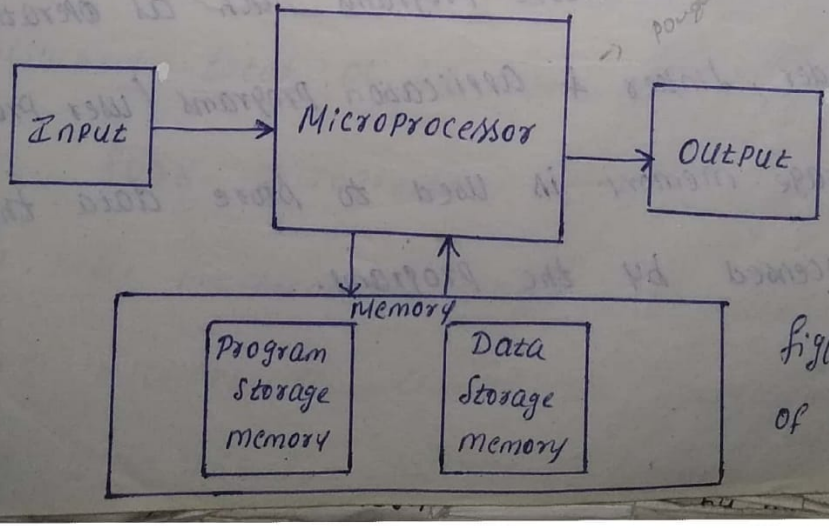
Overview Of Microcomputer Structure And Operation

In the late 1960's, the CPU was designed with discreate components on various boards. With the advent of the integrated circuit technology, it became possible to build entire CPU on a single board known as Microprocessor. The below figure (1) shows the block diagram of a traditional computer & figure(2) shows the block diagram of a microcomputer system.

Rajeshwari BS
Assistant Professor
Dept. of CSE
BMS College of Engg
Bengaluru.



fig(1): Block diagram of a traditional computer



fig(2): Block diagram of microcomputer

As shown in above figure, Microcomputer System has four functional components: Input Unit, Output Unit, Memory Unit & Microprocessing Unit (CPU)

The heart of the Microcomputer System is Microprocessing Unit known as Microprocessor that is capable of performing all arithmetic & logical operations & making decisions to change the sequence of program execution.

Input Unit & Output Unit are the means by which microprocessor communicates with the outside world. Microprocessor communicates with input device such as keyboard to accept data & communicates with the output device such as monitor, printer to send the output data.

Memory unit is a storage unit used to store programs & data. Memory is partitioned into program storage memory & data storage memory. Program storage memory is used to store programs such as operating system, compiler, Loader, Linker & application programs (user programs). Data storage memory is used to store data that are to be processed by the programs.

Evolution of Microprocessor

King
PAGE NO:
DATE: / / 200

Microprocessor is a semiconductor device consisting of electronic logic circuits manufactured by using either Large Scale Integration (LSI) or very Large Scale Integration (VLSI) technology capable of performing various arithmetic & logical operations & making decisions to change the sequence of program execution.

The world's first microprocessor, intel 4004 introduced in the year 1971. It was a 4 bit microprocessor was particularly useful in calculators, 4004 microprocessor instruction set contained only 45 instructions. The processing speed was only 50 kilo instructions per second (50 KIPS)

4004 microprocessor was followed by an 8 bit microprocessor 8008 introduced in the year 1972. 8008 microprocessor instruction set contained few additional instructions total of 48 instructions.

8008 microprocessor was followed by an 8 bit top intel microprocessor 8085 introduced in the year 1976 8085 microprocessor executed the instruction at even higher speed with 7,69,230 instruction executed per second.

8 bit 8085 microprocessor was followed by a 16 bit microprocessor Intel 8086 introduced in the year 1978. 8086 microprocessor can address 1 MB (Mega Byte) of memory & can execute 2.5 millions of instructions per second. Another important feature added to 8086 processor is was 6 byte instruction Queue was added on chip to prefetch the instructions in advance before executing to support parallel execution to speed up the execution. few additional instructions were added to the instruction set of 8086 processor.

8086 microprocessor was followed by another 16 bit microprocessor 80186 introduced in the year 1980, which included 8086 processor along with a number of I/O interface & other logical units on the same chip. A small number of new instructions were added to the instruction set of 80186. Memory capacity of 80186 was 1 MB.

1 MB memory capacity of 80186 processor proved limited memory for large database & for other applications. This led intel to introduce next version microprocessor 80286 in the year 1982. Memory capacity of

80286 microprocessor was 16MB. 80286 microprocessor was also a 16 bit microprocessor. Few additional instructions were added to the instruction set of 80286.

80286 microprocessor can execute 4.0 millions instructions per second

Applications began to demand still faster microprocessor & more memory. This led to introduce a 32 bit microprocessor intel 80386 in the year 1985. memory capacity of 80386 microprossor was 4 GB (Giga Byte)

intel 80386 microprocessor was followed by another 32 bit microprocessor intel 80486 introduced in the year 1989. memory capacity of 80486 microprocessor was 4 GB + 8 kB cache memory. 80486 microprocessor has on chip floating point unit to support floating point operation, on chip 8 kB of cache memory. 80486 microprocessor can execute 50 million instructions per second.

80486 microprocessor was followed by the pentium microprocessor introduced in the year 1993. pentium processor sometimes referred as 80586 Processor. memory capacity of pentium processor was 4 GB + 16 kB

On chip Cache memory. Pentium is a 64 bit processor. Pentium processor contained 8 kB of instruction cache memory & 8 kB of data cache memory.

Next introduced processor was Pentium 2 with a memory capacity of 64 GB + 32 KB instruction cache + 256 kB of data cache.

Next introduced processor was Pentium 3 with a memory capacity of 64 GB + 512 kB instruction cache + 256 kB of data cache

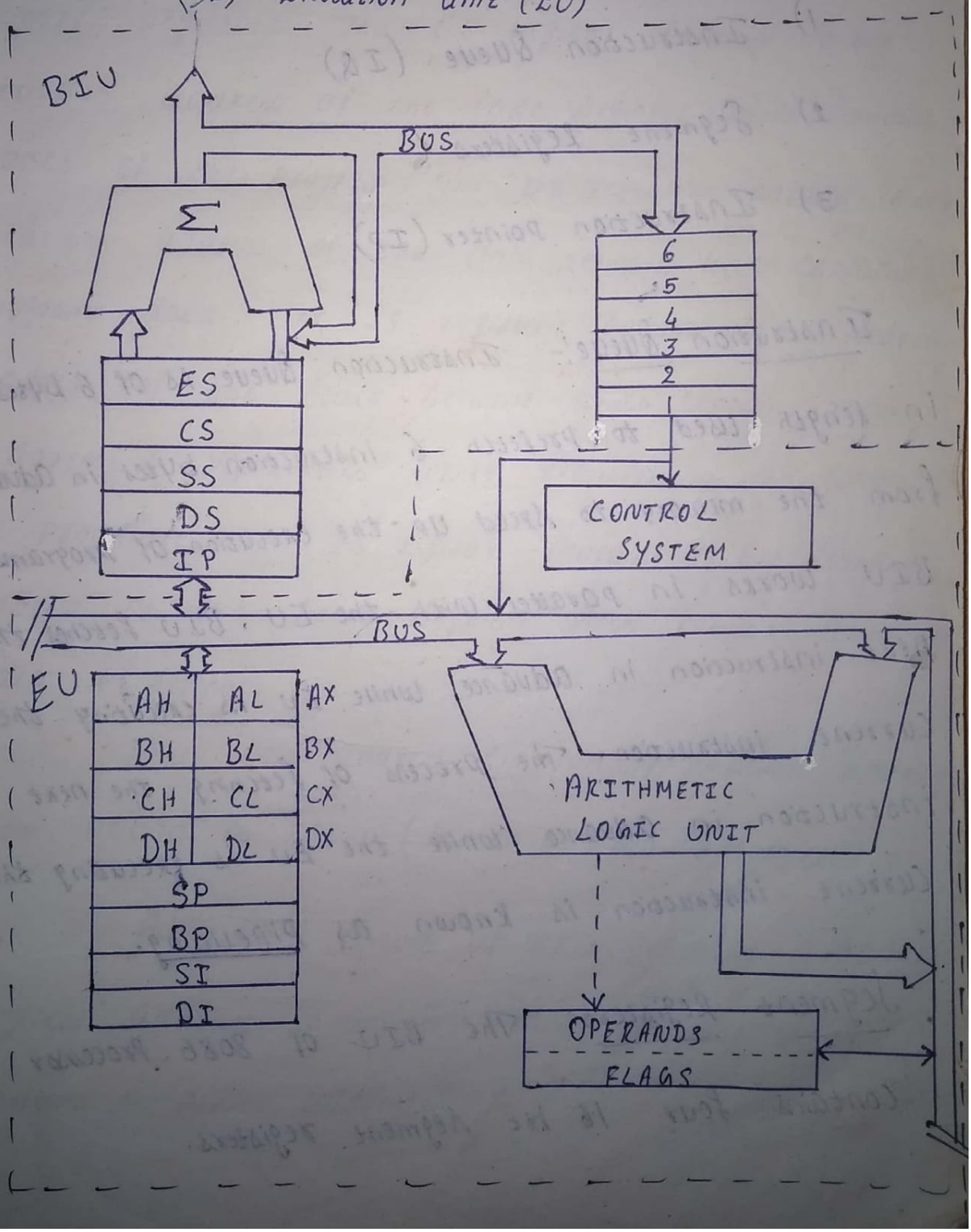
Pentium 4 microprocessor was made available in the year 2000. Pentium 4 microprocessor operates at a frequency of 3.2 GHz. It supports the concept of multithreading.

Pentium 4 microprocessor was followed by Pentium 4 D Dual core processor, which is a 64 bit microprocessor. The memory capacity of Dual core processor is 1 TB (Tera Byte)

8086 Internal Architecture

The Internal Architecture of 8086 Microprocessor is as shown below. It consists of two independent functional units

1) Bus Interface Unit (BIU)
2) Execution Unit (EU)

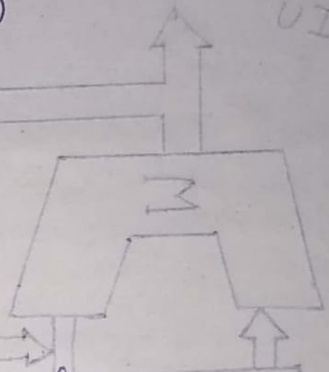


Bus Interface Unit (BIU)

BIU is responsible for instruction fetch, data fetch, computation of physical address of the memory, & address transfer on address bus to fetch instruction & data

The functional parts of the BIU are

1) Instruction Queue (IQ)
2) Segment Registers
3) Instruction pointer (IP)



Instruction Queue:- Instruction Queue is of 6 bytes in length used to prefetch 6 instruction bytes in advance from the memory to speed up the execution of programs. BIU works in parallel with the EU. BIU fetches the next instruction in advance, while EU is executing the current instruction. The process of fetching the next instruction in advance while the EU is executing the current instruction is known as **pipelining.**

Segment Registers: The BIU of 8086 Processor contains four 16 bit segment registers.

1) Code Segment (CS) Register
2) Data Segment (DS) Register
3) Stack Segment (SS) Register
4) Extra Segment (ES) Register

These 4 segment registers are used to store the 16 bit starting address of the corresponding four memory segments. The CS register contains 16 bit starting address of the code segment, which contains codes of the program. The DS register contains 16 bit starting address of the data segment, which contains program data. The SS register contains 16 bit starting address of the stack segment, which contains return address & parameters while transferring the control to procedure. The ES register contains 16 bit starting address of the Extra segment which contains additional data & strings.

BIU can address the memory locations starting from 00000 to FFFFF (ie. 0 to 1MB of memory)

Adder (Σ) circuit of BIU generates a 20 bit physical address of memory using segment register content & offset address.

Instruction Pointer (IP)

Instruction pointer is a 16 bit register contains the offset address of memory of the next instruction to be execute. As the instruction is executed, IP will be incremented to point to next memory location.

Execution Unit (EU)

Execution Unit is responsible for decoding & executing an instruction. It reads instruction bytes from an instruction Queue, decodes the instruction & identifies, which operation to be performed & performs the specified operation. It generates data addresses, if necessary, passes it to BIU either to read data from memory or input device to perform operation or to write result into the specified memory location or output devices.

The functional parts of EU are

1) Control System & Instruction Decoder
2) Arithmetic & Logic Unit
3) General purpose registers
4) Stack pointer Register
5) Pointer & Index Register
6) Flag Register

Control System & Instruction Decoder.

Control System of the Eu directs all the internal operations of the processor. Instruction decoder of the Eu decodes the fetched instruction & identifies the operation to be performed.

Arithmetic & Logic Unit

Arithmetic & logical circuit of Eu performs various 8 bit or 16 bit arithmetic operations such as addition, subtraction, multiplication & division & logical operations such as AND, OR, NOT, XOR & shift operations

General Purpose Registers

Eu of 8086 processor has 4 16 bit general purpose registers AX, BX, Cx & DX. This 4 16 bit general purpose registers can be used to perform 16 bit operations. Each of these 16 bit registers can be divided as 2 8 bit registers identified as high order registers & low order registers referred as AH, AL, BH, BL, CH, CL, DH & DL can be used to perform 8 bit operations.

Stack Pointer Register (SP)

Stack pointer register is a 16 bit register contains address of top of the stack.

Pointer & Index Register

8086 processor has 16 bit pointer register known as Base pointer (BP) used as a pointer to a memory location to access the operand similar to Index register SI & DI.

8086 processor has 2 Index registers

* Source Index (SI)
* Destination Index (DI)

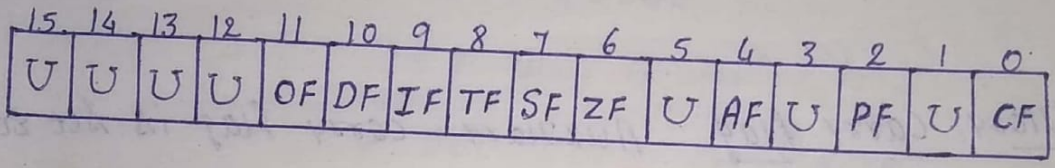
SI & DI registers are 16 bit registers used as a pointer to a memory location or to store index values for accessing the array elements.

Flag Register

The flag register of 8086 processor is a 16 bit register stores the information about status of the processor & status of the recently executed instructions.

Flag Register of 8086

8086 Microprocessor has a 16 bit flag register that stores the information about the status of the procesor & status of the most recently executed instruction. The format of 8086 flag register is as shown below.



Out of 16 bits, 8086 processor uses only 9 bits to store the information of status of the processor & status of the most recently executed instruction.

Out of 9 flags, 6 flags are called as Status flags stores status of most recently executed instruction and 3 flags are called as control flags stores the information about the status of the processor.

6 status flags & 3 control flags are

Status flags

1) Carry flag (CF)
2) Parity flag (PF)
3) Auxillary carry flag (AF)
4) Zero flag (ZF)
5) Sign flag (SF)
6) Overflow flag (OF)

Control flags

1) Trap flag (TF)
2) Interrupt flag (IF)
3) Direction flag (DF)

Carry flag:- Carry flag is set to 1 (ie, CF=1), if an addition produces a carry out of MSB or if a subtraction needs a borrow for MSB, otherwise it is reset to 0.

Parity flag:- Parity flag is set to 1 (ie, PF=1), if the low order 8 bits of the result contains even number of 1's, otherwise it is reset to 0

Auxillary Carry flag Auxillary carry flag is set to 1 (ie, AF=1), if an addition produces a carry from 3rd bit to 4th bit or subtraction needs a borrow from 4th bit to 3rd bit, otherwise it is reset to 0.

Zero flag:- Zero flag is set to 1 (ie, ZF=1), if the result after performing addition or subtraction operation is 0, otherwise it is reset to 0.

Sign flag:- Sign flag is set to 1 (ie, SF=1), if the most significant bit (MSB) of the result is 1, ie, If the result is negative, otherwise it is reset to 0

Overflow flag:- Overflow flag will be affected only, when signed numbers are added or subtracted. Signed numbers are represented in 8086 based computer as follows.

Representation of Signed numbers:- Positive numbers are represented with Most significant bit 0 & remaining 7 bits represents magnitude

for eg:- +3 using 8 bits

0 0000 0ϕϕ

+3 using 16 bits

0 000 0000 0000 0ϕϕ

ie, maximum signed value that can be stored in 8 bit register or 8 bit memory variable is 7F (ie, 0111 1111) but maximum unsigned value that can be stored in 8 bit register or 8 bit memory variable is FF (ie, 1111 1111), maximum signed value that can be stored in 16 bit register or 16 bit memory variable is 7FFF (ie, 0111 1111 1111 1111) but maximum unsigned value that can be stored in 16 bit register or 16 bit memory variable is FFFF (ie, 1111 1111 1111 1111)

Negative numbers are represented in 8086 with Most significant bit 1 & remaining 7 bits represents magnitude

Negative numbers are represented by its 2's complement notation.

∴ −3 is represented as

0000 0000 0000 0011 = 3H

Its 2's Complement

1111 1111 1111 1100

+ 1

1111 1111 1111 1101 = − FFFDH

Overflow flag will be affected only, when signed numbers are added or subtracted. If the result after performing addition of same signed numbers or subtraction of opposite signed number is more than 7 bits for an 8 bit operation or more than 15 bits for an 16 bit operation, then the overflow flag is set to 1, otherwise it is reset to 0.

Eg: 1)

$$\begin{array}{llll} 0010 & 1100 & 0011 & 0101 = 2C35H \\ +\ 0010 & 0101 & 0101 & 1010 = {}^{+}255AH \\ \hline 0101 & 0001 & 1000 & 1111 \end{array}$$

CF = 1, ZF = 0, SF = 0, PF = 0, AF = 0, OF = 0 (Always 0 for unsigned operation)

2)

$$\begin{array}{llll} 1010 & 0101 & 0000 & 1010 = A50A \\ 1110 & 0000 & 1010 & 1000 = {}^{+}E0A8 \\ \hline 1000 & 0101 & 1011 & 0010 \end{array}$$

CF = 1, AF = 1, PF = 1, SF = 1, OF = 0, ZF = 0

(Always 0 for unsigned operation)

Eg: 3) Add -1234 & -7F2E

-1234
-7F2E
-9162

-1234 ⇒ Sign bit 1 001 0010 0011 0100
-7F2E ⇒ Sign bit 1 111 1111 0010 1110

Sign bit ← 1 001 0001 0110 0010

∴ result = -1162 (Wrong Answer ∵ result overflow 14th bit producing carry to 15th bit, which is sign bit)

CF=1, AF=1, ZF=0, SF=1, OF=1, PF=1

∴ result is overflow to sign bit

NOTE: Unsigned numbers range 0000 - FFFF (16 bits)
00 - FF (8 bits)

Signed numbers range -7F to +7F (8 bits)
1111 1111 to 0111 1111
-7FFF to +7FFF (16 bits)
1111 1111 1111 1111 to 0111 1111 1111 1111

Trap flag: If TF=1, then 8086 microprocessor goes for an Single step mode of operation. If TF=1, then 8086 Processor executes an instruction & then transfers the control to Trap Interrupt Service routine, which displays current register contents & memory variables contents on CRT. After the execution of Service routine control comes back to program & 8086 processor executes next instruction

After the execution of instruction, once again control will be transfered to Trap interrupt service routine & displays current contents of register & memory variables. This type of operation is useful for debugging program. If TF=0, then 8086 processor executes complete program & displays result. There is no direct instruction to set the trap flag, but user indirectly can set trap flag to 1.

Interrupt flag:- If IF=1, then external interrupt signal comes on INTR pin of 8086 processor can be recognised by the processor & services the request. If IF=0, then 8086 microprocessor does not recognise external interrupt signal comes on INTR pin. User can set the IF by the instruction STI (Set interrupt). ~~If~~ can clear interrupt flag by the instruction CLI (clear Interrupt flag).

Direction flag:- Direction flag determines the direction in which string operations should process. If DF=0, then the string is processed with the first character having lowest address & progressing towards highest address. If DF=1, then the string is processed from last character having highest address & progressing towards lowest address. DF can be reset by the user by the instruction CLD

(Clear Direction flag) & SI & DI register content will be automatically incremented by 1. DF can be set by the user using an instruction STD (Set Direction flag) & SI & DI register content will be automatically decremented by 1

Decimal	Hexadecimal	Binary (2^4 2^3 2^2 2^1 2^0)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001

I Conversion from Decimal number to Hexadecimal

1) $17_{(10)}$

$\Rightarrow 16 \mid 17$

$1 - 1$

$\Rightarrow 11_{(16)} = 11H$

2) $25_{(10)}$

$\Rightarrow 16 \mid 25$

$1 - 9$

$\Rightarrow 19_{(10)} = 19H$

3) $125_{(10)}$

$\Rightarrow 16 \mid 125$

$7 - D$

$\Rightarrow 7D_{(16)} = 7DH$

4) $256_{(10)}$

$\Rightarrow 16 \mid 256$

$16 \mid 16 - 0$

$1 - 0$

$\Rightarrow 100_{(16)} = 100H$

5) $1234_{(10)}$

$\Rightarrow 16 \mid 1234$

$16 \mid 77 - 2$

$4 - D$

$\Rightarrow 4D2_{(16)} = 4D2H$

6) $12583_{(10)}$

$\Rightarrow 16 \mid 12583$

$16 \mid 786 - 7$

$16 \mid 49 - 2$

$3 - 1$

$\Rightarrow 3127H$

II Conversion from Hexadecimal to Decimal

1) $11_{(16)}$

$= 1 \times 16^1 + 1 \times 16^0$

$= 16 + 1 = 17_{(10)}$

2) $19_{(16)}$

$= 1 \times 16^1 + 9 \times 16^0$

$= 16 + 9 = 25_{(10)}$

3) $7D_{(16)}$

$= 7 \times 16^1 + 13 \times 16^0$

$= 7 \times 16 + 13 \times 1$

$= 112 + 13 = 125_{(10)}$

4) $100_{(16)}$

$= 1 \times 16^2 + 0 \times 16^1 + 0 \times 16^0$

$= 1 \times 16^2 + 0 + 0$

$= 1 \times 16 \times 16 + 0 + 0$

$= 256 + 0 + 0 = 256_{(10)}$

5) $4D2_{(16)}$

$= 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0$

$= 4 \times 16 \times 16 + 13 \times 16 + 2 \times 1$

$= 1024 + 208 + 2$

$= 1234_{(10)}$

6) $12D5E_{(16)}$

$= 1 \times 16^4 + 2 \times 16^3 + 13 \times 16^2 + 5 \times 16^1 + 14 \times 16^0$

$= 65536 + 8192 + 3328 + 80 + 14$

$= 77150_{(10)}$

Binary coded decimal (BCD)

Binary coded decimal (BCD) is a method of representing each decimal digit by a 4 bit Binary value.

for eg. 5 = 0101

9 = 1001

32 = 0011 0010

65 = 0110 0101

BCD data are sometimes used in a computer system to store decimal data.

BCD data is stored in computer in two ways

BCD — Unpacked BCD
BCD — Packed BCD

In unpacked BCD, Each digit are stored in one byte

for eg:- decimal 12

unpacked BCD
01 02 (0000 0001
0000 0010)
0 2

625 06 02 05
(0000 0110
0000 0010
0000 0101)
0 6
0 2
0 5

In packed BCD, Two digits are stored in one byte

for eg:- decimal 12

packed BCD
12 (0001 0010)
1 2

625 625 (0000 0110 0010 0101)
0 6 2 5

Addressing modes

Addressing modes is defined as a method of specifying effective address (EA) of an operand

8086 Processor supports following addressing modes to specify EA of an operand.

1) Immediate Addressing mode
2) Register Addressing mode
3) Direct Addressing mode
4) Indirect Addressing mode
5) Register Relative Addressing mode
6) Based Indexed Addressing mode
7) Relative Based Indexed Addressing mode
8) Implicit Addressing mode

Rajeshwari. B.S
Assistant Professor
CSE Department
BMSCE, Bangalore

The registers of 8086 involved in the computation of Effective address of an operand are

1) Base Registers : BX & BP
2) Index Registers : SI & DI

1) Immediate Addressing mode

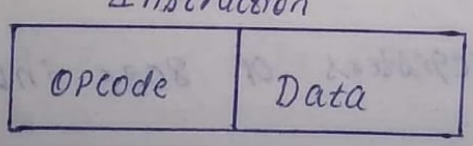
In Immediate Addressing mode, the operand on which the instruction operates are specified in the instruction itself.

for eg:- 1) MOV AX, 1234

2) ADD AL, 30

The operand 1234 in the instruction MOV AX, 1234 is a part of the instruction. This instruction moves data 1234 to AX register.

The operand 30 in the instruction ADD AL, 30 is a part of the instruction. This instruction adds AL register content with immediate value 30 & stores the result back to AL register. The process of Immediate addressing mode is as shown



2) Register Addressing mode

In Register Addressing mode, the operand on which instruction operates are stored in any of the registers. For a 16 bit operand, the register may be

AX, BX, CX, DX, SI, DI, SP, BP & for an 8 bit Operand, the register may be AH, AL, BH, BL, CH, CL, DH and DL.

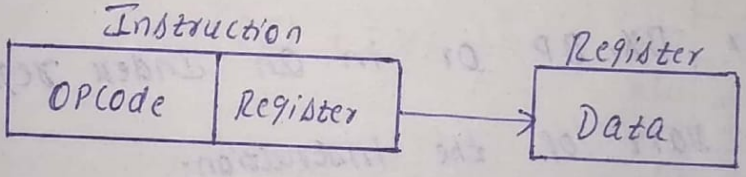
for eg:- 1) MOV BX, CX

The 16 bit Operand is in CX register. This instruction copies Operand from CX register to BX register

2) ADD AL, BL

The Operands are in registers AL & BL, After addition result will be stored back to AL register.

The process of Register addressing mode is as shown below



3) Direct Addressing mode

In direct addressing mode, the effective address of an Operand on which instruction Operates are specified directly in the instruction itself.

for eg:- MOV AL, A

If address of A is 2000. After converting Pgm into Object Code, A will be replaced by [2000]

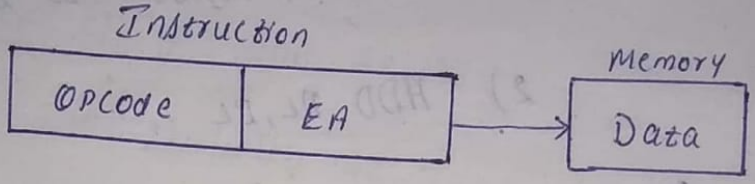
(1)

MOV AL, [2000]

The Operand is in memory location 2000, this instruction copies the content of memory location 2000 to AL register.

A	10	2000

The process of Direct Addressing mode is as shown below



4) Indirect Addressing mode

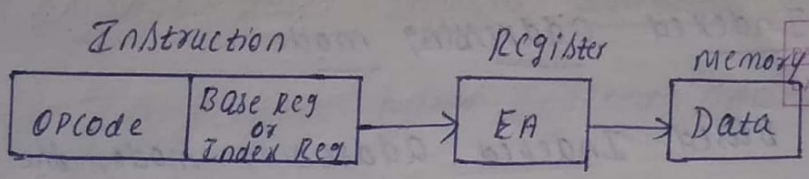
In Indirect addressing mode, the effective address of an operand is specified either in Base register BX, BP or in an Index register SI, DI, which is a part of the instruction.

for eg:- LEA BX, A [2000] ; Load effective address of a variable A into BX register

MOV AL, [BX]

If BX register contains 2000, the effective address of a variable A, then the instruction MOV AL, [BX] copies the content of location 2000 pointed by BX register to AL register.

The process of Indirect addressing mode is as shown



5) Register Relative addressing mode

In Register Relative addressing mode, the effective addres of an operand is generated by adding the content of Base register BX, BP or Index register SI, DI with an 8 bit or 16 bit displacement value (constant)

ie,

$$EA = \begin{cases} (BX) + DISP \\ (BP) + DISP \\ (SI) + DISP \\ (DI) + DISP \end{cases}$$

A[0]	10	2000
A[1]	20	2001
A[2]	30	2002
A[3]	40	2003
A[4]	50	2004

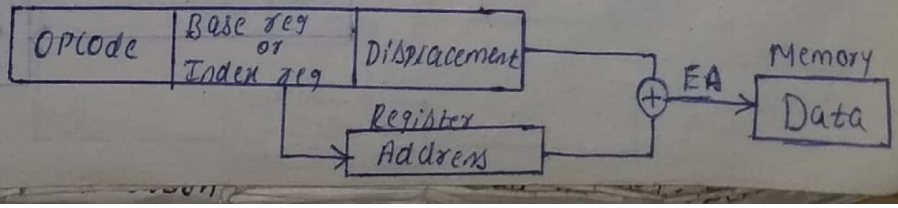
for eg. LEA BX, A

~~MOV SI, 03~~

MOV AL, [BX+03] ; Copies content of A[3] into AL reg

If base address of array A is 2000, then 2000 +03 = 2003 ie, A[3] location content will be moved to AL reg.

The process of Register Relative addressing mode is as shown below.



6) Based Indexed addressing mode

In Based Indexed addressing mode, the effective address of an operand is generated by adding the content of base register (BX or BP) and content of Index register (SI or DI)

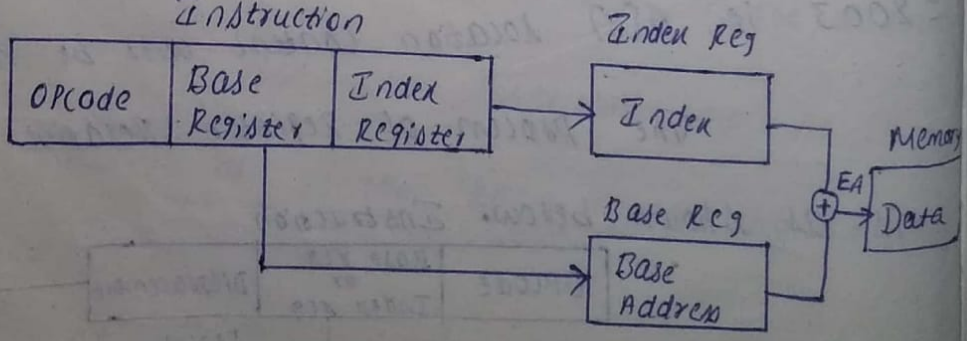
$$ie,\ EA = \begin{cases} (BX) + (SI) \\ (BX) + (DI) \\ (BP) + (SI) \\ (BP) + (DI) \end{cases}$$

Eg:-

```
LEA  BX, A
MOV  SI, 03
MOV  AL, [BX][SI]
```

If BX contains 2000, base addrem of array A, then
EA = 2000 + 03 = 2003 content ie, A[3] content will be moved to AL register.
[BX] [SI]

The Process of Based Indexed addressing mode is as shown below



7) Relative Based Indexed addressing mode

In relative based Indexed addressing mode, the effective address of an operand is generated by adding the contents of Base register (BX or BP) and content of Index register (SI or DI) with an 8 bit or 16 bit displacement value (constant)

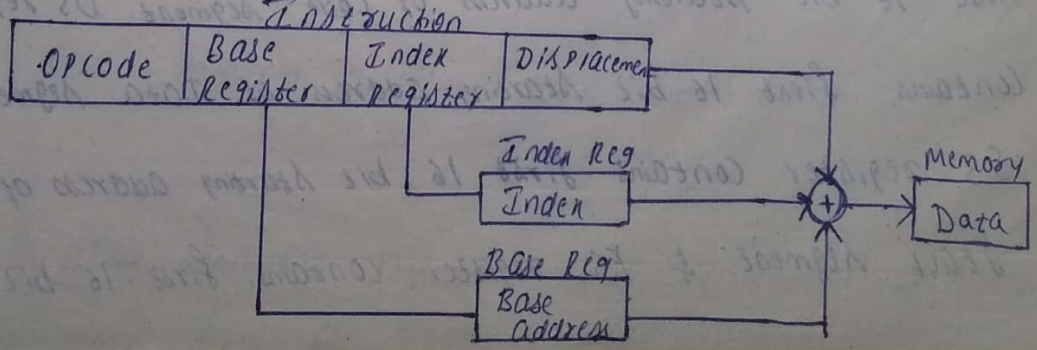
ie, $$EA = \begin{cases} (BX) + (SI) + disp \\ (BX) + (DI) + disp \\ (BP) + (SI) + disp \\ (BP) + (DI) + disp \end{cases}$$

Eg:-

```
LEA  BX, A
MOV  SI, 03
MOV  AL, [BX] [SI+03]
```

If BX contains 2000, base address of array A, then EA = 2000 + 03 + 03 = 2006 content ie, A[6] content will be moved to AL register.
[BX] [SI] Disp

The process of Relative Based Indexed addressing mode is as shown below



8) Implicit addressing mode

In Implicit addressing mode, the instruction itself assumes that the data is in some predefined register. The register on which instruction operates vary from instruction to instruction.

Eg:- CMC ; Complement Carry This instruction assumes that the data is carry flag content & complements it

STD: Set Direction flag, This instruction assumes that the data is in direction flag & sets it content to 1.

Generating Physical memory address

Since in 8086, all the registers are 16 bits long. Segment registers contains first 16 bit starting address of the corresponding segments. CS register contains first 16 bit starting address of code segment. DS register contains first 16 bit starting address of Data segment. SS register contains first 16 bit starting address of Stack segment & ES register contains first 16 bit

Starting address of Extra Segment.

Apart from this, Instruction pointer contains Offset address (displacement with respect to base address). Instruction pointer (IP) contains offset address of next instruction to be executed in the code segment. If data is to be taken from the data segment, IP contains offset address of data to be fetch from the data segment.

Physical address is an 20 bit address used to access the memory.

Since all segment registers & instruction pointer are 16 bits registers, but physical address is 20 bits, there is required to convert 16 bit logical address into 20 bit physical address to access instructions or data from the memory.

The physical address is computed in 8086 as follows

Physical Address = 10H * Segment address + offset address

If Segment address (CS) = 2000

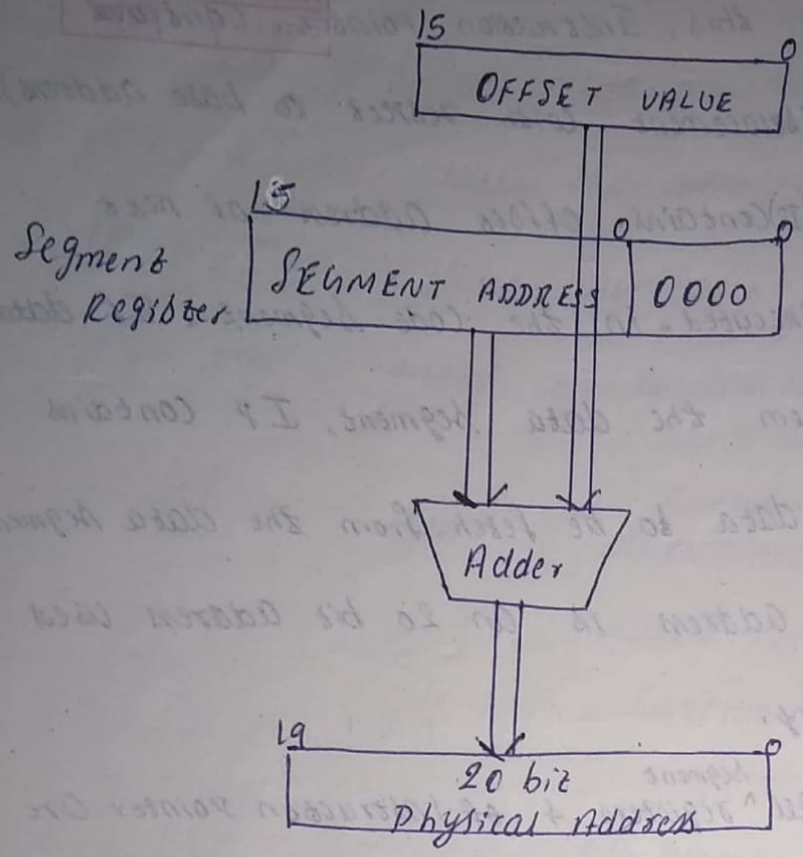
offset address (IP) = 0050

then Physical address = 10H * 2000 + 0050

= 20050H

After computing physical address, it fetches the instruction or data from the memory location 20050H.

Generating Physical address is as shown



Eg: If CS = 1234 = 0001 0010 0011 0100

Shifting left by four positions & filling with zeros resulting in segment base address

0001 0010 0011 0100 0000 = 12340

If IP = 0022 = 0000 0000 0010 0010

Then Adding segment base address 12340 & offset address 0022 gives 20 bit Physical address

0001 0010 0011 0100 0000 = 12340
0000 0000 0010 0010

0001 0010 0011 0110 0010 = 12362 H

8086 Instruction Set

King
PAGE NO:
DATE: / /200

An instruction in an assembly language program is represented by a set of characters called as Mnemonics. for eg. ADD, SUB, JMP, AND etc.

8086 processor provides exhaustive set of instructions to write assembly language program to perform a specific task by taking input, processing & producing result.

The instructions of 8086 processor are classified into following groups based on its functions

1) Data Transfer Instructions
2) Arithmetic & Logical Instructions
3) Branch Instructions
4) String Instructions
5) Processor control Instructions.

Assembler Instruction format

The general format of assembly language instruction is as follows

Label : Mnemonic Operand, Operand ; comment

Eg. [illegible]

* A Label name followed by a colon (:) is an identifier. The address of the label name can be used in Branch instruction to branch to the labeled instruction. The use of label with the instruction is optional.

* Mnemonics indicates the operation to be performed on the specified operands

* The operand specifies on which operation has to takes place. Number of operands to be specified depends upon the instruction. Instruction may contain zero operand Eg:- CMC, or One operand Eg: MUL BL or 2 operands Eg:- ADD AL,BL. If there are two operands, then the operands should be separated by comma (,).

* A comment must start with the character semicolon (;), which can be used for commenting the logical operation of an instruction. The comment field is optional.

Eg:-

```
        MOV AL,00
        MOV SI,00
        MOV CX, 05
BACK:   ADD AL, A[SI]
        INC SI
        LOOP BACK
```

Data Transfer Instructions

8086 processor supports variety of data transfer instructions for transferring of address & data to register or memory location. It involves in transferring of data from register to register, register to memory, memory to register, immediate data to register, immediate data to memory, register to segment register etc.

Data transfer instructions generally involves two operands source and Destination. Source can be register or memory variable or immediate value. The destination can be register or memory variable.

Both source & destination cannot refer to memory location

Both source & destination must be of same data type either byte type or word type.

Data transfer instructions does not affect CPU flags.

The various data transfer instructions supported by 8086 Processor are

1) MOV : Move
2) XCHG : Exchange
3) LEA : Load Effective Address
4) PUSH : Push register or memory onto the stack
5) POP : POP from the stack to register or memory
6) PUSHF : Push flag register onto the stack
7) POPF : POP from the stack to flag register
8) LAHF : Load AH with flags
9) SAHF : Store AH to flags
10) XLAT : Translate a byte
11) LDS : Load register & DS register
12) LES : Load register & ES register.

1) MOV: Move

The MOV instruction transfers either a byte or a word from source to destination. The source can be immediate value, or register or memory variable. The destination can be either a register or memory

Variable. Both source & destination cannot refer to memory location. Both source & destination must be of same data type either byte type or word type.

General form:

MOV dest, src ; dest ← src

Eg:

MOV AX, BX ; Copies content of BX to AX

MOV AX, A ; Copies content of memory location A to AX

MOV AL, 12 ; Copies immediate value 12 to AL

MOV A, CL ; Copies CL reg content to memory location A

MOV A, B ; Invalid, both cannot refer to memory

MOV AX, BL ; Invalid, Data type must match

2) XCHG: Exchange

The XCHG instruction exchanges the content of source register or memory location with the content of destination register or memory location. The source can be a register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory locations. Both source & destination must be of same

data type either byte type or word type

General form:-

XCHG dest, Src ; dest ⟷ Src

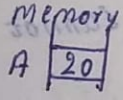
Eg:-

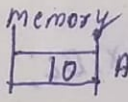
1) XCHG AX, BX

Before	After
AX = 1234	AX = 5678
BX = 5678	BX = 1234

2) XCHG AL, A

Before	After
AL = 10	AL = 20





XCHG AX, BL ; Invalid, Both must be of same data type

XCHG A, B ; Invalid, Both operands cannot refer memory location

3) LEA : Load Effective Address

LEA instruction determines offset address (Effective address) of a memory variable specified as a source & places it into the specified 16 bit

register. The source can be either memory variable or Index register & Base register. The destination must be 16 bit register.

General form

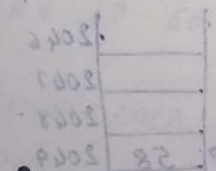
LEA reg16, src ; reg16 ← EA of src

Eg: LEA BX, A ; BX ← Effective Address of A variable

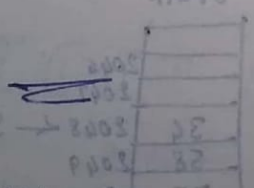
LEA AX, [BX][DI] ; If BX = 1000 (EA of A)
DI = 3
then
AX ← 1000 + 3 = 1003

Difference b/w MOV & LEA

LEA AX, [BX][DI] ; If BX = 1000 (EA of A)
DI = 3
then
AX ← 1000 + 3 = 1003



MOV AX, [BX][DI] ; If BX = 1000 (EA of A)
DI = 3
then
AX ← Content of location 1003



4) PUSH: Push register or memory location onto stack

PUSH instruction pushes the content of the specified source onto the stack. This instruction

Pushes a word from the specified source on to the stack pointed by stack pointer (SP). The source can be any 16 bit register, Segment register, Index register, Base register or memory variable. This instruction decrement SP by 1 & loads lower byte of word to stack ^, again SP will be decremented by 1 & loads higher byte of word on to stack.

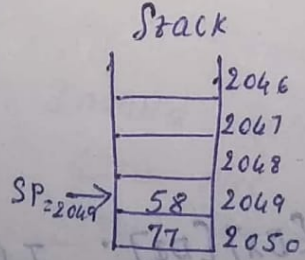
General form:

PUSH Src ; SP ← SP-1, [SP] ← lower byte of Src

SP ← SP-1, [SP] ← higher byte of Src

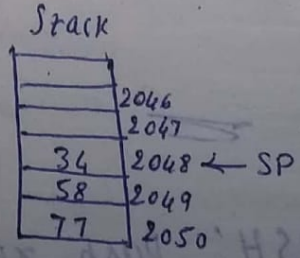
Eg:- ① PUSH AX

If AX = 1234



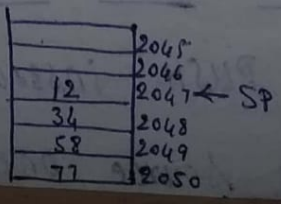
SP ← SP-1 = 2049-1 = 2048

[SP] ← 34

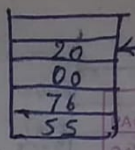


SP ← SP-1 = 2048-1 = 2047

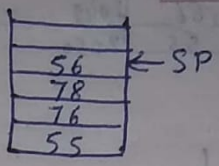
[SP] ← 12



② PUSH DS If DS = 2000



③ PUSH A If A = 5678



5) POP: POP register or memory from stack

The POP instruction POPs a word from the stack pointed by SP into the specified word register or word memory location. The destination can be any 16 bit register, segment register, Index register, Base register or memory variable. This instruction copies byte pointed by SP to higher order register or memory location & increments SP by 1. Copies byte pointed by SP to lower order register or memory location & increments SP by 1.

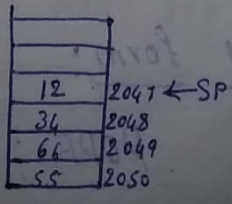
General form:

POP dest ; higher byte of src ← [SP]
SP ← SP+1
lower byte of src ← [SP]
SP ← SP+1

Eg: ① POP AX

Before

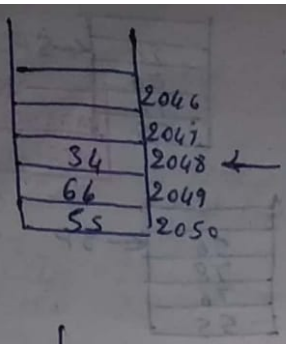
AX = 5578



AH = 12

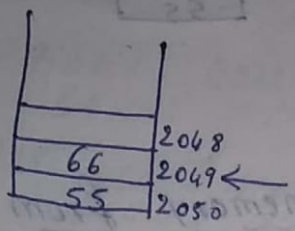
∴ AX = 1278

SP ← SP+1 = 2047+1 = 2048



AL = 34 ∴ AX = 1234

SP ← SP + 1 = 2048 + 1 = 2049



(2) POP A

(3) POP DS

(6) PUSHF: Push flag register onto the stack

The PUSHF instruction pushes the contents of flag register on to the stack

General form:-

PUSHF ; SP ← SP - 2 [SP] ← Flags

(7) POPF: POP from stack to flag register

The POPF instruction POPs the word from the stack pointed by SP into flag register

General form:

POPF ; flag reg ← [SP] & SP ← SP + 2

8) LAHF: Load AH register with flags

PAGE NO:
DATE: / / 200

LAHF instruction loads the lower byte of flag register to AH register. This instruction LAHF along with SAHF instruction is very useful for the simulation of 8085 microprocessor on the 8086 based system.

General form

LAHF ; AH ← Lower byte of Flag reg

9) SAHF : Store AH register to flag

The SAHF instruction copies (stores) the content of AH register into the lower byte of flag register. This instruction SAHF along with LAHF instruction is useful for the simulation of 8085 microprocessor on the 8086 microprocessor based system.

General form:

SAHF ; Flag lower byte of reg ← AH

=

10) XLAT / XLATB : Translate a Byte

The XLAT instruction replaces a byte in the AL register with a byte from the look up table. Before the execution of XLAT instruction, BX register should be loaded with the effective address of look up table & AL should be loaded with the code to be converted. When XLAT instruction is executed, then the byte pointed by (BX+AL) is transferred to AL register

General form

XLAT / XLATB ; AL ← [BX+AL]

Eg:

1) LEA BX, SQUARE ; BX ← 0050 (EA of Square)
   MOV AL, 06 ; AL ← 06
   XLAT ; AL ← [BX+AL]
   AL ← [0050+06]
   AL ← [0056]
   AL ← 36

	Square
0	20050
1	20051
4	20052
9	20053
16	20054
25	20055
36	20056
49	20057
64	20058
81	20059
100	20060

Eg: 2)

Convert a decimal number which is less than 16 into its equivalent hexadecimal number

```
.Data
        Hexa  DB  '0123456789ABCDEF'
        Code  DB  11
.code
        MOV  AX, @data
        MOV  DS, AX
        LEA  BX, Hexa  ; BX ← 0060
        MOV  AL, code  ; AL ← 11
        XLAT           ; AL ← [BX+AL]
End
```

AL ← [0060+11]

AL ← [0071]

AL ← 'B'

Hexa	
0	20060
1	20061
2	20062
3	20063
4	20064
5	20065
6	20066
7	20067
8	20068
9	20069
A	20070
B	20071
C	20072
D	20073
E	20074
F	20075

II) LDS: Load register and DS register

The LDS instruction copies the contents of specified first two memory locations to a specified 16 bit register & contents of next two memory locations to the DS register

General form:

LDS reg16, mem; reg16 ← [mem0 & mem1]
DS ← [mem2 & mem3]

Eg: LDS BX, P

BX ← 0050 (offset)

DS ← 2000 (Base address)

P[0]	50
P[1]	00
P[2]	00
P[3]	20

LDS BX, P instruction copies the content of P[0] to BL reg & content of P[1] to BH reg & copies the content of P[2] & P[3] to DS register.

12) LES: Load Register & ES register

The LES instruction copies the content of specified first two memory locations to a specified 16 bit register & contents of next two memory locations to the ES register

General form:

LES reg16, mem; reg16 ← [mem of mem]

ES ← [mem2 & mem3]

Eg: LES BX, P

BX ← 0060 (offset)

Es ← 4000 (Base address)

P[0]	60
P[1]	00
P[2]	00
P[3]	40

SHIFT INSTRUCTIONS AND ROTATE INSTRUCTIONS

Shift instructions shifts the specified byte or word either to the left or right by the specified number of bits.

8086 processor supports 4 types of shift instructions

1) SHL (Shift Logical Left)
2) SHR (Shift Logical Right)
3) SAR (Shift Arithmetic Right)
4) SAL (Shift Arithmetic Left)

1) SHL : Shift Logical Left

The SHL instruction shifts the destination byte or word left by the number of bits specified in the count. As the bits are shifted in left direction (←), most significant bit (MSB) goes to carry flag (CF) & in the vacancy created in least significant bit position (LSB), zero will be inserted.

The operation performed by SHL instruction is as shown below

CF ← MSB ← ——— LSB ← 0

General form

SHL dest, count

The destination can be a byte or word operand. It can be a register or memory variable. If count = 1, then it may be specified directly in the instruction. If count > 1, then the count value must be specified in CL register before SHL instruction.

Eg:- 1) SHL AL, 01

Before CF = 0 1 1 0 1 0 1 0 1

After CF = 1 1 0 1 0 1 0 1 0 ← 0

MOV CL, 03

2) SHL AL, CL

If AL = 1 1 0 1 0 1 0 1

CF = 1 1 0 1 0 1 0 1 0 ← 0

CF = 1 0 1 0 1 0 1 0 0 ← 0

CF = 0 1 0 1 0 1 0 0 0 ← 0

If AL = 0000 0011 = 03

Shift by 1 bit left ←

0000 0110 = 06 (3*2)

2nd shift ←

0000 1100 = 12 (3*4)

3rd shift ←

0001 1000 = 24 (3*8)

Shift by 1 bit left = *2

Shift by 2 bits left = *4

Shift by 3 bits left = *8

Shift by 4 bits left = *16

2) SHR : Shift Logical Right

The SHR instruction shift the destination byte or word right by the number of bits specified in the count. As the bits are shifted in right direction (→), LSB bit goes to CF & in the vacancy created in MSB position, 0 will be inserted.

The operation performed by SHR instruction is as shown below

0 → MSB → LSB → CF

General form

SHR dest, count

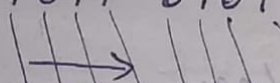
The destination can be a byte or word operand.

It can be a register or memory variable. If count=1, then it may be specified directly in the instruction. If count >1, then it must be stored in CL register before using SHR instruction.

Eg:

1) SHR AL, 01

Before: AL = 1011 0101 CY=0

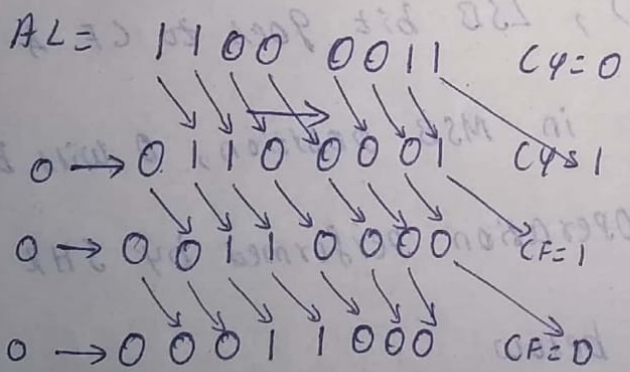


After 0 → 0101 1010 CY=1

2) MOV CL, 03

SHR AL, CL

If AL = 1100 0011 CY=0



0 → 0 1 1 0 0 0 0 1 CY=1

0 → 0 0 1 1 0 0 0 0 CF=1

0 → 0 0 0 1 1 0 0 0 CF=0

If AL = 0001 1000 = 24

Shift right by 1 bit →

0000 1100 = 12 24/2

2nd Shift →

0000 0110 = 6 24/4

Shift right by 1 bit = /2 (24/2)

Shift right by 2 bits = /4 (24/4)

Shift right by 3 bits = /8 (24/8)

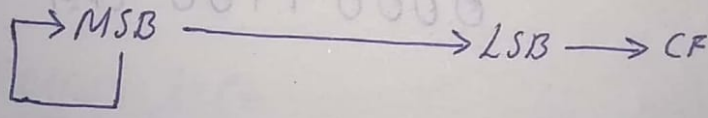
3rd Shift ⟶

0000 0011 = 3 $(24/8)$

3) SAR: Shift Arithmetic Right

The SAR instruction shifts the signed byte or word right by the number of bits specified in the count. As the bits are shifted in right direction (⟶), LSB bit goes to CF & in the vacancy created in MSB, same MSB bit is retained.

The operation of SAR instruction is as follows



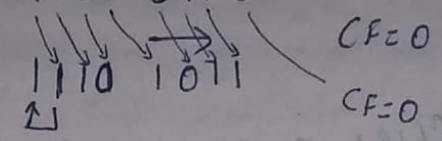
General form

SAR dest, count

The destination can be a byte or word operand. It can be a register or memory variable. If count=1, then it may be specified directly in the instruction. If count>1, then it must be stored in CL register before using of SAR instruction.

Eg:- SAR AL, 01

Before : AL = 1101 0110

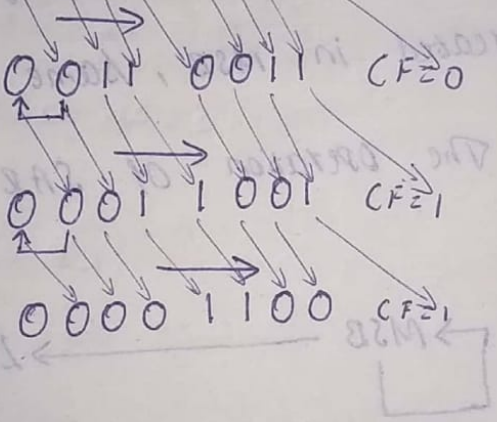


After: 1110 1011

2) MOV CL, 03
SAR AL, CL

Before

AL = 0110 0110



4) SAL : Shift Arithmetic Left

SAL instruction is same as SHL, shifts the destination byte or word operand left by the number of bits specified in the count. As the bits are shifted in left direction (←), MSB bit goes to CF & in the vacancy created in LSB, 0 will be inserted

Rotate Instruction

Rotate instruction rotates the destination byte or word operand either to the right or left by the specified number of bits.

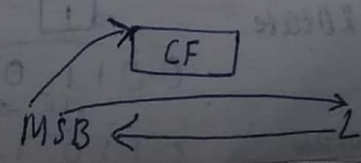
8086 Processor supports 4 types of rotate instructions

1) ROL (Rotate Left)
2) ROR (Rotate Right)
3) RCL (Rotate through carry Left)
4) RCR (Rotate through carry Right)

1) ROL: Rotate Left

The ROL instruction rotates the destination byte or word operand left by the number of bits specified in the count. As the bits are rotated in left direction (←), MSB bit goes to CF as well as to the vacancy created in LSB bit position.

The operation performed by ROL instruction is as shown below



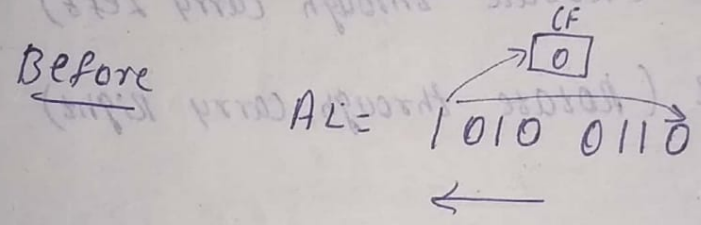
General form:

ROL dest, count

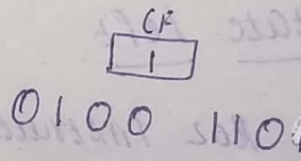
The destination can be byte or word operand. It may be a register or memory variable. If count=1, then it may be specified directly in the instruction. If count >1, then it must be stored in CL register before using ROL instruction.

Eg:- 1) ROL AL, 01

Before

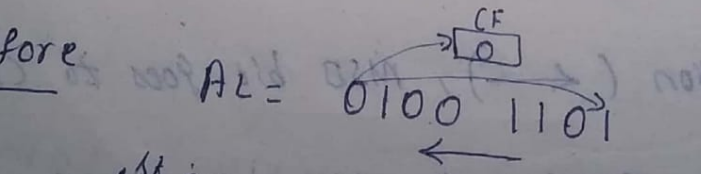


After

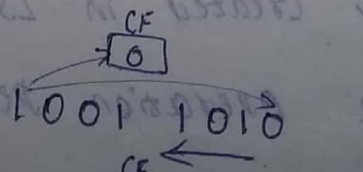


Eg: 2) MOV CL, 03
ROL AL, CL

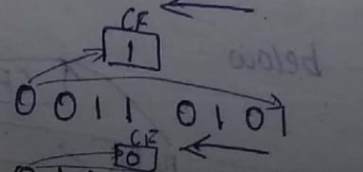
Before



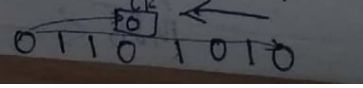
1st Rotate



2nd Rotate



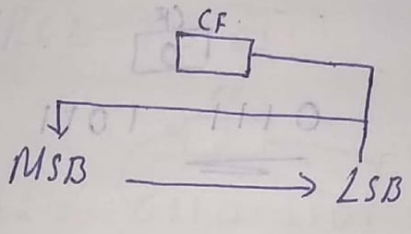
After: 3rd Rotate



2) ROR : Rotate Right

The ROR instruction rotates the destination byte or word operand right by the number of bits specified in the count. As the bits are rotated in right direction (→), LSB bit goes to CF as well as to the vacancy created in the MSB position.

The operation performed by ROR instruction is as follows

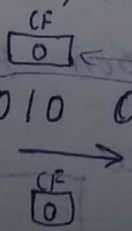


General form:

ROR dest, count

The destination can be either a byte or word operand. It may be either register or memory variable. If count = 1, then it may be specified directly in the instruction. If count > 1, then it must be stored in CL register before using ROR instruction

Eg:- ROR AL, 01

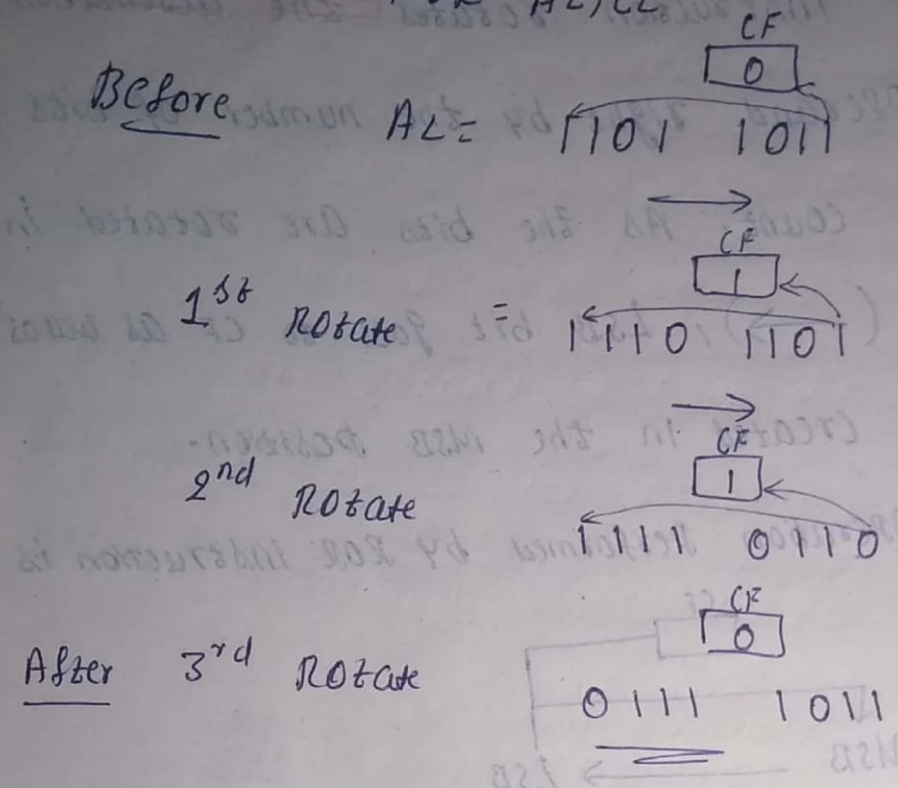


Before AL = 1010 0110

After 0101 0011

Eg: 2) MOV CL,03

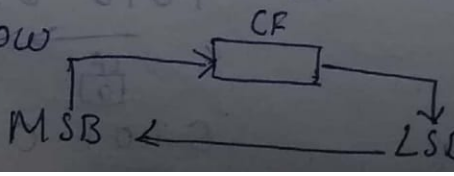
ROR AL,CL



3) RCL : Rotate through carry Left

RCL instruction rotates destination byte or word operand left through carry by the number of bits specified in the count. As the bits are rotated in left direction (←), MSB bit goes to CF & previous CF content are placed in the vacancy created in LSB position

The operation performed by RCL instruction is as shown below



General form

RCL dest, count

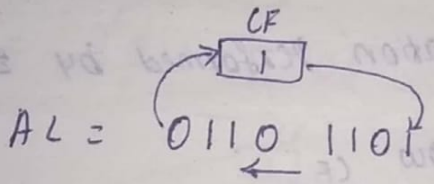
destination can be either byte or word operand. It may be register or memory variable. If count=1, then it may be specified directly in the instruction. If count > 1, then it must be stored in CL register before using of RCL instruction.

Rajeshwari. B.S
Assistant Professor
CSE Dept
BMSCE, Bangalore

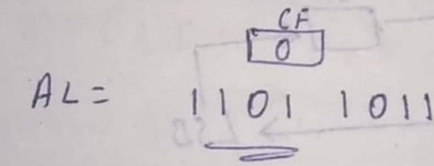
Eg:

1) RCL AL, 01

Before

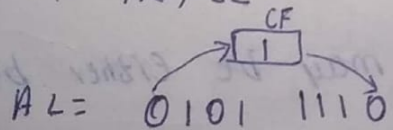


After

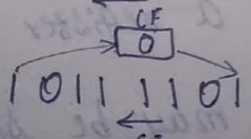


2) MOV CL, 03
RCL AL, CL

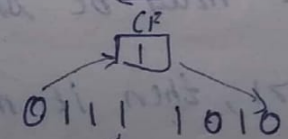
Before



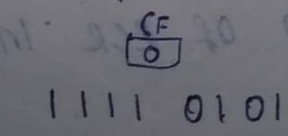
1st Rotate



2nd Rotate



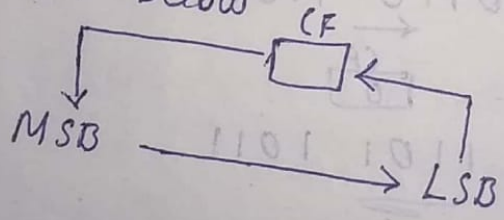
After: 3rd Rotate



4) RCR : Rotate through Carry Right

The RCR instruction rotates the destination byte or word operand right through carry bit by the number of bits specified in the count. As the bits are rotated in right direction (→), LSB bit goes to CF & previous CF content are placed in the vacancy created in MSB position.

The operation performed by the RCR instruction is as shown below

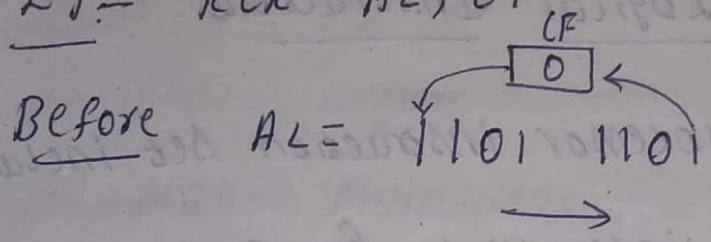


General form

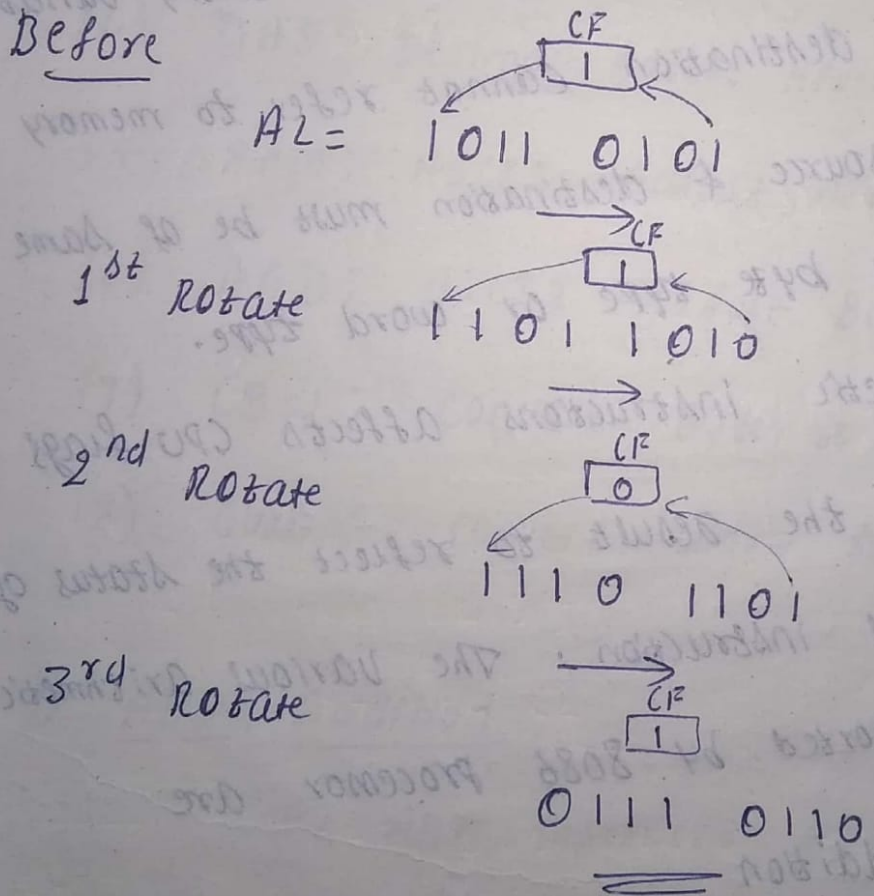
RCR dest, count

The destination may be either byte or word operand. It may be a register or memory variable. If count = 1, then it may be specified directly in the instruction. If count > 1, then it must be stored in CL register before using of RCR instruction.

Eg:- RCR AL, 01



2) MOV CL, 03
RCR AL, CL



Arithmetic & Logical Instructions

The 8086 processor instruction set includes various arithmetic instructions for performing arithmetic operations. Arithmetic instruction generally involves two operands source & destination. The source may be immediate value, or register or memory variable. The destination may be either register or memory variable. Both source & destination cannot refer to memory location. Both source & destination must be of same data type esther byte type or word type.

Arithmetic instructions affects CPU flags depending upon the result to reflect the status of recently executed instruction. The various arithmetic instructions supported by 8086 processor are

1) ADD : Addition
2) ADC : Add with carry
3) SUB : Subtraction
4) SBB : Subtract with Borrow
5) INC : Increment
6) DEC : Decrement

7) CMP : Compare

8) NEG : Negate

9) MUL : Multiplication

10) DIV : Division

11) AAA : ASCII Adjust After Addition

12) DAA : Decimal Adjust After Addition

13) AAS : ASCII Adjust After Subtraction

14) DAS : Decimal Adjust After Subtraction

15) AAM : ASCII Adjust After Multiplication

16) AAD : ASCII Adjust Before Division

17) CBW : Convert Byte to WORD

18) CWD : Convert word to Double word.

1) ADD : Addition

The ADD instruction adds the contents of the source to the destination & stores the result in destination. The source can be a immediate value or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory variables. Both soure &

Destination must be of same type either byte type or word type. If word + byte is to be added, then byte should be converted into word by filling upper bytes with zero before performing addition operation.

General form:

ADD dest, src ; dest ← dest + src

Flags affected

CF, PF, ZF, AF, SF, OF

Eg: 1) ADD AL, BL

16 | 110
6 - E

If AL = 110 = 0110 1110
BL = 85 = 0101 0101
1100 0011

CF = 0 AF = 1
PF = 1 SF = 1
ZF = 0 OF = 0 (Always OF = 0 for unsigned operation)

2) ADD AL, BL

If AL = -83 = 1101 0011
BL = -85 = 1101 0101
1010 1000

-83
-85
168

32
8
40

Sign bit = -

= -40 Invalid answer

CF = 1, OF = 1, ZF = 0 PF = 0, SF = 1, AC = 0

∴ 6th bit is producing carry to 7th bit, which is overwriting sign bit

Maximum value can be store is -127, but here -168

ADD AL, 12H ; Valid

ADD AL, BL ; Valid

ADD AL, A ; Valid

ADD A, BL ; Valid

ADD A, B ; Invalid, Both src & dest cannot refer to memory location

ADD AX, B ; Invalid, Data type must match

ADD 12, BL ; Invalid destination need not be immediate value.

2) ADC: Add with carry

The ADC instruction adds the content of source to the destination along with the carry flag content & result will be stored back to destination. The source can be immediate value, or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory location. Both source & destination must be of same data type either byte type or word type. If a byte & word is to be added, then byte should be converted in word by filling upper byte with zero before performing addition.

General form

ADC dest, src ; dest ← dest + src + CF

Flags affected : CF, ZF, AF, PF, OF, SF

Eg:- 1) ADC AL, BL

If AL = 0101 0101 = 85

BL = 0110 1101 = 109

CF = 1

1100 0011 = 195

CF = 0, SF = 1, ZF = 0, AF = 1, PF = 1, OF = 0 (Always OF = 0 for unsigned operation)

2) ADC AL, BL

If AL = 0000 1101 = 13

BL = 0000 1101 = 13

CF = 0 26

0001 1010 = 26

CF = 0, SF = 0, AF = 1, PF = 0, ZF = 0, OF = 0 (Always OF = 0 for unsigned operation)

3) SUB : Subtraction

The SUB instructions subtracts source from the destination & places the result in the destination. The source can be a register or immediate value or memory variable. The destination can be a

register or memory variable. Both source & destination cannot refer to memory locations. Both source & destination must be of same data type either byte type or word type. If a byte is to be subtracted from word, then byte should be converted into word by filling the upper bytes with zeros before performing subtraction operation.

General form: SUB dest, src ; dest ← dest - src

Flags affected: CF, PF, OF, SF, AF, ZF

Eg: 1) SUB AL, BL

If AL = 0110 1110 = 110 (decimal)
BL = 0101 0101 = 85 (decimal)
0001 1001 = 25 (decimal)

CF=0, SF=0, OF=0, PF=0, ZF=0, AF=0

2) SUB AL, BL

AL = 0101 0011 = 83
BL = 0101 0101 = 85
1111 1110 = FE
CF=1

83
-85
-2

FF = -1
FE = -1

Since CF=1 means -ve & FE = -2

-2

4) SBB : Subtract with Borrow

SBB instruction subtracts source from the destination & carry flag content are subtracted from the result & places the result in the destination. The source can be a immediate value or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory locations. Both source & destination must be of same data type either byte type or word type. If a byte is to be subtracted from a word, then the byte should be converted into word by filling upper byte with zero before subtraction operation

General form:

SBB dest, src ; dest ← dest - src - CF

Flags affected : CF, SF, PF, AF, ZF, OF

Eg:- SBB AL, BL

If AL = 0110 1110 = 110
BL = 0101 0101 = 85
0001 1001
CF = 1
0001 1000 = 24//

CF=0 AF=0
SF=0 OF=0
PF=1 ZF=0

SBB AL, BL

If AL = 1000 0101 = −5 (sign bit)

BL = 1000 0111 = −7 (sign bit)

1111 1110

Wrong answer ∵ 6th bit is taking borrow from 7th bit, which is a sign bit

5) INC : Increment

INC instruction adds one to the specified register or memory variable & stores the result back to destination. CF will not be affected by the execution of this instruction.

General form:

INC dest ; dest ← dest + 1

Flags affected:

AF, OF, SF, PF, ZF

Eg: 1) INC AL

If AL = 1000 0000 = 128

1

1000 0001 = 129

OF=0 AF=0, SF=1, PF=1, ZF=0

2) INC AL

If AL = 1111 1111 = FF

1

0000 0000

AF=1, PF=1, SF=0, ZF=1, OF=0

FF + 1 = 00

FFFF + 1 = 0000

3) INC AMOUNT[BX]; BX reg contains index of an array. AMOUNT[BX] contents will be incremented by 1

4) INC BYTE PTR[BX]; Increments byte pointed by BX register by 1

5) INC WORD PTR[BX]; Increments word pointed by BX register by 1

6) DEC : Decrement

The DEC instruction decrements the specified register content or memory variable content by 1 & stores the result back to destination. CF will not be affected by the execution of this instruction.

General form :

DEC dest; dest ← dest - 1

Flags affected

AF, PF, SF, OF, ZF

Eg: DEC AL

If AL = 1000 0000 = 128

$$\begin{array}{r} 1000\ 0000 \\ -\ 0000\ 0001 \\ \hline 0111\ 1111 = 127 \end{array}$$

OF=0, SF=0, PF=0, AF=1, ZF=0

DEC AL

If AL = 0000 0000

− 1 = 0000 0001

1111 1111

OF = 0, SF = 1, PF = 1, AF = 1, ZF = 0

$\left\{\begin{array}{r} 00 \\ -\ 1 \\ \hline FF \end{array}\right\}$ $\left\{\begin{array}{r} 0000 \\ -\ 1 \\ \hline FFFF \end{array}\right\}$

7) CMP : Compare

The CMP instruction compares a byte or word specified in the destination with a byte or word specified in the source. The source can be immediate value or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory locations. Both source & destination must be of same type either byte type or word type.

The CMP instruction performs non destructive subtraction operation ie, it performs subtraction of source from destination & affects the flags depending upon the result, but result will not be stored back to destination.

General form

CMP dest, src

1) If (dest > Src)

CF = 0, ZF = 0, SF = 0

2) If (dest < Src)

CF = 1, ZF = 0, SF = 1

3) If (dest = Src)

CF = 0, ZF = 1, SF = 0

Eg:-

1) CMP AL, BL

If AL = 0000 1001 = 9

BL = 0000 1000 = 8 AL > BL

0000 0001 CF = 0, ZF = 0, SF = 0

2) CMP AL, 12

If AL = 10, then AL < 12

∴ CF = 1, SF = 1, ZF = 0

3) CMP AL, A

If AL = 10 + A = 10, then AL = A

CF = 0, ZF = 1, SF =

8) NEG : Negate

The NEG instruction replaces a byte or word in the destination by its 2's complement. The destination can be a register or memory location.

General form:

NEG dest

Eg: 1) NEG AL

If AL = 1011 0110

After 0100 1001 (1's complement)

+ 1 (2's complement)

0100 1010

2) NEG BYTE PTR [BX];

replaces a byte point by BX registed by it 2's complement

7) MUL : Multiplication

King
PAGE NO:
DATE : / / 200

MUL instruction performs unsigned multiplication of source operand with the accumulator. The source can be a register or memory location. MUL instruction performs 2 types of multiplication

8 * 8 ⇒ 16 bit result

16 * 16 ⇒ 32 bit result

If the source is a byte, then it is multiplied with AL register & 16 bit result will be stored in AX register with AH register contains high order byte of result & AL register contains low order byte of result.

If the source is a word, then it is multiplied with AX register & 32 bit result will be stored in DX: AX register with high order word of result stored in DX register & low order word of result stored in AX register.

If a byte & word is to be multiplied, then the byte should be converted into word by filling upper byte with zeros.

Functions

if (byte source) then

Source * AL ⇒ AX

If (word source)

Source * AX ⇒ DX: AX

General form

MUL SRC ; AL * SRC ⇒ AX

AX * SRC ⇒ DX : AX

Flags affected : CF, OF

Eg:- 1) MUL BL

If BL = 07 = 0111

AL = 02 = 0010

Multiply each bit of BL with each bit AL

```
  0010
   0010
  0010
 0000
-------
00001110 = 0E (14 decimal)
```

2) MUL BX ; BX * AX ⇒ DX : AX

3) MUL BYTE PTR [BX] ; AL * [BX] ⇒ AX

byte pointed by BX reg

If a byte & word is to be multiplied, then byte should be converted into word by filling upper byte with zeros

```
MOV AX,1234
MOV CL,56        AX = 1234
MOV CH,00        CX = 0056
MUL CX           AX * CX ⇒ DX : AX
```

10) DIV: Division

DIV instruction performs unsigned division of accumulator by source operand. The source can be a register or memory location. DIV instruction performs 2 types of Division

1) 16/8 ⇒ 8 bit Quotient = AL, 8 bit Remainder = AH

2) 32/16 ⇒ 16 bit Quotient = AX, 16 bit remainder = DX

When a word is to be divided by a byte, then dividend word must be in AX register & 8 bit divisor may be in any of 8 bit register or memory variable, after performing division, 8 bit Quotient will be stored in AL register and 8 bit remainder will be stored in AH register.

When a double word is to be divided by a word, then 32 bit double word dividend must be in DX: AX with high order word in DX & low order word in AX & 16 bit divisor may be in any of 16 bit register or memory variable. After performing division, 16 bit Quotient will be stored in AX & 16 bit remainder will be stored in DX register.

If a byte is to be divided by a byte, then the dividend byte should be converted into word by filling upper byte with zero

General form

DIV src ; AX / byte src
Quotient = AL
Remainder = AH

DX : AX
word src
Quotient = AX
Remainder = DX

Flags affected

AF, CF, OF, ZF, SF, PF

Eg:-

1) DIV CL

If AX = 1234
CL = 50
AL = 246 (decimal)
= F6 (Hexa decimal)
AH = 04

16 | 246
15 - 6

2) DIV CX ; DX : AX / CX
AX → Quotient
DX → Remainder

3) DIV A ; If A is byte variable
AX / A
AL → Quotient
AH → Remainder

9) AAA : ASCII Adjust after Addition

AAA instruction converts the hexadecimal result in AL register after performing addition of 2 ASCII data to unpacked BCD result. This instruction should be executed immediately after the ADD instruction that adds 2 ASCII data.

The data that we enter from the keyboard will be stored in an ASCII format. for eg: If we enter values like 9 and 8, internally inside the computer it will be stored as 39H (ASCII code of 9) and 38H (ASCII code of 8). If we add that 2 numbers using the instruction ADD AL, BL, the result in AL register will be 71H ie,

$$\begin{array}{r} 0011\ 1001 = 39H \\ 0011\ 1000 = 38H \\ \hline 0111\ 0001 = 71H \end{array}$$

, but the result must be 17 (9+8=17). So we need to use AAA instruction immediately after ADD instruction, which converts the result in AL register 71H into unpacked BCD value 0107.

The working of AAA instruction is as follows

1) Clear the high order nibble of AL (AL ← AL & 0F)

2) If lower nibble of AL > 9 or if AF = 1, then

   a) Add 6 to AL (ie, AL ← AL + 6)

   b) Add 1 to AH

   c) Clear high order nibble (ie, AL ← AL & 0FH)

For eg:- If result in AL = 39H + 38H = 71H

1) 71H = 71 & 0F = 01

2) AF = 1 during addition a) ∴ 01 + 6 = 07 ⇒ AL

& b) Add 1 to AH ie, AH = 00 + 01 ⇒ 01

c) AL ← AL & 0F ie, 07 & 0F = 07 ⇒ AL

∴ AX = 0107 (unpacked result 17)

Eg: 2

```
MOV AH, 01   } Read value
INT 21H      }   for eg:- 7 enters reads as 37
MOV BL, AL       in AL register

MOV AH, 01   } Read second value
INT 21H      }   for eg:- 8 enters reads as 38 in AL
MOV AH, 00
ADD AL, BL   }  0011 0111 = 37H
                0011 1000 = 38H
                0110 1111 = 6FH

AAA          }  AL ← 6F & 0F ⇒ 0F > 9
```

a) ∴ 0F + 6 = 15

b) AH ← AH + 1 = 00 + 1 = 01

c) AL ← AL & 0F = 15 & 0F = 05

∴ AX = 0105 (unpacked 15)

Eg:3

```
MOV AL, 07H
MOV BL, 08H
MOV AH, 00
ADD AL, BL      ; AL ← 0FH
AAA             ; AX: 0105
```

12) DAA : Decimal Adjust after Addition

The DAA instruction convert the result in AL register into packed decimal data. This instruction should be executed immediately after the ADD instruction that adds 2 decimal data.

The working of DAA instruction is as follows

1) if (lower nibble of AL > 9 or AF = 1) then
   AL ← AL + 6, AF ← 1

2) if (Upper nibble of AL > 9 or CF = 1) then
   AL ← AL + 60 ; CF ← 1

For eg:

```
MOV AL, 45H
MOV BL, 38H
ADD AL, BL
DAA             ; AL ← 7D
```

0100 0101 ⇒ 45H
0011 1000 ⇒ 38H
0111 1101 ⇒ 7D
with CF=0 & AF=0
lower nibble D > 9 ∴ 7D + 6

```
  0111 1101   7D
+ 0000 0110  + 06
-------------------
  1000 0011   83
```

∴ AL = 83

higher nibble 8 < 9 ∴ no addition with 60

2)
```
MOV AL, 07
MOV BL, 08
ADD AL, BL
DAA
```

```
  0000 0111
+ 0000 1000
-----------
  0000 1111
```

with CF=0, AF=0

AL ← 07 + 08 = 0F

AL ← 15

3) If AL = 63H, BL = 88H

```
  0110 0011
+ 1000 1000
-----------
  1110 1011  = EBH ⇒ AL
```

with CF=0 & AF=0

AL = EB

lower nibble B > 9 ∴

```
  EB
+ 06
----
  F1
```

& makes AF=1

higher nibble F > 9 ∴

```
  F1
+ 60
-----
CF=1 51
```

& makes CF=1

∴ result = 63 + 88 = 151

4) If AL = 53D, BL = 36D

```
  0101 0011
+ 0011 0110
-----------
  1.000 1001  = 89H
```

with CF=0 & AF=0

Since result = 89

9 <= 9 & AF=0 ∴ no addition 6 to lower nibble

8 <= 9 & CF=0 ∴ no addition 6 to higher nibble

5) AL = 99 1001 1001
BL = 88 1000 1000
0010 0001 = 21

With CF=1 & AF=1

Since AF=1, Add 06 ∴ 21
06
27 & makes AF=1

Since CF=1 Add 60 ∴ 27
60
87 & makes CF=1

∴ CF=1 AL=87 ∴ result = 187

6) AL = 63D 0110 0011
BL = 42D 0100 0010
1010 0101 = A5 with CF=0 & AF=0

A5

lower nibble 5 < 9 & AF=0, ∴ no addition

higher nibble A > 9 ∴ A5
60
105 & makes CF=1

∴ CF=1 & AL=05 ∴ result = 105

13) AAS: ASCII Adjust After Subtraction

The AAS instruction converts the hexadecimal result in AL register after performing subtraction of 2 ASCII data into unpacked BCD result. This instruction should be executed immediately after SUB instruction that subtracts 2 ASCII data & stores the result in AL register

Syntax: AAS

The working of AAS instruction is as follows

1) Clear high order nibble of AL (AL ← AL & 0F)

2) if (lower nibble of AL > 9 or AF = 1) then

   a) Subtract 6 from AL (AL ← AL - 6)

   b) Clear high order nibble of AL (AL ← AL & 0F)

   c) Subtract 1 from AH (AH ← AH - 1)

   d) Set CF & AF (AF = CF = 1)

Eg:- 1) If AL = 38H (ASCII code of 8)

BL = 33H (ASCII code of 3)

AH = 00H

SUB AL, BL

AL = 0011 1000 = 38

BL = 0011 0011 = 33

AL = 0000 0101 = 05

AL = 05.

1) AL ← AL & 0F = 05 & 0F = 05

AH	AL
00	05

2) If AL = 35H (ASCII code of 5)
BL = 38H (ASCII code of 8)
AH = 00

SUB AL, BL
AAS

```
AL = 0011 0101 = 35 (ASCII code of 5)
BL = 0011 1000 = 38 (ASCII code of 8)
CY=1
-------------------
AL = 1111 1101
```

AL = FD

AAS Instruction performs following operation of AL register

I AL ← AL + 0F
← FD + 0F
← 0D

```
1111 1101
0000 1111
---------
0000 1101
```

II if lower nibble of AL > 09 or AF=1
Then

1) Subtract 6 from AL

```
  0D      0000 1101
- 06      0000 0110
----      ---------
  07      0000 0111
```

2) Clear higher nibble of AL

AL ← 07 + 0F = 07

3) Subtract 1 from AH

AH = 00 - 1 = FF

4) Set CF=1 AF=1 ∵ CF=1, means result is -ve

Result: AH FF AL 07 07 is the 10's complement result of 03

14) DAS : Decimal Adjust After Subtraction

DAS instruction converts hexadecimal result in AL register into its equivalent Packed BCD result. This DAS instruction should be used immediately after SUB instruction that subtracts 2 decimal numbers.

Syntax: DAS

Working of DAS instruction is as follows

1) if (lower nibble of AL > 9 or AF=1)

AL ← AL − 06 ; AF=1

2) if (higher nibble of AL > 09 or CF=1)

AL ← AL − 60 ; CF=1

Eg:-

(1) If AL = 45H

BL = 38H

SUB AL, BL

DAS

AL = 0100 0101 = 45H

BL = 0011 1000 = 38H

0000 1101 = 0DH

AL = 0DH

if (lower nibble of AL D > 9)

AL ← AL − 06

0000 1101
0000 0110
0000 0111 = 07 AF=1

higher nibble 0 ≠ 9 or CF ≠ 1

∴ AL = result 07

2) AL = 63H

BL = 88H

SUB AL,BL 0110 0011 = 63H

DAS (CY=1) 1000 1000 = 88H

AL = 1101 1011 = DBH

AL = DB

1) lower nibble B > 09

AL = DB − 06 1101 1011
0000 0110
= D5 & Set AF=1 1101 0101

2) higher nibble D > 9

AL = D5 − 60 1101 0101
0110 0000
= 75 & Set CF=1 0111 0101

75 is 10's complement of 25 (63−88)

∴ CF=1, result will be taken as −ve

−25

3) AL = 68H

BL = 85

SUB AL,BL

DAS

```
  0110 1000 = 68H
(CF=1) 1000 0101 = 85H
-------------------
  1110 0011

  E3
```

1) lower nibble $3 \neq 9$ or $AF \neq 1$

2) higher nibble $E > 9$

$AL \leftarrow E3 - 60$ [1110 0011
0110 0000
1000 0011

$= 83H$ & set CF=1

AL = result 83, which is 10's complement of 17 (68-85)

Since CF=1, result will be -ve -17

15) AAM: ASCII Adjust After Multiplication.

AAM instruction converts the hexadecimal result in AL register after performing multiplication of 2 unpacked BCD numbers into unpacked BCD result. This instruction must be executed immediately after the MUL instruction.

Syntax: AAM

Eg:- MOV AL, 09H

MOV BL, 07H

MUL BL ; AX ← AL * BL = 09 * 07 = 003F

AAM

AX: 003F

1001
0111
0111
0000
0000
0111 = 3F

After execution of AAM instruction, result 003F will be converted into unpacked BCD result

AH AL
06 03 (7*9 = 63)

16) AAD: ASCII Adjust Before Division

AAD instruction converts the unpacked BCD values in AH & AL registers into its equivalent hexadecimal values & places in AL register. This instruction should be executed before DIV instruction.

Syntax: AAD

for eg: To divide 62/8

MOV AH, 06 ; Unpacked BCD 6

MOV AL, 02 ; Unpacked BCD 2

MOV CL, 08

AAD ; Converts AH AL 0602 ⇒ AH AL 003E (Hexadecimal equivalent of 62)

DIV CL ; 3EH/08 ⇒ 62/08 ⇒ AL = 07 (Quotient) AH = 06 (Remainder)

Eg: 2 To divide 25/02

```
MOV AH,02 ; unpacked BCD of 2
MOV AL,05 ; unpacked BCD of 5
MOV CL,02 ; Divisor
AAD       ; Converts AH AL      AH AL
                     02 05 ⇒ 00 19H (Hexadecimal equal of 25)
DIV CL; 19H/02 ⇒ 25/02 ⇒ AL=0C (Quotient)
                         AH=01 (Remainder)
```

17) CBW: Convert Byte to word

The CBW instruction copies the extended sign of a byte in AL register to all the bits in AH register. This instruction must be executed before a signed byte in AL is divided by another signed byte

Syntax: CBW

for eg: (1)

```
CBW
```

Before

AH= 10101010 AL= 10011011

After

AH= 11111111 AL= 10011011

(2) CBW

Before

AH= 1101 0101 AL= 0110 1111

After

AH= 0000 0000 AL= 0110 1111

18) CWD : Convert word to Double word

The CWD instruction copies the extended sign of a word in AX register to all the bits in DX register. This instruction must be executed before a signed word in AX is divided by another signed word.

Syntax: CWD

for eg: (1)

CWD

Before

DX: 1010 1010 0101 0101 AX: 1000 1010 1010 0101

After

DX: 1111 1111 1111 1111 AX: 1000 1010 1010 0101

(2) CWD

Before

DX: 0110 0101 1010 0011 AX: 0001 0010 0011 0100

After

DX: 0000 0000 0000 0000 AX: 0001 0010 0011 0100

Logical Instructions

The various logical instructions supported by 8086 processor are

1) AND: Logical AND
2) OR: Logical OR
3) NOT: Logical NOT
4) XOR: Logical Exclusive OR
5) TEST: Test & Set flags

1) AND: Logical AND

The AND instruction performs logical AND of 2 operands & stores the result back to destination. The source can be immediate value, or register or memory variable. The destination can be register or memory variable. Both source & destination cannot refer to memory locations. Both source & destination must be of same data type either byte or word type.

AND instruction follows AND truth table

I/P1	I/P2	O/P
0	0	0
0	1	0
1	0	0
1	1	1

Syntax: AND dest, src ; dest ← dest & src.

Flags affected: CF, OF, PF, SF, ZF

Eg:- AND AL, BL

if AL = 1011 0101

BL = 0110 0101

AL = 0010 0101 CF=0, SF=0, PF=0, ZF=0, OF=0

2) OR: Logical OR

The OR instruction performs logical OR of 2 operands & stores the result back to destination. The source can be immediate value, or register or memory variable. The destination can be register or memory variable. Both source & destination cannot refer to memory variable. Both source & destination must be of same data type either byte or word type

OR instruction follows OR truth table

I/p 1	I/p 2	O/p
0	0	0
0	1	1
1	0	1
1	1	1

Syntax: OR dest, src ; dest ← (dest | src)

Flags affected:- CF, SF, ZF, PF, OF

Eg:- OR AL, BL

If AL = 0100 1101
BL = 0011 1100
AL = 0111 1101

CF=0, OF=0, ZF=0, PF=1, SF=0

3) NOT: Logical NOT

NOT instruction inverts each bit of a byte or a word & places the result in destination.
ie, it finds 1's complement of a specified byte or word

Syntax: NOT dest ; dest ← ~ dest

Eg:-

NOT BL

if BL = 1011 0110

After

BL = 0100 1001

4) XOR: Logical Exclusive OR

XOR instruction performs exclusive OR of 2 Operands & stores the result back to destination.

The source can be a immediate value or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory location. Both source & destination must be of same type either byte or word type.

The XOR instruction follows XOR truth table

I/p 1	I/p 2	O/p
0	0	0
0	1	1
1	0	1
1	1	0

ie, If 2 bits are different, then resultant bit = 1
If 2 bits are same as 0,0 or 1,1, then resultant bit = 0

Syntax:

XOR dest, src : dest ← dest ^ src

Eg: XOR CL, 25H

If CL = 1011 0110

Immediate value = 0010 0101

1001 0011

CF = 0, OF = 0, SF = 1, PF = 1, ZF = 0

5) Test: Test & Set flags

Test instruction performs non destructive AND operation ie, It performs logical AND of source & destination & affects the flags depending upon the result, but the result will not be stored back to destination. neither of the operands are modified. The source can be immediate value or register or memory variable. The destination can be a register or memory variable. Both source & destination cannot refer to memory variables. Both source & destination must be of same type either byte or word.

Syntax: TEST dest, src ; Non destructive AND of src & dest
flags ← sets as results

Eg: TEST AL, BL

Before If AL = 1010 0110 = A6H
BL = 0110 1101 = 6DH
0010 0100

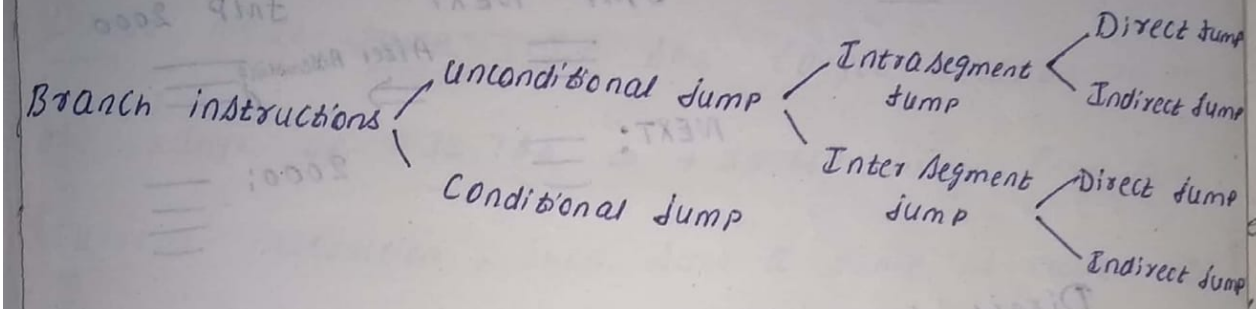
After

AL = A6H
BL = 6DH
flags: CF=0, OF=0, SF=0, ZF=0, PF=1

Branch Instructions

A branch instructions transfers the control from the normal sequence of instruction execution to a specified instruction.

Branch instructions are broadly classified as



Unconditional jump

Unconditional jump instruction transfers the control to the specified instruction without checking any condition.

Unconditional jump instruction supported by 8086 processor is JMP.

If a JMP instruction transfers the control to a specified instruction with in the same code segment, then such a JMP instruction is called as Intra segment jump or jump with in segment. or NEAR JUMP.

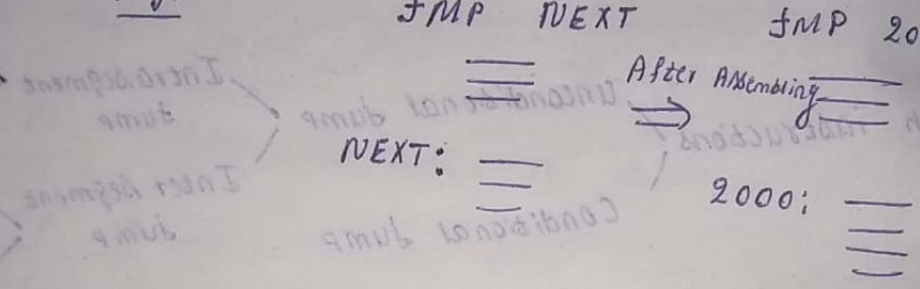
Intra segment jump instruction modifies only Instruction Pointer (IP) to transfer the control to

the specified instruction. Intra-Segment jump may be Direct jump or Indirect jump.

Direct jump

If the address of target location is specified directly in the jump instruction, such a jump instruction is called as Direct jump

Eg:-



Direct jump is of 2 types

1) Direct Short jump
2) Direct near jump

Direct Short jump:- If the target location is in the range of -128 to +127 bytes from the current instruction, then such a jump is called as Direct Short jump.

Eg:-

① JMP SHORT NEXT

NEXT:

JMP NEXT

OR

NEXT:

Eg: ②

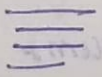
BACK:

JMP BACK

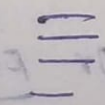
NEXT & BACK is a label to which branch has to takes place, whose displacement (distance) with respect to current JMP instruction is in the range of -128 to +127

Direct near jump:- If the target location is in the range of -32,768 to +32,767 bytes from the current instruction, then such a jump is called as Direct near jump.

Eg: ① JMP NEAR NEXT



NEXT:



OR

JMP NEXT

NEXT:

②

BACK:

JMP NEAR BACK

NEXT & BACK is a label to which branch has to takes place, whose displacement (distance) with respect to current JMP instruction is in the range of -32,768 to +32,767

Indirect Jump

If a 16 bit register or memory variable which contains address of target location is specified along with JMP instruction, then such a JMP is called as Indirect Jump.

Eg:

```
JMP BX ; IP ← BX
```

Inter-Segment Jump

If a JMP instruction transfers the control to the specified location in different code segment, such a Jump instruction is called as Inter-Segment Jump or FAR Jump.

Inter Segment Jump instructions modifies both CS register as well as IP to transfer the control to the specified location.

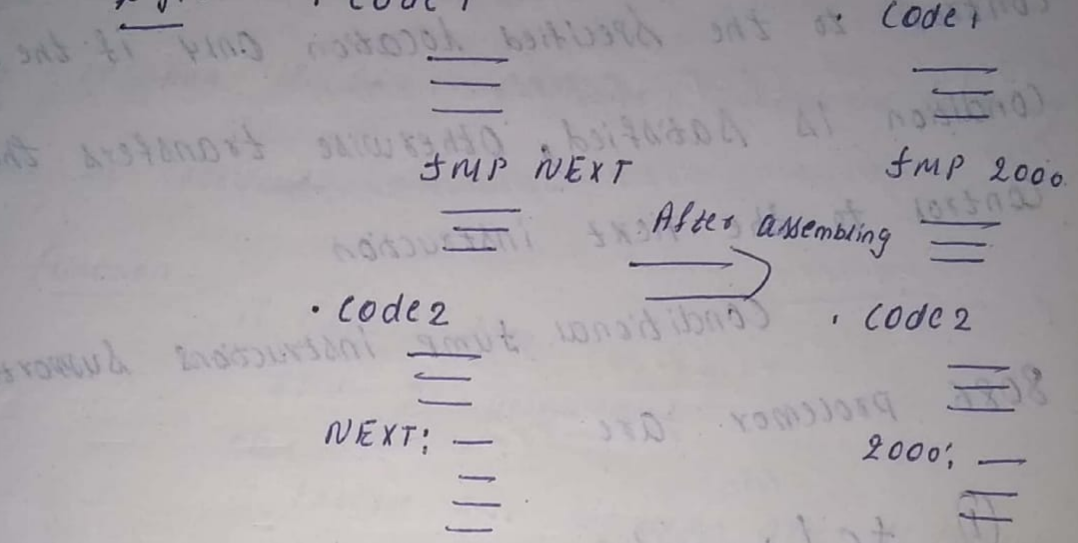
Inter-Segment Jump may be Direct FAR JMP or Indirect FAR JMP.

Direct FAR JMP

If the address of the target location

is specified directly in the JMP instruction itself, then it is called as Direct FAR JMP.

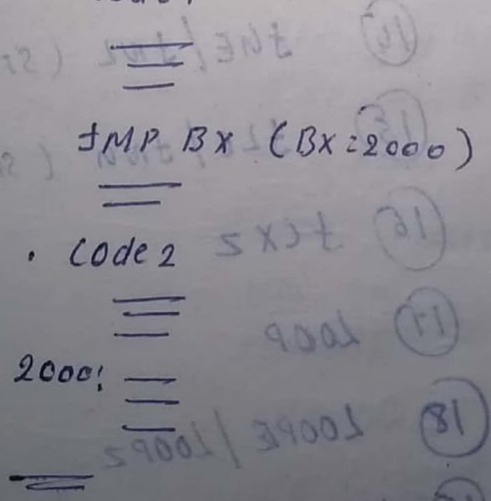
Eg:-



Indirect FAR JMP

If the register or memory variable, which contains address of target location is used in JMP instruction, then such a JMP is called as Indirect FAR JMP.

Eg:



Conditional Jump

Conditional jump instruction transfers the control to the specified location only if the specified condition is satisfied, otherwise transfers the control to the next instruction

Conditional jump instructions supported by 8086 processor are

1. JC / JNAE / JB :
2. JNC / JAE / JNB
3. JZ / JE
4. JNZ / JNE
5. JP / JPE
6. JNP / JPO
7. JS
8. JNS
9. JO
10. JNO
11. JA / JNBE (CF=0, ZF=0)
12. JG / JNLE (SF=OF & ZF=0)
13. JL / JNGE (SF≠OF)
14. JGE / JNL (SF=OF & ZF=1)
15. JLE / JNG (SF≠OF & ZF=1)
16. JCXZ
17. LOOP
18. LOOPE / LOOPZ
19. LOOPNE / LOOPNZ

1) JC / JNAE / JB : Jump if Carry / Jump if Not Above or Equal / Jump if Below.

PAGE NO:
DATE: / / 200

This instructions transfer the control to the target location, if CF = 1 Otherwise instructions following JC will be executed.

Function

```
if (CF=1) then
    IP ← IP + displacement
else
    Execute next instruction after JC instruction
```

Eg:-

Program to find smallest of 2 numbers

```
.Data
        A DB 35      MSG1 DB ODH,OAH, "First number is Smallest"
        B DB 26      msg2 DB ODH,OAH, "Second number is Smallest"
.Code
        MOV AX, @Data
        MOV DS, AX
        MOV AL, A
        CMP AL, B
        JC First
        LEA DX, MSG2
        MOV AH, 09H
        INT 21H
        JMP LAST
First : LEA DX, MSG1
        MOV AH, 09H
        INT 21H
LAST :  MOV AH, 4CH
        INT 21H
End
```

2) JNC / JAE / JNB : Jump if no carry / Jump if Above or Equal / Jump if not Below.

This instruction transfers the control to the target location if CF=0, Otherwise transfers the control to the next instruction after JNC instruction

Function

If (CF=0) then
    IP ← IP + displacement
else
    Execute next instruction after JNC instruction

Eg: Program to find Largest of 2 numbers

```
.Data
    A DB 55H
    B DB 43H
    msg1 DB 0DH, 0AH, "First number is largest $"
    msg2 DB 0DH, 0AH, "Second number is Largest $"
.Code
    mov AX, @Data
    mov DS, AX
    MOV AL, A
    CMP AL, B
    JNC First
    LEA DX, msg2
    MOV AH, 09H
    INT 21H
    JMP LAST
```

```
First: LEA DX, msg1
       MOV AH, 09H
       INT 21H
LAST:  MOV AH, 4CH
       INT 21H
       End
```

3) JZ/JE : Jump if zero / Jump if Equal

This instruction transfers the control to the target location, if ZF=1, Otherwise transfers the control to the next instruction after JZ instruction.

Function

```
If (ZF=1) then
    IP ← IP + displacement
else
    Execution next instruction after JZ instruction
```

Eg:

Pgm to check whether student has just passed the examination

```
.Data
    msg1 DB 0DH, 0AH, "Student just passed $"
    msg2 DB 0DH, 0AH, "Student has not just passed$"
    Marks DB 50
.Code
    MOV Ax, @data
    MOV Ds, Ax
    MOV AL, Marks
    CMP AL, 35
    JE just_pass
    LEA DX, msg2
    MOV AH, 09H
    INT 21H
```

```
        JMP LAST
SUBT_PASS: LEA DX, MSG1
        MOV AH, 09H
        INT 21H
LAST:   MOV AH, 4CH
        INT 21H
        End
```

4) JNZ / JNE : Jump if no zero / Jump if not Equal

This instruction transfers the control to the target location if ZF=0, Otherwise next instruction following JNZ instruction will be executed.

Function:

If (ZF=0) then

IP ← IP + displacement

Else

execute next instruction after JNZ instruction

Eg:

Pgm to check whether the result is zero or not after subtracting 2 numbers

```
. Data
        A   DB  55
        B   DB  25
        MSG1 DB ODH, OAH, "Result is Zero $"
        MSG2 DB ODH, OAH, "Result is Not Zero $"
```

· Code

```
        MOV AX, @Data
        MOV DS, AX
        MOV AL, A
        SUB AL, B
        JNZ NOT_ZERO
        LEA DX, MSG1
        MOV AH, 09H
        INT 21H
        JMP LAST
NOT_ZERO: LEA DX, MSG2
        MOV AH, 09H
        INT 21H
LAST:   MOV AH, 4CH
        INT 21H
        End
```

5) JP/JPE : Jump if parity / Jump if parity Even

This instruction transfers the control to the target location, if PF=1, Otherwise next instruction after JP instruction will be executed

Function

```
IF (PF=1) then
    IP ← IP + displacement
Else
    Execute next instruction after JP instruction
```

Eg: Pgm to Check whether result after performing Logical OR Operation is Even parity or Odd parity

```
.Data
        A DB 58H
        B DB 63H
        msg1 DB ODH, OAH, " Even parity $"
        msg2 DB ODH, OAH, " ODD Parity $"
.Code
        MOV AX, @Data
        MOV DS, AX
        MOV AL, A
        OR AL, B
        JP EVEN
        LEA DX, msg2
        MOV AH, 09H
        INT 21H
        JMP LAST
EVEN:   LEA DX, msg1
        MOV AH, 09H
        INT 21H
LAST:   MOV AH, 4CH
        INT 21H
        End
```

6) JNP / JPO : Jump if NO parity / Jump if parity ODD

This instruction transfers the control to the target location if PF=0. Otherwise instruction following JNP

instruction will be executed.

King
PAGE NO:
DATE: / / 200

Function

If (PF=0) then

IP ← IP + displacement

Else

Execute next instruction after JNP instruction

Eg: Pgm to check whether given number contains even number of One's or odd number of One's.

```
.Data
    A    DB 7DH
    msg1 DB ODH,0AH," Given number contains Even
                      number of 1's $"
    msg2 DB ODH,0AH," Given number contains ODD number
                      of 1's $"
.Code
        MOV AX,@Data
        MOV DS,AX
        MOV AL,A
        OR  AL,AL
        JPO ODD
        LEA DX,msg1
        MOV AH,09H
        INT 21H
        JMP LAST
ODD:    LEA DX,msg2
        MOV AH,09H
        INT 21H
LAST:   MOV AH,
        INT 21H
End
```

7) JS: Jump if Sign

This instruction transfers the control to the target location if SF=1, Otherwise next instruction after JS will be executed.

Function

```
If (SF=1) then
    IP ← IP + displacement
Else
    execute next instruction after JS instn
```

Eg:-

Pgm to Check whether the result after performing subtraction is negative result or positive result

```
.Data
    A   DB   25H
    B   DB   85H
    msg1 DB ODH, OAH, "Result is Positive$"
    msg2 DB ODH, OAH, "Result is negative$"
.Code
        MOV  AX, @Data
        MOV  DS, AX
        MOV  AL, A
        SUB  AL, B
        JS   NEGATIVE
        LEA  DX, msg1
        MOV  AH, 09H
        INT  21H
        JMP  LAST
```

8)

```
NEGATIVE: LEA DX, msg2
          MOV AH, 09H
          INT 21H
LAST:     MOV AH, 4CH
          INT 21H
          End
```

8) JNS : Jump if NO Sign

This instruction transfers the control to the target location if SF=0, Otherwise next instruction after JNS will be executed

Function

```
if (SF=0) then
    IP ← IP + displacement
Else
    next instruction after JNS will be executed.
```

Eg: Pgm to find sum of series 50+49+ ---- +1+0

```
.Data
    Res DW ?
.COde
    MOV AX, @Data
    MOV DS, AX
    MOV AX, 00
    MOV CL, 50
BACK: ADD AL, CL
NOCARRY: DEC CL        → JNC NOCARRY → INC AH
    JNS BACK
    MOV Res, AX
    MOV AH, 4CH
End INT 21H
```

9) JO! Jump if overflow

This instruction transfers the control to the target location if OF=1, Otherwise next instruction after JO instruction will be executed.

Function

if (OF=1) then