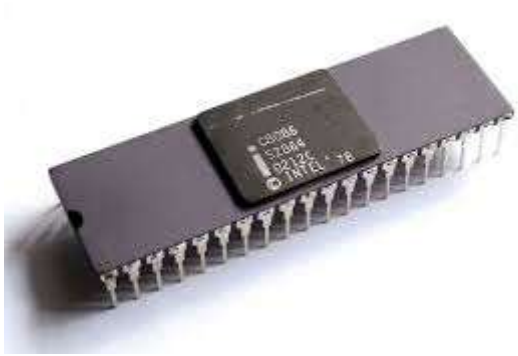
18ECC203J: Module – 2

Faculty of Engineering and Technology, SRM Institute of Science and Technology

Programming with Intel 8086



Organized by

Dr. K. A. Sunitha, Associate Professor, EIE, SRMIST

Dr. B. Ramakrishna, Assistant Professor, ECE, SRMIST

Picture Courtesy :Google

Session - 1

- Addressing modes of 8086

Assembly Instruction Format

General format

mnemonic	operand(s)	;comments

```
MOV        destination,source    ;copy source operand to destination
```

Example:

```
MOV    DX,CX
```

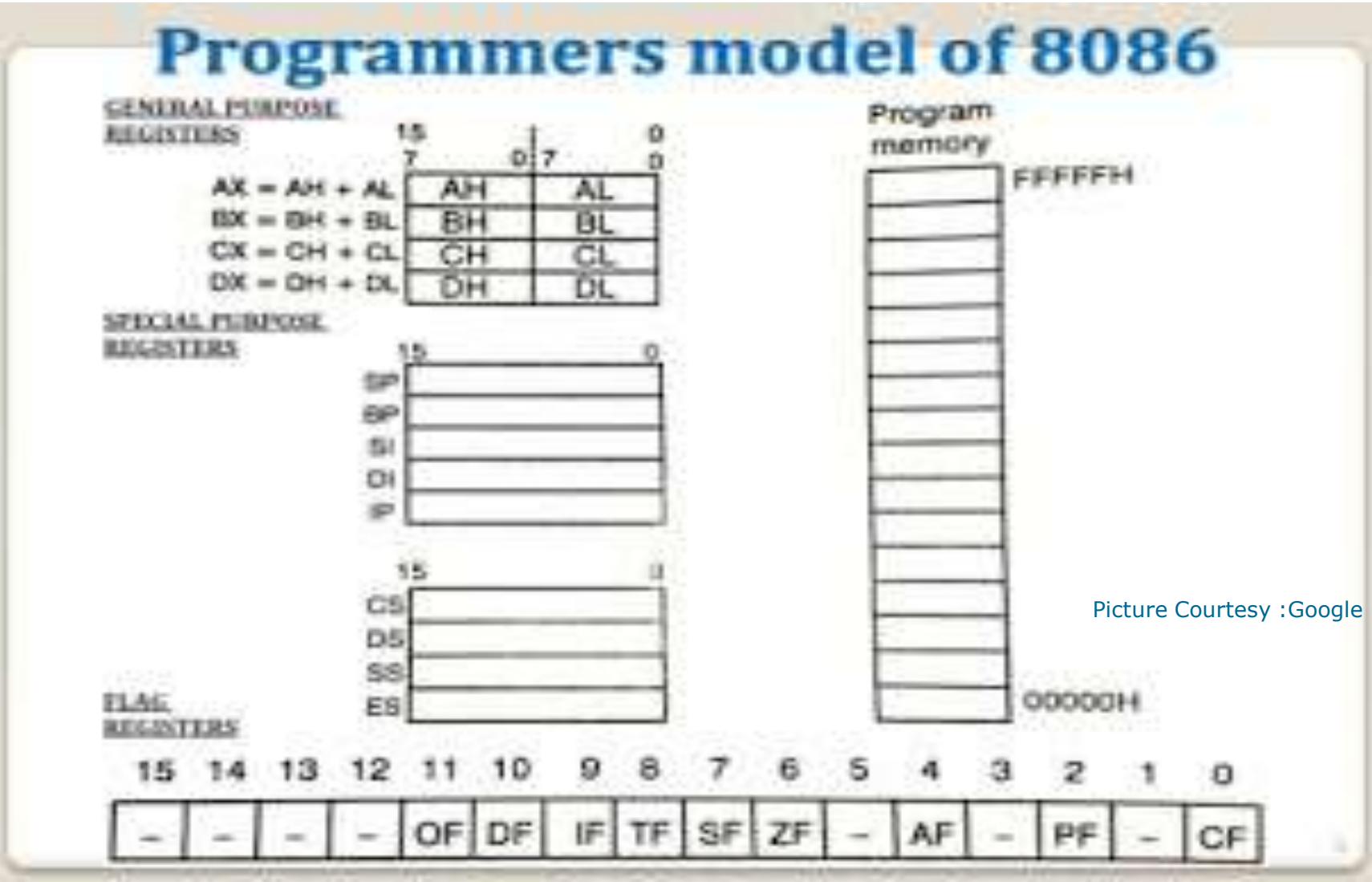
Example 2:

```
MOV    CL,55H
MOV    DL,CL
MOV    AH,DL
MOV    AL,AH
MOV    BH,CL
MOV    CH,BH
```

AH	AL
BH	BL
CH	CL
DH	DL

Picture Courtesy :Google

@ Students...Recall the registers of 8086



Picture Courtesy :Google

Addressing Modes in 8086

- Register addressing mode
- Immediate addressing mode
- Data memory addressing modes
  - Direct, Base, Index,
- Program memory addressing modes
- Stack memory addressing modes

Register Addressing Mode

- In this addressing mode, the data present in register is moved or manipulated and the result is stored in register.

Examples:

- MOV AL,BL ; Move content of BL to AL
- MOV CX,BX ; Move content of BX to CX
- ADD CL,BL ; Add content of CL and BL and store result in CL
- ADC BX,DX ; Add content of BX, carry flag and DX, and store result in BX

Immediate Addressing Mode

- In this mode, the data is directly given in the instruction.

Examples:

- MOV AL,50H ; Move data 50H to AL
- MOV BX,23A0H ; Move data 23A0H to BX
- MOV [SI],43C0H ; Move data 43C0H to memory at [SI]
- In the last example, [SI] represents the memory location in data segment at the offset address specified by SI register.

Data Memory Addressing Modes

- The term Effective Address (EA) represents the offset address of the data within a segment which is obtained by different methods, depending upon the addressing mode that is used in the instruction.
- Let us assume that the various registers in 8086 have the following values stored in them.

Direct Addressing Mode:

- In this mode, the 16 bit offset address of the data within the segment is directly given in the instruction.

Examples: @ DS = 3000H

- a) MOV AL, [1000H]

- EA is given within square bracket in the instruction in this addressing mode and hence EA=1000H in the above instruction. Since the destination is an 8-bit register (i.e. AL), a byte is taken from memory at the address given by DS x 10H + EA=31000H and stored in AL.

- b) MOV BX,[2000H]

- EA=2000H in this instruction. Since the destination is a 16 bit register (i.e. BX), a word is taken from memory address DSx10H+EA =32000H and stored in BX. (Note: Since a word contains two bytes, the byte present at memory address 32000H and 32001H are moved to BL and BH respectively.)

Base Addressing Mode

- In this mode, the EA is the content of BX or BP register. When BX register is present in the instruction, data is taken from the data segment and if BP is present in the instruction, data is taken from the stack segment.

Examples: @ DS = 3000H

- MOV CL,[BX] EA=(BX)=2000H
- Memory address=DSx10+(BX)=32000H. The byte from the memory address 32000H is read and stored in CL.
- @ SS = 3000H
- MOV DX,[BP] EA=(BP)=1000H
- Memory address=SSx10H+(BP)=41000H. The word from the memory address 41000H is read and stored in DX.

Base Relative Addressing Mode

- In this mode, the EA is obtained by adding the content of the base register with an 8-bit or 16 bit displacement. The displacement is a signed number with negative values represented in two's complement form. The 16 bit displacement can have value from -32768 to +32767 and 8 bit displacement can have the value from -128 to +127.

Examples: @ DS = 3000H

MOV AX,[BX+5] EA=(BX)+5

- Memory address=DSx10H+(BX)+5 = 30000H+2000H+5=32005H
- The word from the memory address 32005H is taken and stored in AX.
- MOV CH,[BX-100H] EA=(BX)-100H
- Memory address=DSx10H+(BX)-100H = 30000H+2000H-100H=31F00H
- The byte from the memory address 31F00H is taken and stored in CH.

Index Addressing Mode

- In this mode, the EA is the content of SI or DI register which is specified in the instruction. The data is taken from data segment.

Examples: @ DS = 3000H

- MOV BL,[SI] EA=(SI)=1000H
- Memory address=DSx10H+SI=30000H+1000H=31000H
- A byte from the memory address 31000H is taken and stored in BL.
- MOV CX,[DI] EA=(DI)=3000H
- Memory address=DSx10H+(DI)=30000H+3000H=33000H
- A word from the memory address 33000H is taken and stored in CX.

Index Relative Addressing Mode

- This mode is same as base relative addressing mode except that instead of BP or BX register, SI or DI register is used.

Example @ DS = 3000H

- MOV BX,[SI-100H] EA=(SI)-100H
- Memory Address=DSx10H+(SI)-100H = 30000H+1000H-100H =30F00H
- A word from the memory address 30F00H is taken and stored in BX.
- MOV CL,[DI+10H] EA=(DI)+10H
- Memory address=DSx10H+(DI)+10H =30000H+3000H+10H = 33010H
- A byte from the memory address 33010H is taken and stored in CL.

Base Plus Index Addressing Mode

- In this mode, the EA is obtained by adding the content of a base register and index register.

Example

- MOV AX,[BX+SI] EA=(BX)+(SI)
- Memory address=DSx10H+(BX)+(SI)=30000H+2000H+1000H =33000H
- A word from the memory address 33000H is taken and stored in AX.
- Base relative, index relative and base plus index addressing modes are used to access a byte or word type data from a table of data or an array of data stored in data segment one by one.

Base Relative Plus Index Addressing Mode

- In this mode, the EA is obtained by adding the content of a base register, an index and a displacement.

Example:

- MOV CX,[BX+SI+50H] EA= (BX)+(SI)+50H
- Memory address=DSx10H+(BX)+(SI)+50H
- =30000H+2000H+1000H+50H
- =33050H
- A word from the memory address 33050H is taken and stored in CX.

- Base relative plus index addressing is used to access a byte or a word in a particular record of a particular file in memory. A particular application program may process many files stored in the data segment.

- Each file contains many records and a record contains few bytes or words of data. In base relative plus index addressing, base register may be used to hold the offset address of a particular file in the data segment; index register may be used to hold the offset address of a particular record within that file and the relative value is used to indicate the offset address of particular byte or word within that record.

Program Memory Addressing Mode

- Program memory addressing modes are used with JMP and CALL instructions and consist of three distinct forms namely direct, relative and indirect.
- Direct Addressing: The direct program memory addressing stores both the segment and offset address where the control has to be transferred with the opcode.
- The above instruction is equivalent to JMP 32000H.
- When it is executed, the 16 bit offset value 2000H is loaded in IP register and the 16 bit segment value 3000H is loaded in CS.
- When the microprocessor calculates the memory address from where it has to fetch an instruction using the relation CSx10H+IP, the address 32000H will be obtained using the above CS and IP values.

Relative Addressing Mode

- The term relative here means relative to the instruction pointer (IP). Relative JMP and CALL instructions contain either an 8 bit or a 16 bit signed displacement that is added to the current instruction pointer and based on the new value of IP thus obtained, the address of the next instruction to be executed is calculated using the relation CSx10H+IP.
- The **8-bit or 16 bit signed displacement** which allows a forward memory reference or a reverse memory reference. A one byte displacement is used in short jump and call instructions, and a two byte displacement is used in near jump and call instructions. Both types are considered intersegment jumps since the program control is transferred anywhere within the current code segment.
- An 8 bit displacement has a jump range of between **+127 and -128** bytes from the next instruction while a 16 bit displacement has a jump range of between -32768 and +32767 bytes from the next instruction following the jump instruction in the program. The opcode of relative short jump and near jump instructions are EBH and E9H respectively.
- While using assembler to develop 8086 program, the assembler directive SHORT and NEAR PTR is used to indicate short jump and near jump instruction respectively.
- Examples:
- **a) JMP SHORT OVER**
- **b) JMP NEAR PTR FIND**
- In the above examples, OVER and FIND are the label of memory locations that are present in the same code segment in which the above instructions are present.

Indirect Addressing Mode

- The indirect jump or CALL instructions use either any 16 bit register **(AX,BX,CX,DX,SP,BP,SI or DI) or any relative register ([BP],[BX],[DI] or [SI]) or any relative register with displacement**. The opcode of indirect jump instruction is FFH. It can be either intersegment indirect jump or intersegment indirect jump instruction

- If a **16 bit register holds the jump address of in a indirect JMP instruction, the jump is near.** If the CX register contains 2000H and JMP CX instruction present in a code segment is executed, the microprocessor jumps to offset address 2000H in the current code segment to take the next instruction for execution (This is done by loading the IP with the content of CX without changing the content of CS).

- When the instruction JMP [DI] is executed, **the microprocessor first reads a word in the current data segment from the offset address specified by DI and puts that word in IP register.** Now using this new value of IP, 8086 calculates the address of the memory location where it has to jump using the relation CSx10H+IP.

Stack Memory Addressing Mode

- **The stack holds data temporarily and also stores return address for procedures and interrupt service routines**. The stack memory is a last-in, first-out **(LIFO) memory**. Data are placed into the stack using PUSH instruction and taken out from the stack using POP instruction. The CALL instruction uses the stack to hold the return address for procedures and RET instruction is used to remove return address from stack.

- The stack segment is maintained by two registers: the stack pointer (SP) and the stack segment register (SS). Always a word is entered into stack. Whenever a word of data is pushed into the stack, the higher-order 8 bits of the word are placed in the memory location specified by SP-1 (i.e. at **address SSx10H + SP-1)and** the lower-order 8 bits of the word are placed in the memory location specified by SP-2 in the current stack segment (SS) (i.e. at address **SSx10H + SP-2**). The SP is then decremented by 2. The data pushed into the stack may be either the content of a 16 bit register or segment register or 16 bit data in memory.

- Since SP gets decremented for every push operation, the stack segment is said to be growing downwards as for successive push operations, the data are stored in lower memory addresses in stack segment. Due to this, the **SP is initialized with highest offset address according to the requirement, at the beginning of the program**.

Segment Override Prefix

- The segment override prefix which can be added to almost any instruction in any memory related addressing mode, allows the programmer to deviate from the default segment and offset register mechanism. The segment override prefix is an additional byte that appears the front of an instruction to select an alternate segment register. **The jump and call instructions cannot be prefixed with the segment override prefix since they use only code segment register (CS) for address generation.**
- Example:
- MOV AX,[BP] instruction accesses data within stack segment by default since BP is the offset register for stack segment. But if the programmer wants to get data from data segment using BP as offset register in the above instruction, then the instruction is modified as
- **MOV AX, DS:[BP]**

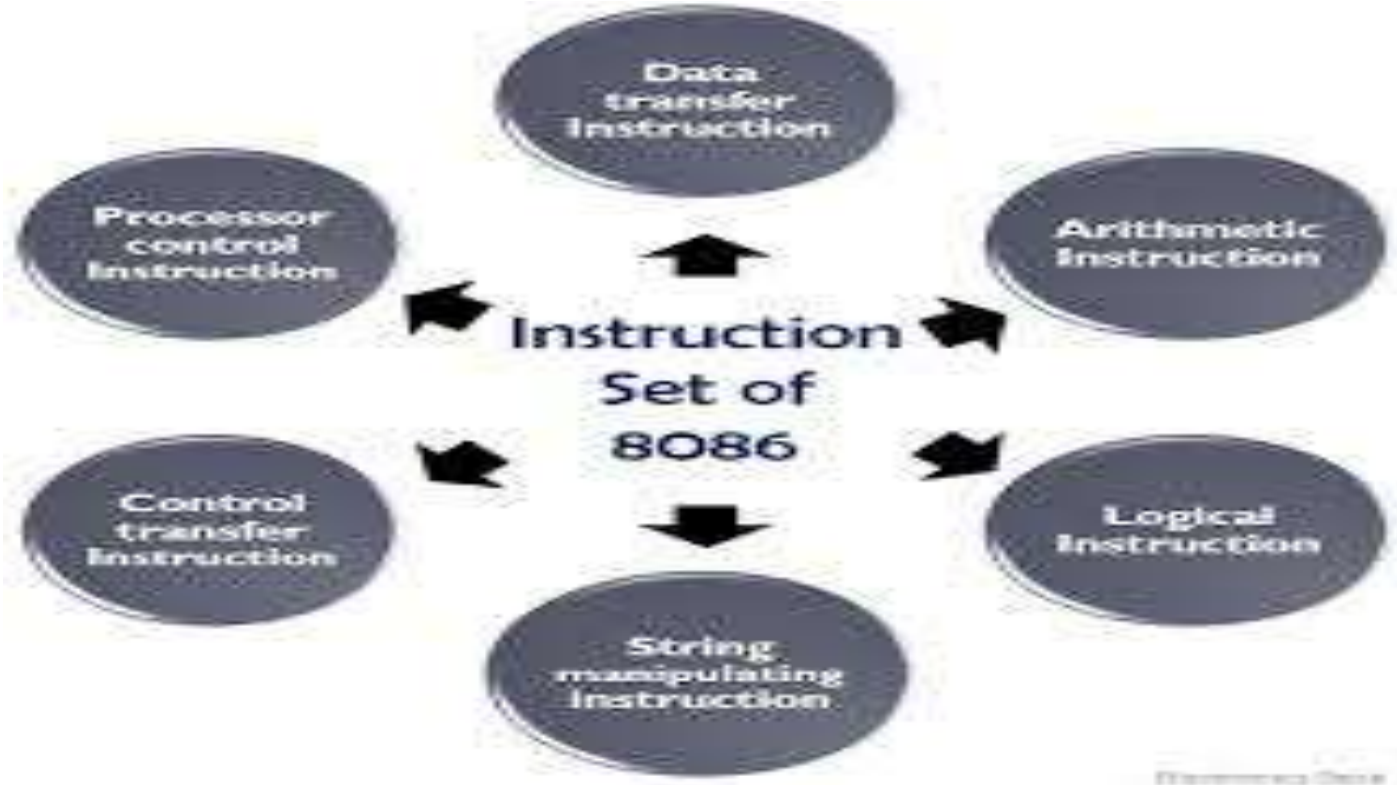
Session - 2

- Instruction Set of 8086: Data Transfer Instructions

&

- Example programs

Towards…..



Picture Courtesy :Google

Instruction Set Classification of 8086

- The instructions of 8086 are classified into
- Data transfer,
- Arithmetic,
- Logical,
- Flag manipulation,
- Machine control instructions
- Shift/rotate,
- String manipulation and

Data Transfer Instructions

The Data Transfer Instructions include

- MOV,
- PUSH, POP,
- XCHG, XLAT,
- IN, OUT,
- LEA, LDS, LES, LSS,
- LAHF and SAHF

Data Transfer Instructions

- MOV: MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register or a memory location or an immediate number. The general format of MOV instruction is

- **MOV Destination, Source**

Examples: MOV BL, 50H ; Move immediate data 50H to BL

- MOV CX, [BX] ; Copy word from memory at [BX] to CX

- MOV AX, CX ; Copy contents of CX to AX

- Note: [BX] indicates the memory location at offset address specified by BX in data segment.

What if ...

MOV AL,DX

Rule #1:

moving a value that is too large into a register will cause an error

```
MOV   BL,7F2H        ;Illegal: 7F2H is larger than 8 bits
MOV   AX,2FE456H     ;Illegal
```

Rule #2:

Data can be moved directly into nonsegment registers only

(Values cannot be loaded directly into any segment register. To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register.)

```
MOV   AX,2345H          MOV   DI,1400H
MOV   DS,AX             MOV   ES,DI
```

Rule #3:

If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

```
MOV BX, 5          BX =0005
                BH = 00, BL = 05
```

Data Transfer Instructions

- **PUSH: PUSH** instruction is used to store the word in a register or a memory location into **stack** as explained in stack addressing modes. SP is decremented by 2 after execution of PUSH.
- Examples:
- **PUSH CX; PUSH CX content in stack**
- **PUSH DS; PUSH DS content in stack**
- **PUSH [BX]; PUSH word in memory at [BX] into stack**
- **PUSH [5000H]**; content of location 5000H & 5001H in DS are pushed onto the stack

Data Transfer Instructions

- **POP:** POP instruction copies the top word from the **stack** to a destination specified in the instruction. The destination can be a general purpose register, a segment register or a memory location. After the word is copied to the specified destination, the SP is incremented by 2 to point to the next word in the stack.
- Examples:
- POP BX ; Pop BX content from the stack
- POP DS ; Pop DS content from the stack
- POP [SI] ; Pop a word from the stack and store it in memory at [SI]
- Note: [SI] indicates the memory location in data segment at offset address specified by SI.

Data Transfer Instructions

- **XCHG:** The XCHG instruction exchanges the contents of a register with the contents of a memory location. It cannot exchange directly the contents of two memory locations. The source and destination must both be words or they must both be bytes. The segment registers cannot be used in this instruction.
- Examples:
- XCHG AL, BL; Exchanges content of AL and BL
- XCHG CX, BX; Exchanges content of CX and BX
- XCHG AX, [BX]; Exchanges content of AX with content of memory at [BX]

- **XLAT:** The XLAT instruction is used to translate a byte in AL from one code to another code. The instruction replaces a byte in the AL register with a byte in memory at [BX], which is data in a lookup table present in memory.
- Before XLAT is executed, the lookup table containing the desired codes must be put in data segment and the offset address of the starting location of the lookup table is stored in BX. The code byte to be translated is put in AL. When XLAT is now executed, it adds the content of the AL with BX to find the offset address of the data in the above lookup table and the byte in that offset address is copied to AL.

Data Transfer Instructions

- **IN:** The IN instruction copies data from a port to AL or AX register. If an 8-bit port is read, the data is stored in AL and if a 16 bit port is read, the data is stored in AX. The IN instruction has two formats namely fixed port and variable port.
- For the fixed port type IN instruction, the 8-bit address of a port is specified directly in the instruction. With this form, anyone of 256 possible ports can be addressed.
- Examples:
- **IN AL, 80H; Input a byte from the port with address 80H to AL**
- **IN AX, 40H; Input a word from port with address 40H to AX**
- For the variable port type IN instruction, the port address is loaded into DX register before the IN instruction. Since DX is a 16 bit register, the port address can be any number between 0000H and FFFFH. Hence up to 65536 ports are addressable in this mode. The following example shows a part of the program having IN instruction and the operations done when the instructions are executed are given in the corresponding comment field.
- Examples:
- **MOV DX, 0FE50H ; Initialize DX with port address of FE50H**
- **IN AL, DX ; Input a byte from 8-bit port with port address FE50H into AL**
- **IN AX, DX ; Input a word from 16-bit port with port address FE50H into AX**

Data Transfer Instructions

- **OUT:** The OUT instruction transfers a byte from AL or a word from AX to the specified port. Similar to IN instruction, OUT instruction has two forms namely fixed port and variable port.
- Examples for fixed port OUT instruction:
- **OUT 48H,AL ; Send the content of AL to port with address 48H**
- **OUT 0F0H,AX ; Send the content of AX to port with address FOH**
- Example for variable port OUT instruction:
- The following example shows a part of the program having OUT instruction and the operations done when the instructions are executed are given in the corresponding comment field.
- MOV DX, 1234H; Load port address 1234H in DX
- OUT DX, AL; Send the content of AL to port with address 1234H
- OUT DX,AX; Send the content of AX to port with address 1234H

Data Transfer Instructions

- **LEA:** Load Effective Address
- The general format of LEA instruction is
- LEA register, source
- This instruction determines the offset address of the variable or memory location named as the source and puts this offset address in the indicated 16 bit register.
- Examples:
- LEA BX, COST; Load BX with offset address of COST in data segment where
- COST is the name assigned to a memory location in data segment.
- LEA CX, [BX][SI]; Load CX with the value equal to (BX)+(SI) where (BX) and
- (SI) represents content of BX and SI respectively.

Data Transfer Instructions

- LDS: Load register and DS with words from memory
- The general form of this instruction is
- LDS register, memory address of first word
- The LDS instruction copies a word from the memory location specified in the instruction into the register and then copies a word from the next memory location into the DS register.
- LDS is useful for initializing SI and DS registers at the start of a string before using one of the String instructions.
- Example:
- LDS SI,[2000H]; Copy content of memory at offset address 2000H in data segment to lower byte of SI, content of 2001H to higher byte of SI. Copy content at offset address 2002H in data segment to lower byte of DS and 2003H to higher byte of DS.

Data Transfer Instructions

- **LES, LSS:** LES and LSS instructions are similar to LDS instruction except that instead of DS register, ES and SS registers are loaded respectively along with the register specified in the instruction.
- **LAHF:** This instruction copies the low byte of flag register into AH.
- **SAHF:** Store content of AH in the low byte of flag register.
- Except SAHF and POPF instructions, all other data transfer instructions do not affect flag register.

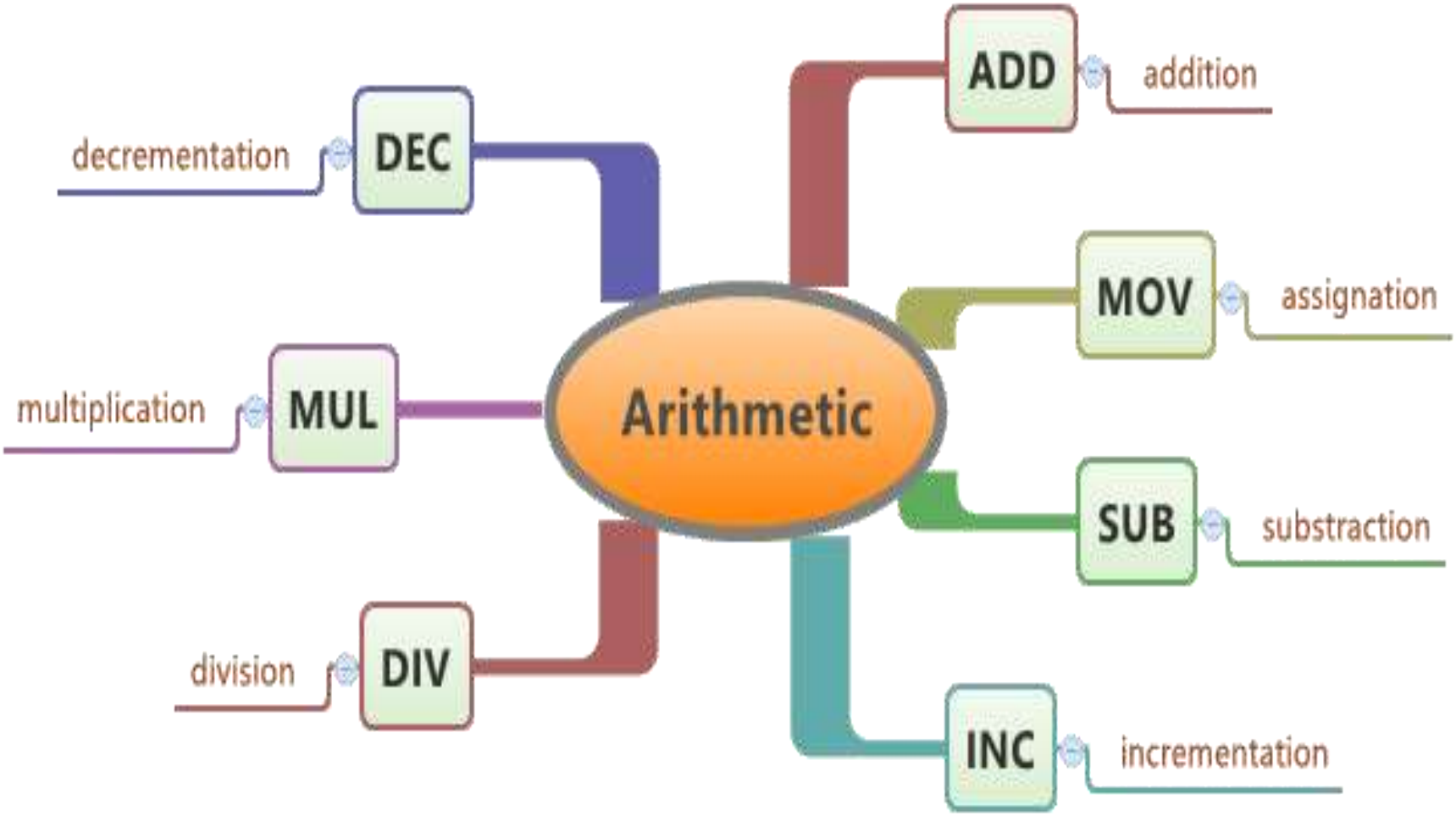
Session - 3

- Data Conversion Instructions, Arithmetic Instructions

&

- Example Programs

Arithmetic Instructions



Picture Courtesy :Google

Arithmetic Instructions

- ADD: The general format of ADD instruction is
- ADD destination, source
- The data from the source and destination are added and the result is placed in the destination. The source may be an immediate number, a register or a memory location. The destination can be a register or memory location. But the source and destination cannot both be memory locations. The data from the source and destination must be of the same type either bytes or words.
- Examples: ADD BL,80H; Add immediate data 80H to BL
- ADD CX,12B0H; Add immediate data 12B0H to CX
- ADD AX,CX; Add content of AX and CX and store result in AX
- ADD AL,[BX]; Add content of AL and the byte from memory at [BX] and store result in AL.
- ADD CX,[SI]; Add content of CX and the word from memory at [SI] and store result in CX.
- ADD [BX],DL; Add content of DL with the byte from memory at [BX] and store result in memory at [BX].
- **The flags AF, CF, OF, PF, SF and ZF flags are affected by the execution of ADD instruction**

Arithmetic Instructions

- **ADC:** This instruction adds the data in source and destination along with the content of carry flag and stores the result in the destination. The general format of this instruction is
- ADC destination, source
- All the rules specified for ADD is also applicable for ADC instruction.
- **SUB:** The general form of subtract (SUB) instruction is
- SUB destination, source
- It subtracts the number in the source from the number in the destination and stores the result in destination. The source may be an immediate number, a register or a memory location. The destination can be a register or memory location. But the source and destination cannot both be memory locations. The data from the source and destination must be of the same type either bytes or words.

Arithmetic Instructions

- For subtraction, the carry flag (CF) functions as borrow flag. If the result is negative after subtraction, CF is set, otherwise it is reset. The rules for source and destination are same as that of ADD instruction. The flags AF, CF, OF, PF, SF and ZF are affected by SUB instruction.
- Examples:
- **SUB AL, BL; Subtract BL from AL and store result in AL**
- **SUB CX, BX; Subtract BX from CX and store result in CX**
- **SUB BX, [DI]; Subtract the word in memory at [DI] from BX and store result in BX**
- **SUB [BP], DL; Subtract DL from the byte in memory at [BP] and store result in memory at [BP].**

Arithmetic Instructions

- **SBB: Subtract with Borrow**
- The general form of this instruction is
- SBB destination, source
- SBB instruction subtracts the content of source and content of carry flag from the content of destination and stores the result in destination. The rules for the source and destination are same as that of SUB instruction.
- AF, CF, OF, PF, SF and ZF are affected by this instruction.

Arithmetic Instructions

- **INC:** The increment (INC) instruction adds 1 to a specified register or to a memory location. The data incremented may be a byte or word. Carry flag is not affected by this instruction. AF, OF, PF, SF and ZF flags are affected.
- Examples:
- INC CL; Increment content of CL by 1
- INC AX; Increment content of AX by 1
- INC BYTE PTR [BX]; Increment byte in memory at [BX] by 1
- INC WORD PTR [SI]; Increment word in memory at [SI] by 1
- In the above examples, the term BYTE PTR and WORD PTR are assembler directives which are used to specify the type of data (byte or word respectively) to be incremented in memory.

Arithmetic Instructions

- **DEC:** The decrement (DEC) instruction subtracts 1 from the specified register or memory location. The data decremented may be a byte or word. CF is not affected and AF,OF, PF, SF and ZF flags are affected by this instruction.
- **NEG:** The negate (NEG) instruction replaces the byte or word in the specified register or memory location by its 2's complement (i.e. changing the sign of the data). CF,AF, SF, PF, ZF and OF flags are affected by this instruction.
- Examples:
- NEG AL; Take 2's complement of the data in AL and store it in AL
- NEG CX; Take 2's complement of the data in CX and store it in CX
- NEG BYTE PTR[BX]; Take 2's complement of the byte in memory at [BX] and store result in the same place.
- NEG WORD PTR[SI]; Take 2's complement of the word in memory at [SI] and store result in the same place.

Arithmetic Instructions

- **CMP:** The general form of compare (CMP) instruction is given below:
- CMP destination, source
- This instruction compares a byte or word in the source with a byte or word in the destination and affects only the flags according to the result. The content of source and destination are not affected by the execution of this instruction. The comparison is done by subtracting the content of source from destination.
- AF, OF, SF, ZF, PF and CF flags are affected by the instruction. The rules for source and destination are same as that of SUB instruction.
- Example:
- After the instruction CMP AX, DX is executed, the status of CF, ZF and SF will be as follows:

	CF	ZF	SF
If AX=DX	0	1	0
If AX>DX	0	0	0
If AX<DX	1	0	1

Arithmetic Instructions

- **MUL:** The multiply (MUL) instruction is used for multiplying two unsigned bytes or words. The general form of MUL instruction is
- MUL Source
- The source can be byte or word from a register or memory location which is considered as the multiplier. The multiplicand is taken by default from AL or AX for byte or word type data respectively. The result of multiplication is stored in AX or DX-AX (i.e. Most significant word of result in DX and least significant word of result in AX) for byte or word type data respectively. (Note: Multiplying two 8 bit data gives 16 bit result and multiplying two 16 bit data gives 32 bit result).
- Examples:
- MUL CH; Multiply AL and CH and store result in AX
- MUL BX; Multiply AX and BX and store result in DX-AX
- MUL BYTE PTR [BX]; multiply AL with the byte in memory at [B X] and store result in DX-AX
- If the most significant byte of the 16 bit result is 00H or the most significant word of a 32 bit result is 0000h, both CF and OF will both be 0s. Checking these flags allows us to decide whether the leading 0s in the result have to be discarded or not. AF, PF, SF and ZF flags are undefined (i.e. some random number will be stored in these bits) after the execution of MUL instruction

Arithmetic Instructions

- **IMUL:** The IMUL instruction is used for multiplying signed byte or word in a register or memory location with AL or AX respectively and stores the result in AX or DX-AX respectively. If the magnitude of the result does not require all the bits of the destination, the unused bits are filled with copies of the sign bit.
- If the upper byte of a 16 bit result or upper word of a 32 bit result contains only copies of the sign bit (all 0s or all 1s) then CF and OF will both be 0 otherwise both will be 1. **AF, PF, SF and ZF are undefined after IMUL.**
- To multiply a signed byte by a signed word, the byte is moved into a word location and the upper byte of the word is filled with the copies of the sign bit. If the byte is moved into AL, by using the CBW (Convert Byte to Word) instruction, the sign bit in AL is extended into all the bits of AH. Thus AX contains the 16 bit sign extended word.
- Examples:
- IMUL BL; multiply AL with BL and store result in AX
- IMUL AX; multiply AX and AX and store result in DX-AX
- IMUL BYTE PTR [BX]; multiply AL with byte from memory at [BX] and store result in AX
- IMUL WORD PTR [SI]; Multiply AX with word from memory at [SI] and store result in DX-AX

Arithmetic Instructions

- **DIV:** The divide (DIV) instruction is used for dividing unsigned data. The general form of DIV instruction is
- DIV source
- Where source is the divisor and it can be a byte or word in a register or memory location. The dividend is taken by default from AX and DX-AX for byte or word type data division respectively.
- Examples
- DIV DL; Divide word in AX by byte in DL. Quotient is stored in AL and remainder is stored in AH
- DIV CX; Divide double word (32 bits) in DX-AX by word in CX. Quotient is stored in AX and remainder is stored in DX
- DIV BYTE PTR [BX]; Divide word in AX by byte from memory at [BX]. Quotient is stored in AL and remainder is stored in AH.

Arithmetic Instructions

- **IDIV:** The IDIV instruction is used for dividing signed data. The general form and the rules for IDIV instruction are same as DIV instruction. The quotient will be a signed number and the sign of the remainder is same as the sign of the dividend.

- To divide a signed byte by a signed byte, the dividend byte is put in AL and using CBW (Convert Byte to Word) instruction, the sign bit of the data in AL is extended to AH and thereby the byte in AL is converted to signed word in AX. To divide a signed word by a signed word, the dividend byte is put in AX and using CWD (Convert Word to Double word) instruction, the sign bit of the data in AX is extended to DX and thereby the word in AX is converted to signed double word in DX-AX.

- If an attempt is made to divide by 0 or if the quotient is too large or two low to fit in AL or AX for 8 or 16 bit division respectively (i.e. either when the result is greater than +127 decimal in 8 bit division or +32767 decimal in 16 bit division or if the result is less than -128 decimal in 8 bit division or -32767 decimal in 16 bit division), the 8086 automatically generate a type 0 interrupt. All flags are undefined after a DIV instruction.

Arithmetic Instructions

- **DAA:** Decimal Adjust AL after BCD addition
- This instruction is used to get the result of adding two packed BCD numbers (two decimal digits are represented in 8 bits) to be a BCD number. The result of addition must be in AL for DAA to work correctly. If the lower nibble (4 bits) in AL is greater than 9 after addition or AF flag is set by the addition then the DAA will add 6 to the lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the carry flag is set by the addition, then the DAA will add 60H to AL.
- Examples:
- Let AL=0101 1000=58 BCD
- CL=0011 0101=35 BCD
- Consider the execution of the following instructions:
- ADD AL, CL; AL=10001101=8DH and AF=0 after execution
- DAA - Add 0110 (decimal 6) to AL since lower nibble in AL is greater than 9
- AL=10010011= 93 BCD and CF=0
- Therefore the result of addition is 93 BCD.

Arithmetic Instructions

- **DAS:** Decimal Adjust after BCD subtraction
- DAS is used to get the result is in correct packed BCD form after subtracting two packed BCD numbers. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by subtraction then the DAS will subtract 6 from the lower nibble of AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS will subtract 60H from AL.
- Examples:
- Let AL=86 BCD=1000 0110
- CH=57 BCD=0101 0111
- Consider the execution of the following instructions:
- SUB AL, CH; AL=0010 1111=2FH and CF=0 after execution
- DAS; Lower nibble of result is 1111, so DAS subtracts 06H from AL to make
- AL=0010 1001=29 BCD and CF=0 to indicate there is no borrow.
- The result is 29 BCD.

Arithmetic Instructions

- **AAA:**
- AAA (ASCII Adjust after Addition) instruction must always follow the addition of two unpacked BCD operands in AL. When AAA is executed, the content of AL is changed to a valid unpacked BCD number and clears the top 4 bits of AL. The CF is set and AH is incremented if a decimal carry out from AL is generated.
- Example:
- Let AL=05 decimal=0000 0101
- BH=06 decimal=0000 0100
- AH=00H
- Consider the execution of the following instructions:
- ADD AL, BH ; AL=0BH=11 decimal and CF=0
- AAA ; AL=01 and AH=01 and CF=1
- Since 05+06=11(decimal)=0101 H stored in AX in unpacked BCD form.
- When this result is to be sent to the printer, the ASCII code of each decimal digit is easily formed by adding 30H to each byte.

Arithmetic Instructions

- **AAS:** ASCII Adjust after Subtraction
- This instruction always follows the subtraction of one unpacked BCD operand from another unpacked BCD operand in AL. It changes the content of AL to a valid unpacked BCD number and clears the top 4 bits of AL. The CF is set and AH is decremented if a decimal carry occurred.
- Example:
- Let AL=09 BCD=0000 1001
- CL=05 BCD =0000 0101
- AH=00H
- Consider the execution of the following instructions:
- SUB AL, CL; AL=04 BCD
- AAS ; AL=04 BCD and CF=0
- ; AH=00H
-
- AAA and AAS affect AF and CF flags and OF, PF, SF and ZF are left undefined. Another salient feature of the above two instructions are that the input data used in the addition or subtraction can be even in ASCII form of the unpacked decimal number and still we get the result in ordinary unpacked decimal number form and by adding 30H to the result , again we get ASCII form of the result.

Arithmetic Instructions

- **AAD:** The ASCII adjust AX before Division instruction modifies the dividend in AH and AL, to prepare for the division of two valid unpacked BCD operands. After the execution of AAD, AH will be cleared and AL will contain the binary equivalent of the original unpacked two digit numbers. Initially AH contains the most significant unpacked digit and AL contains the least significant unpacked digit.
- Example: To perform the operation 32 decimal / 08 decimal
- Let AH=03H; upper decimal digit in the dividend
- AL=02H; lower decimal digit in the dividend
- CL=08H; divisor
- Consider the execution of the following instructions:
- AAD; AX=0020H (binary equivalent of 32 decimal in 16 bit form)
- DIV CL; Divide AX by CL; AL will contain the quotient and AH will contain the remainder.
- AAD affects PF, SF and ZF flags. AF, CF and OF are undefined after execution of AAD.

Arithmetic Instructions

- **AAM:** The ASCII Adjust AX after Multiplication instruction corrects the value of a multiplication of two valid unpacked decimal numbers. The higher order digit is placed in AH and the low order digit in AL.
- Example:
- Let AL=05 decimal
- CL=09 decimal
- Consider the execution of the following instructions:
-
- MUL CH; AX=002DH=45 decimal
- AAM; AH=04 and AL=05 (unpacked BCD form decimal number of 45)
- OR AX, 3030H; To get ASCII code of the result in AH and AL (Note:this instruction is used only when it is needed). AAM affects flags same as that of AAD.

Example-Simple programs

```
MOV AX, [1100]
MOV BX, [1102]
ADD AX,BX
MOV [1200],AX
HLT
```

```
MOV AX, [1100]
MOV BX, [1102]
SUB AX,BX
MOV [1200],AX
HLT
```

```
MOV AX, [1100]
MOV BX, [1102]
MUL BX
MOV [1200],AX
MOV [1202],DX
HLT
```

```
MOV AX, [1100]
MOV BX, [1102]
DIV BX
MOV [1200],AX
MOV [1202],DX
HLT
```

Learning Resources

1. K. M. Bhurchandi and A. K. Ray, *Advanced Microprocessors and Peripherals-with ARM and an Introduction to Microcontrollers and Interfacing*, 3rd edition, Tata McGraw Hill, 2015.

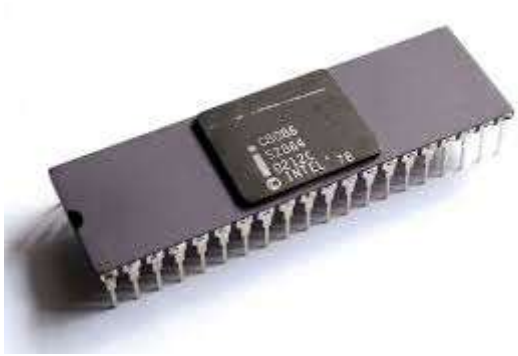
2. Yu-cheng Liu, Glenn A. Gibson, *Microcomputer systems: The 8086/8088 family-Architecture, programming and design,* 2nd edition, Prentice Hall of India, 2007.

Thank You

18ECC203J: Module – 2

Faculty of Engineering and Technology, SRM Institute of Science and Technology

Programming with Intel 8086



Organized by

Dr. K. A. Sunitha, Associate Professor, EIE, SRMIST

Dr. B. Ramakrishna, Assistant Professor, ECE, SRMIST

Picture Courtesy :Google

Session – 6

- Logical instructions and Processor control instructions

&

- Example Programs

Logical Instructions

- **AND:** The AND instruction perform logical AND operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of AND instruction is
- AND destination, Source
- The rules for destination and source for AND instruction are same as that of ADD instruction. CF and OF are both 0 after AND. PF, SF and ZF are updated after execution of AND instruction. AF is undefined. PF has meaning only for ANDing 8-bit operand.
- **OR:** The OR instruction perform logical OR operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of OR instruction is
- OR destination, Source
- The rules for the source and destination and the way flags are affected for OR instruction are same as that of AND instruction.

Logical Instructions

- **XOR:** The XOR instruction performs logical XOR operation between the corresponding bits in the source and destination and stores the result in the destination. Both the data can be either bytes or words. The general form of XOR instruction is
- XOR destination, source
- The rules for the same source and destination and the way flags are affected for XOR instruction are same as that of AND instruction.

- **NOT:** The Not instruction inverts each bit (forms the 1's complement) of the byte or word at the specified destination. The destination can be a register or a memory location. No flags are affected by the NOT instruction.
- Example:
- NOT AL; Take 1's complement of AL
- NOT BX; Take 1's complement of BX
- NOT [SI]; Take 1's complement of data in memory at [SI]

Logical Instructions

- **TEST:** This instruction ANDs the content of a source byte or word with the content of the specified destination byte or word respectively. Flags are updated, but *neither operand is changed*. The TEST instruction is often used to set flags before a conditional jump instruction. The general form of TEST instruction is

- TEST destination, source

- The rules for the source and destination are same as that of AND instruction and the way flag are affected is also same as that of AND instruction.

- Example:

- Let AL=0111 1111 =7FH

- TEST AL, 80H; AL=7FH (unchanged)

- **ZF=1 since (AL) AND (80H)=00H; SF=0; PF=1**

Example-Simple programs

```
MOV AX, [1100H]
AND AX,0F0FH
MOV [1200H],AX
HLT
```

```
   MOV AX, 0000
OR AX,0F0F
MOV [1200],AX
   HLT
```

Machine or Processor Control Instructions

- **HLT:** The halt instruction stops the execution of all instructions and places the processor in a halt state. An interrupt or a reset signal will cause the processor to resume execution from the halt state.

- **LOCK:** The lock instruction asserts for the processor an exclusive hold on the use of the system bus. It activates an external locking signal ( ) of the processor and it is placed as a prefix in front of the instruction for which a lock is to be asserted. The lock will function only with the XCHG, ADD, OR, ADC, SBB, AND, SUB, XOR, NOT, NEG, INC and DEC instructions, when they involve a memory operand. An undefined opcode trap interrupt will be generated if a LOCK prefix is used with any instruction not listed above.

- **NOP:** No operation. This instruction is used to insert delay in software delay programs.

Machine or Processor Control Instructions

- **ESC:** This instruction is used to pass instructions to a coprocessor such as the 8087, which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6- bit code embedded in the escape instruction.

- As the 8086 fetches instruction bytes from memory the coprocessor also catches these bytes from the data bus and puts them in a queue. However the coprocessor treats all the normal 8086 instructions as NOP instruction. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6- bit code specified in the instruction.

- **WAIT:** When this instruction is executed, the 8086 checks the status of its input pin and if the input is high, it enters an idle condition in which it does not do any processing. The 8086 will remain in this state until the 8086's input pin is made low or an interrupt signal is received on the INTR or NMI pins. If a valid interrupt occurs while the 8086 is in this idle state, it will return to the idle state after the interrupt service routine is executed. WAIT instruction does not affect flags. It is used to synchronize 8086 with external hardware such as 8087 coprocessor.

Plus …

- Shift and Rotate Instructions
- &
- Flag Manipulation Instructions

Shift Instructions

- **Shift and Rotate instructions:** The Shift instructions perform logical left shift and right shift, and arithmetic left shift and right shift operation. The arithmetic left shift (SAL) and logical left shift (SHL) have the same function.

- SAL/SHL: The general format of SAL/SHL instruction is

- **SAL/SHL Destination, Count**

- The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB position, a 0 is put in the LSB position. The MSB will be shifted into carry flag (CF) as shown below: CF MSB LSB 0

- If the number of shifts to be done is 1 then it can be directly specified in the instruction with count equal to 1. For shifts of more than one bit position, the desired number of shifts is loaded into the CL register and CL is put in the count position of the instruction. CF, SF and ZF are affected according to the result. PF has meaning only when AL is used as destination. SAL instruction can be used to multiply an unsigned number by a power of 2. Doing one bit or two bits left shift of a number multiplies the number by 2 or 4 respectively and so on.

- Examples: **SAL AX,1; Shift left the content of AX by 1 bit**

- **SAL BL,1; Shift left the content of BL by 1 bit**

- **SAL BYTE PTR [SI],1; Shift left the byte content of memory at [SI] by 1 bit**

Shift Instructions

- **SAR:** The general format of SAR instruction is
- SAR Destination, Count
- The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position (i.e. the sign bit is copied into the MSB). The LSB will be shifted into carry flag (CF) as shown below:
- MSB MSB LSB CF
- The rules for the Count in the instruction are same as that of the SAL instruction. CF, SF and ZF are affected according to the result. PF has meaning only when AL is used as destination.

Shift Instructions

- **SHR:**
- The general format of SHR instruction is
- SHR Destination, Count
- The destination can be a register or a memory location and it can be a byte or a word. This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in the MSB position. The LSB will be shifted into carry flag (CF) as shown below:
- 0 MSB LSB CF
- The rules for the Count in the instruction are same as that of the SHL instruction. CF, SF and ZF are affected according to the result. PF has meaning only when 8 bit destination is used.

Rotate Instructions

- **ROR:** This instruction rotates all the bits of the specified byte or word by some number of bit positions to the right. The operation done when ROR is executed is shown below. The general format of ROR instruction is
- ROR Destination, Count
- The data bit moved out of the LSB is also copied into CF. ROR affects only CF and OF. For single bit rotate, OF will be 1 after ROR, if the MSB is changed by the rotate. ROR is used to swap nibbles in a byte or to swap two bytes within a word. It can also be used to rotate a bit in a byte or word into CF, where it can be checked and acted upon by the JC and JNC instruction. CF will contain the bit most recently rotated out of LSB in the case of multiple bit rotate.
- The rules for count are same as that of the shift instruction which is discussed above.
- Examples: **ROR CH, 1 : rotate right byte in CH by one bit position.**
- **ROR BX, CL: Rotate right word in BX by number of bit positions given by CL.**

Rotate Instructions

- ROL: ROL rotates all the bits in a byte or word in the destination to the left either by 1 bit position and more than 1 bit positions using CL as shown below:

- The data bit moved out of the MSB is also copied into CF. ROL affects only the CF and OF. OF will be 1 after a single bit ROL if the MSB was changed by the rotate. ROL is used to swap nibbles in a byte or to swap two bytes within a word. It can also be used to rotate a bit in a byte or word into CF, where it can be checked and acted upon by the JC and JNC instruction. CF will contain the bit most recently rotated out of LSB in the case of multiple bit rotate.

Rotate Instructions

- **RCR:** Rotate the byte or word in the destination right through carry flag (CF) either by one bit position or the number of bit positions given by the CL as shown below.
- The flag are affected by similar to ROR.
- **RCL:** Rotate the byte or word in the destination left through carry flag (CF) either by one bit position or the number of bit positions given by the CL as shown below.
- The flags are affected similar to ROL.

Flag Manipulations Instructions

Mnemonics	Function
LAHF	Load low byte of flag register into AH
SAHF	Store AH into the low byte of flag register
PUSHF	Push flag register's content into stack
POPF	Pop top word of stack into flag register
CMC	Complement carry flag (CF = complement of CF)
CLC	Clear carry flag (CF= 0)
STC	Set carry flag (CF= 1)
CLD	Clear direction flag (DF= 0)
STD	Set direction flag (DF= 1)
CLI	Clear interrupt flag (IF= 0)
STI	Set interrupt flag (IF=1)

Session - 7

- String Instructions

&

- Example Programs

String Instructions

- The string instructions operate on element of strings of bytes or word.
- Registers SI and DI contain the offset address within a segment of an element (Byte or Word) in the source string and the destination string respectively.
- The source string is in the data segment at the offset address given by SI and destination string is in the extra segment at the offset address given by DI.
- After each string operation, SI and/or DI are automatically incremented or decremented by 1 or 2 (for byte or word operation) according to the D flag in the flag register. If D=0, SI and/or DI are automatically incremented and if D=1, SI and/or DI are automatically decremented.

String Instructions

MNEMONICS	FUNCTION
MOVSB	Move string byte from DS:[SI] to ES:[DI]
MOVSW	Move string word from DS:[SI] to ES:[DI]
CMPSB	Compare string byte (Done by subtracting byte at ES:[DI] from the byte at DS:[SI]). Only flags are affected and the content of bytes compared is unaffected.
CMPSW	Compare string word (Done by subtracting word at ES:[DI] from the word at DS:[SI]). Only flags are affected and the content of words compared is unaffected.
LODSB	Load string byte at DS:[SI] into AL
LODSW	Load string word at DS:[SI] into AX
STOSB	Store string byte in AL at ES:[DI]
STOSW	Store string word in AX at ES:[DI]
SCASB	Compare string byte (Done by subtracting byte at ES:[DI] from the byte at). Only flags are affected and the content of bytes compared is unaffected.
SCASW	Compare string word (Done by subtracting word at ES:[DI] from the byte at AX). Only flags are affected and the content of words compared is unaffected.
REP	Decrement CX and Repeat the following string operation if CX ≠ 0.
REPE or REPZ	Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=1.
REPNE or REPNZ	Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=0.

Session - 8

- Branch Instructions

&

- Example Programs

Control Transfer Instructions

Mnemonics	Descripton
JMP addr	Jump unconditionally to addr
CALL addr	Call procedure or subroutine starting at addr
RET	Return from procedure or subroutine
JA addr	Jump if above to addr (jump if CF = ZF =0)
JAE addr	Jump if above or equal to addr (jump if CF=0)
JB addr	Jump if below to addr (jump if CF=1)
JBE addr	Jump if below or equal to addr (Jump if CF =1 or ZF = 1)
JC addr	jump if carry to addr (jump if CF = 1)
JCXZ addr	Jump if CX = 0
JE addr	Jump if equal (jump if ZF = 1)

Control Transfer Instructions

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

JG addr	Jump if greater (Jump if ZF = 0 and SF = OF)
JGE addr	Jump if greater or equal (Jump if SF = OF)
JL addr	Jump if less (Jump if SF ≠ OF)
JLE addr	Jump if less or equal (Jump if ZF=1 or SF ≠ OF)
JNA addr	Jump if not above (Jump if CF = 1 or ZF = 1)
JNAE addr	Jump if not above or equal (Jump if CF = 1)
JNB addr	Jump if not below (Jump if CF = 0)
JNBE addr	Jump if not below or equal (Jump if CF = ZF = 0)
JNC addr	Jump if not carry (Jump if CF = 0)
JNE addr	Jump if not equal (jump if ZF =0)
JNG addr	Jump if not greater (jump if ZF =1 or SF ≠ OF)
JNGE addr	Jump if not greater or equal (jump if SF ≠ OF)

Control Transfer Instructions

JL addr	Jump if not less (Jump if SF=OF)
JNLE addr	Jump if not less or equal (Jump if ZF = 0 and SF = OF)
JNO addr	Jump if not overflow (Jump if OF = 0)
JNP addr	Jump if not parity (Jump if PF = 0)
JNS addr	Jump if not sign (jump if SF=0)
JNZ addr	Jump if not zero (jump if ZF=0)
JO addr	Jump if overflow (jump if OF=1)
JP addr	Jump if parity (jump if PF=1)
JPE addr	Jump if parity even (jump if PF=1)
JPO addr	Jump if parity odd (jump if PF=0)
JS addr	Jump if sign (jump if SF=1)
JZ addr	Jump if zero (jump if ZF=1)

Conditional Jump Instructions

- JE - JZ
- JNE - JNZ
- JL - JNGE
- JNL - JGE
- JG - JNLE
- JNG - JLE
- JB - JNAE
- JNB - JAE
- JA - JNBE
- JNA - JBE
- JP - JPE
- JNP - JPO

Learning Resources

1. K. M. Bhurchandi and A. K. Ray, *Advanced Microprocessors and Peripherals-with ARM and an Introduction to Microcontrollers and Interfacing*, 3rd edition, Tata McGraw Hill, 2015.

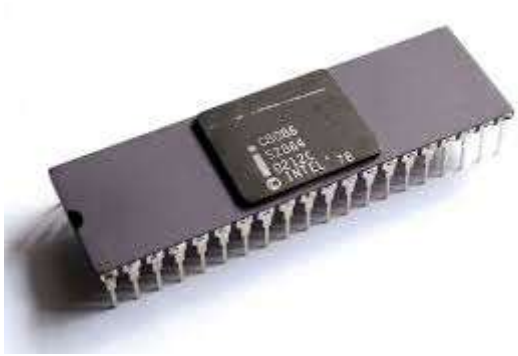
2. Yu-cheng Liu, Glenn A. Gibson, *Microcomputer systems: The 8086/8088 family-Architecture, programming and design,* 2nd edition, Prentice Hall of India, 2007.

Thank You

18ECC203J: Module – 2

Faculty of Engineering and Technology, SRM Institute of Science and Technology

Programming with Intel 8086



Organized by

Dr. K. A. Sunitha, Associate Professor, EIE, SRMIST

Dr. B. Ramakrishna, Assistant Professor, ECE, SRMIST

Picture Courtesy :Google

Session - 11

- Assembly Language Programming of 8086

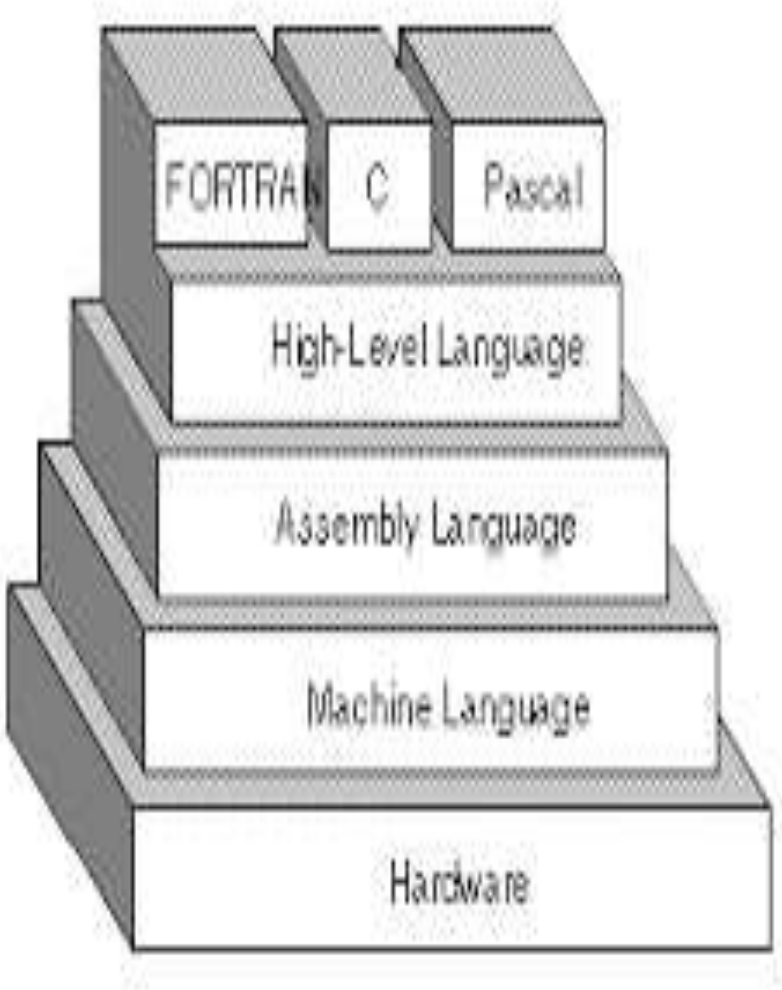
How does our Microprocessor Based System looks like..?

Working Model from Laboratory

Note: Information Provided in the next slides is subjected with © Oxford University Press 2013

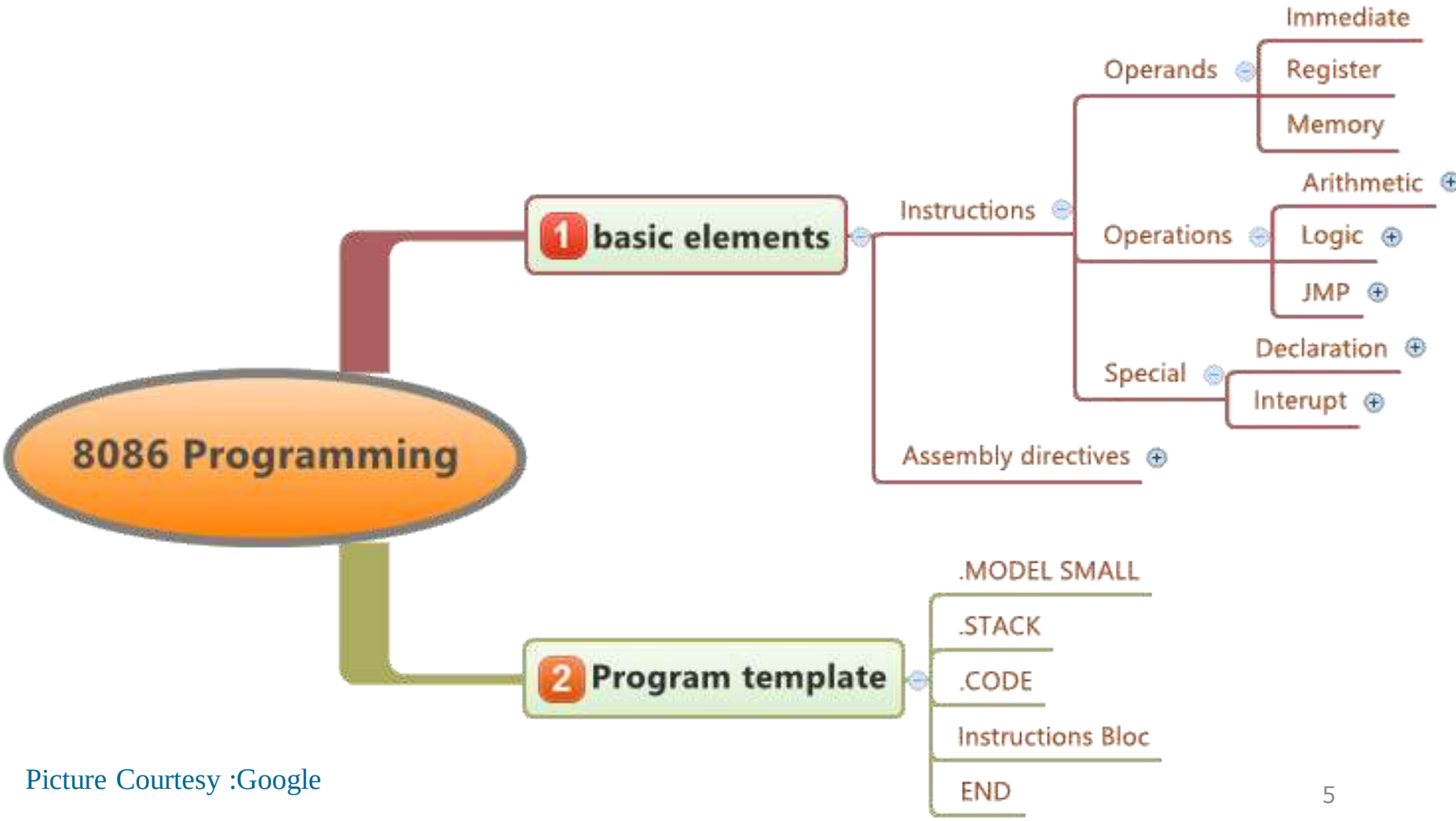
Review of Basics-Layers of Programming

- **Machine Language**
  - **Binary**
  - **Hexadecimal**
  - **Machine code or object code**
- **Assembly Language**
  - **Mnemonics**
  - **Assembler**
- **High-Level Language**
  - **Pascal, Basic, C**
  - **Compiler**



Picture Courtesy :Google

Program Structure



Picture Courtesy :Google

8086 Assembly Language Programming

- A few assembly language programming examples similar to the assembly language programming of 8085 are given in this section.
- These programs can be converted into machine language programs and executed in the 8086 based system either manually finding the opcode for each instruction or by using assembler. As finding the opcode of each instruction of 8086 manually is time consuming, line assembler or assembler is normally used.
- The line assembler converts each mnemonics of an instruction immediately into opcode as it is entered in the system and this type of assembler is used in microprocessor trainer kits. Line assembler is stored in any one of ROM type memory in the trainer kit.
- The assembler needs a personal computer for generating the opcodes of an assembly language program and the generated opcodes can be downloaded to the microprocessor based system such as microprocessor trainer kit or microprocessor based prototype hardware through serial port or parallel port of the computer.

8086 Assembly Language Programming

- There are many assemblers like MASM (Microsoft Macro Assembler), TASM (Turbo Assembler) and DOS Assembler which are used to convert the 8086 assembly language program into machine language program.
- While using these assemblers, the assembly language program is written using assembler directives. Assembler directives are commands to the assembler to indicate the size of a variable (either byte or word), number of bytes or words to be reserved in memory, value of a constant and name of a segment, etc. in a program.
- The assembler directives are not converted directly into opcode but they are used to generate proper opcode of an instruction. The use of Microsoft assembler alone is discussed in the second part of this section.

Writing 8086 program using line assembler

- Write a program to add a word type data located at offset 0800H (Least Significant Byte) and 0801H (Most Significant Byte) in the segment address 3000H to another word type data located at offset 0700H (Least Significant Byte) and 0701H (Most Significant Byte) in the same segment. Store the result at offset 0900H and 0901H in the same segment. Store the carry generated in the above addition in the same segment at offset 0902H.

```
MOV AX, 3000H
MOV DS, AX        ; initialize DS with value 3000H
MOV AX, [800H] ; get first data word in AX
ADD AX, [700H]        ; add AX with second data word
MOV [900H], AX ; store AX at the offset 900H & 901H
JC   CARRY        ; if carry=1, go to the place CARRY
MOV [902H], 00H       ; no carry; hence store 00H at the offset 902H
JMP END           ; go to the place END
CARRY: MOV [902H], 01H      ; store 01H at the offset 902H
END:      HLT             ; stop
```

Writing 8086 Program Using Line Assembler

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

- Write a program to find the smallest word in an array of 100 words stored sequentially in the memory starting from the offset 1000H in the segment address 5000H and store the result at the offset 2000H in the same segment.

```
MOV CX, 99        ; initialize CX with the number of comparisons (=100-1)
MOV AX, 5000H
MOV DS, AX        ; initialize DS with the segment address 5000H
MOV SI, 1000H          ; initialize SI with offset 1000H
MOV AX, [SI]           ; get the first word in AX
START:   INC SI
INC SI                 ; increment SI twice to point the next word
CMP AX, [SI]           ; compare next word with the word in AX
JC REPEAT         ; if AX is smaller then go to REPEAT
MOV AX, [SI]           ; replace the word in AX with the smaller word
REPEAT: LOOP   START   ; repeat the operation from START
MOV  [2000H], AX       ; store smallest number in AX at the offset 2000H
HLT               ; stop
```

Programs: Sorting in Ascending and Descending Order

	MOV SI,1200h
	MOV CL,[SI]
	DEC CL
LOOP3	MOV SI,1200h
	MOV CH,[SI]
	DEC CH
	INC SI
LOOP2	MOV AL,[SI]
	INC SI
	CMP AL,[SI]
	JC LOOP1
	XCHG AL,[SI]
	XCHG [SI-1],AL
LOOP1	DEC CH
	JNZ LOOP2
	DEC CL
	JNZ LOOP3
	HLT

	MOV SI,1200h
	MOV CL,[SI]
	DEC CL
LOOP3	MOV SI,1200h
	MOV CH,[SI]
	DEC CH
	INC SI
LOOP2	MOV AL,[SI]
	INC SI
	CMP AL,[SI]
	JNC LOOP1
	XCHG AL,[SI]
	XCHG [SI-1],AL
LOOP1	DEC CH
	JNZ LOOP2
	DEC CL
	JNZ LOOP3
	HLT

@Students :You may Try to with these Programs….

1. Write an ALP to add first 10 odd numbers and store in the memory location 1234H
2. Write an ALP to generate first 10 Fibonacci numbers and store in 1300H series.

Introducing 8086 Assembler Directives

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

- An **assembler** is a program used to convert an assembly language program into equivalent machine language program. The assembler finds the address of each label and substitutes the value for each of the constants and variables in the assembly language program during the assembly process, to generate the machine language code.

- While doing these things, the assembler may find out syntax errors and it is reported to programmer at the end of assembly process. The logical errors or other programming errors are not found by the assembler.

- For completing all the above tasks, an assembler needs some commands from the programmer namely the required storage class for a particular constant or a variable such as **byte, word or double word, logical name** of the segments such as CODE or STACK or DATA segment, type of different procedures or routines such as FAR or NEAR or PUBLIC or EXTRN and end of a segment, etc.

- These types of commands are given to the assembler using some predefined alphabetical strings called assembler directives which help the assembler to correctly generate the machine codes for the assembly language program.

Assembler directives related to code (i.e. program) location

- A) **ORG:** (**WHY WE ARE USING ORG 100h in Emulator**)

- The ORG (origin) directive directs the assembler to start the memory allocation for a particular segment (data or code or stack) from the declared address in the ORG statement. While starting the assembly process for a memory segment, the assembler initializes a location counter (LC) to keep track of the allotted offset addresses for the segment. When ORG directive is not mentioned at the beginning of the segment, then LC is initialized with the offset address 0000H. When ORG directive is mentioned at the beginning of the segment, then LC is initialized with the offset address specified in the ORG directive.

- **Example: ORG 100H**

- When the above directive is placed at the beginning of code segment, then the location counter is initialized with 0100H and the first instruction is stored from the offset address 0100H within the code segment. If it is placed in data segment then the next data storage will start from the offset address 0100H within the data segment. "You may explore Other Assembler Directives to extend your knowledge"

MACRO

- **Defining a MACRO:**
- A MACRO can be defined anywhere in a program using the directives MACRO and ENDM. The label prior to the MACRO is the macro name which is used in the main program wherever needed. The ENDM directive marks the end of the instructions or statements assigned with the macro name.
- Example:

```
CALCULATE   MACRO
            MOV AX, [BX]
            ADD AX, [BX+2]
            MOV [SI], AX
            ENDM
```

Passing parameters to a MACRO:

- Using parameters in a macro definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called. The above macro (CALCULATE) can be modified to calculate the result for different sets of data and store it in different memory locations as given below:

```
CALCULATE       MACRO     OPERAND, RESULT
                MOV BX, OFFSET OPERAND
                MOV AX, [BX]
                ADD AX, [BX+2]
                MOV SI, OFFSET RESULT
                MOV [SI], AX
                ENDM
```

- The parameters OPERAND and RESULT can be replaced by OPERAND1, RESULT1 and OPERAND2, RESULT2 while calling the above macro as shown below: ……..

```
                CALCULATE       OPERAND1, RESULT1
                ……..
                ……
                CALCULATE       OPERAND2, RESULT2
```

Writing Assembly Language Programs While Using MASM

- MASM (Microsoft Macro Assembler) is one of the assemblers commonly used along with a LINK (linker) program to structure the machine codes generated by MASM in the form of an executable (EXE) file. The MASM reads the assembly language program, which is known as source program and produces an object file as output

- The LINK program accepts the object file produced by MASM along with library files if needed, and produces an EXE file.

- While writing a program for MASM, first the program listing is typed using a text editor in the computer such as Norton's Editor (NE), Turbo C editor, etc. After the program editing is done, it is saved under a name with an extension .ASM, for example MSI.ASM is a valid file name which can be assigned for an assembly language program.

- The programmers have to ensure that all the files namely the editor, MASM.EXE (MASM assembler) and LINK.EXE (linker) are available in the same directory. After editing, assembling of the program has to be done using MASM.

Writing Assembly Language Programs While Using MASM

- If all the above software are present in root directory of C drive in the computer, then to assemble the file name MSI.ASM, the programmer has to type the following at the DOS prompt in the computer:
- C:\ > MASM MSI.ASM
- (or)
- C:\ > MASM MSI
- After entering the above command, the assembler asks for giving the name of the following files, which it will generate after assembly process as given below:
- Object file name [.OBJ]:
- List file name [NUL.LST]:
- Cross reference [NUL.CRF]:

Writing Assembly language programs while using MASM

- The programmer can type a name against every file name and then press ‘enter’ key after giving each name. If no name is entered against each file name before pressing the enter key in all cases then all the above three files will have the same name as the source file.

- The .OBJ (object) file contains the machine codes of the program which is assembled. The .LST (list) file contains the total offset map of the source file including labels, opcodes, offset addresses, memory allotment for different labels and directives.

- The cross reference (.CRF) file is used for debugging the source program as it contains the information such as size of the file in bytes, list of labels, number of labels and routines to be called, etc. in the source program.

- After the cross reference file name is entered, the assembly process starts. If the program contains syntax errors, they are displayed using error code number and the corresponding line numbers at which they appear. Once these errors are corrected by the programmer, the assembly process will be completed successfully.

Writing Assembly Language Programs While Using MASM

- The DOS linking program LINK.EXE is used to link the different object modules of a source program and the function library routines to produce an integrated executable code for the source program. The linker is invoked using the following command:

- C:\> LINK MSI.OBJ

- After entering the above command, the linker asks for giving the name of the following files: Run file [.EXE]:

- List files [NUL.MAP]:

- Libraries [LIB]:

- If no file names are entered for these files, then by default, the source file name is considered with different extensions. The option input 'Libraries' expects any special library name from which the functions were used by the source program. The output of the linker program is an executable file with the entered or default file name with .EXE extension. The executable file name can be entered at the DOS prompt to execute the file as shown below:

- C:\> MSI.EXE

Writing Assembly Language Programs While Using MASM

- In the advanced version of MASM, both assembling and linking is combined under a single menu invokable compile function.
- The DEBUG.com is a DOS utility program that is used for debugging and trouble – shooting 8086 assembly language programs.
- The DEBUG utility enables us to have the control of the processor resources and memory in the computer (PC) which uses one of the INTEL processor as the CPU up to some extent as most of them run under the control of operating system.
- The DEBUG enables us to use the PC as a low-level 8086 microprocessor kit. By typing the DEBUG command at DOS prompt and pressing enter key invokes the debugging facility. A '-'(dash) appears after the successful invoke operation of DEBUG as shown below:
- C:\> DEBUG

Writing Assembly Language Programs While Using MASM

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- Now by typing ‘R’ at the ‘-’ line and pressing enter key, we can see the content of different registers and flags present in the CPU of the PC as shown below:
- -R
- AX=0000H BX=0005H CX=000DH DX=S000H SP=8500H
  BP=9800H SI=2000H DI=7000H
- DS=S000H ES=3000H SS=4000H CS=2000H
- IP = 2000H FLAGS= 0024H
- The remaining DEBUG commands can be seen in any book which discusses assembly language programming in personal computer (PC). In the following section, few examples for writing 8086 assembly language program while using assembler are given.

Write a program to add two 8-bit data (F0H and 50H) in 8086 and to store result in memory, when assembler is used

```
. ASSUME CS: CODE, DS: DATA
DATA    SEGMENT                    ; Beginning of data segment
OPER1   DB  F0H            ; First operand
OPER2   DB  50H            ; Second operand
RESULT  DB  01 DUP (?)         ; A byte of memory is reserved for result
CARRY   DB  01 DUP (?)         ; A byte is reserved for storing carry
DATA    ENDS                       ; End of data segment
 CODE        SEGMENT               ; Beginning of code dement
START:  MOV AX, DATA       ; Initialize AX with the segment address of DS
MOV DS, AX             ; Move AX content to DS
MOV BX, OFFSET OPER1 ; Move the offset address of OPER1 to BX
MOV AL, [BX]           ; Move first operand to AL
ADD AL, [BX+1]         ; Add second operand to AL
MOV SI, OFFSET RESULT      ; Store offset address of RESULT in SI
```

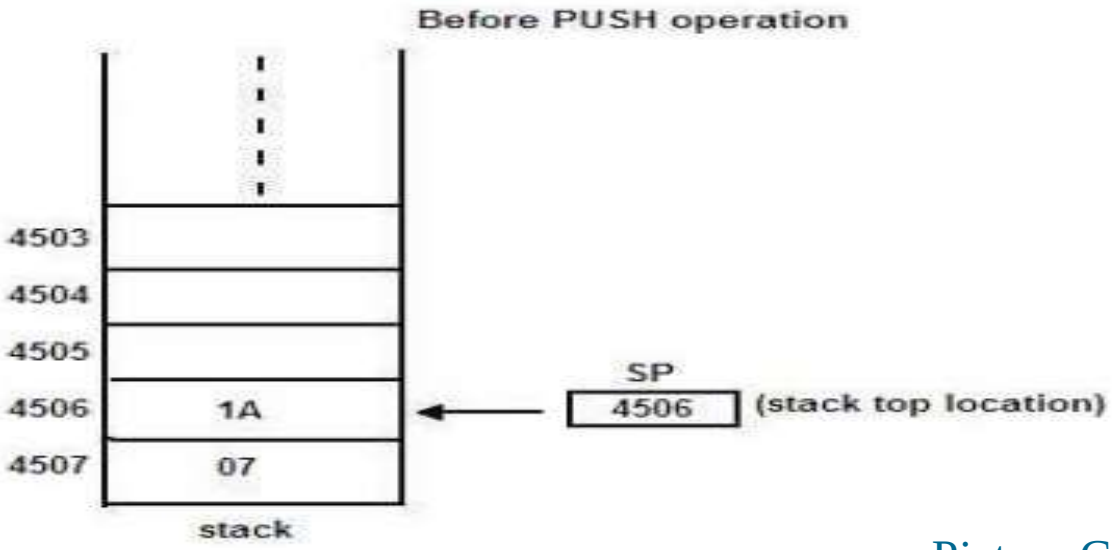
Contd….

- MOV [SI], AL ; Store content of AL in the location RESULT
- INC SI ; Increment SI to point location of carry
- JC CAR ; If carry =1, go to the place CAR
- MOV [SI], 00H ; Store 00H in the location CARRY
- JMP LOC1 ; go to the place LOC1
- CAR: MOV [SI], 01H ; Store 01H in the location CARRY
- LOC1: MOV AH, 4CH
- INT 2IH ; Return to DOS prompt
- CODE ENDS ; End of code segment
- END START ; Program ends
- In the above program, the instructions MOV AH, 4CH and INT 2IH at the end of the program are used for returning to the DOS prompt after executing the program in the computer. If instead of these two instructions, if one writes HLT instruction, the computer will hang after executing the program as the CPU goes to halt state and the user will not be able to examine the result.

Session – 12

- Stack structure and related programming

Stack Structure



Picture Courtesy :Google

- Stack is a set of memory locations in the Read/Write memory which is used for temporary storage of binary information during the execution of a program.
- It is implemented in the Last-in-first-out (LIFO) manner. i.e., the data written first can be accessed last, One can put the data on the top of the stack by a special operation known as PUSH.
- Data can be read or taken out from the top of the stack by another special instruction known as POP.

Stack Structure

- Stack is implemented in two ways. In the first case, a set of registers is arranged in a shift register organization.
- One can PUSH or POP data from the top register. The whole block of data moves up or down as a result of push and pop operations respectively.
- In the second case, a block of RAM area is allocated to the stack. A special purpose register known as stack pointer (SP) points to the top of the stack.
- Whenever the stack is empty, it points to the bottom address. If a PUSH operation is performed, the data are stored at the location pointed to by SP and it is decremented by one. Similarly if the POP operation is performed, the data are taken out of the location pointed at by SP and SP is incremented by one. In this case the data do not move but SP is incremented or decremented as a result of push or pop operations respectively

Stack Structure

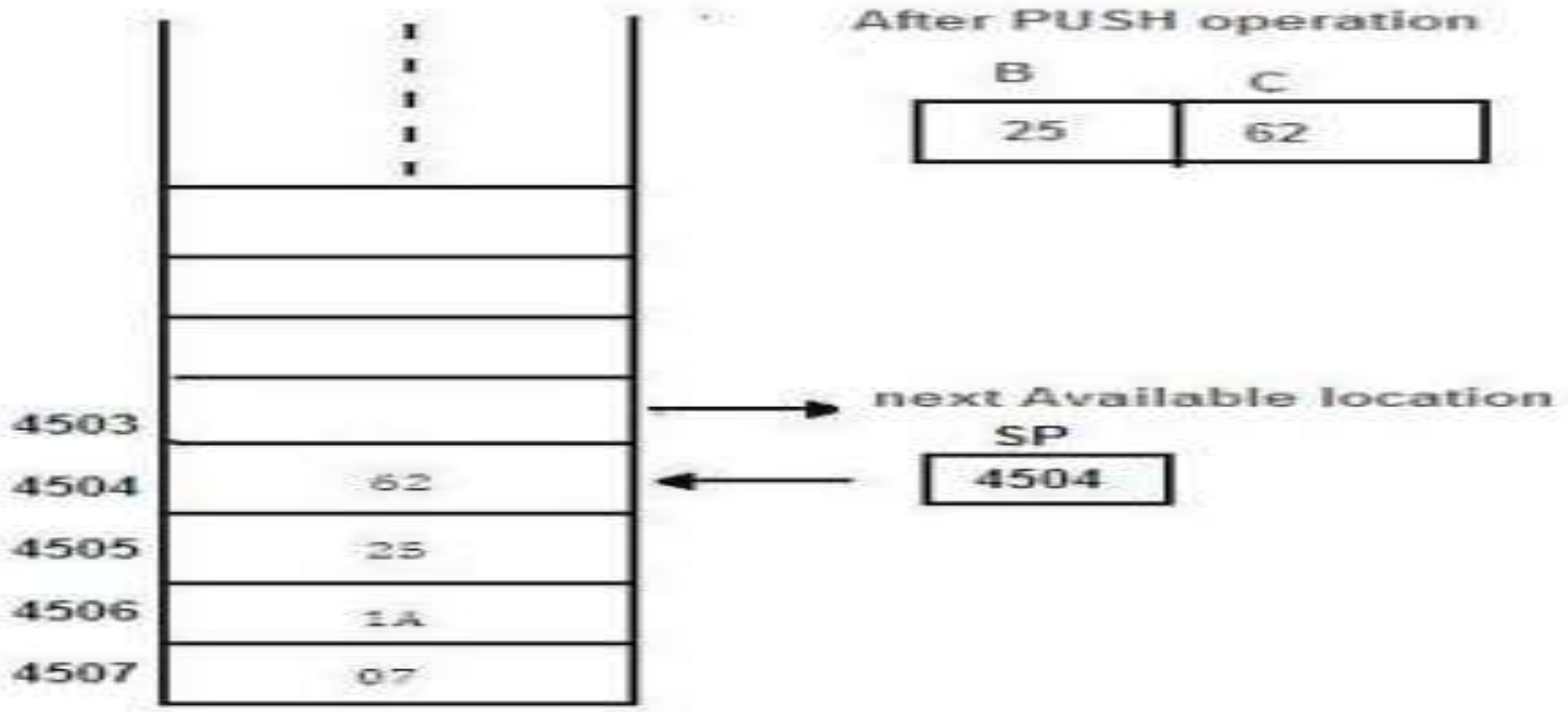
- Application of Stack: Stack provides a powerful data structure which has applications in many situations. The main advantage of the stack is that, We can store data (PUSH) in it with out destroying previously stored data.
- This is not true in the case of other registers and memory locations. stack operations are also very fast The stack may also be used for storing local variables of subroutine and for the transfer of parameter addresses to a subroutine.
- This facilitates the implementation of re-entrant subroutines which is a very important software property. The disadvantage is, as the stack has no fixed address, it is difficult to debug and document a program that uses stack.

Stack Operation

- Stack operation: Operations on stack are performed using the two instructions namely PUSH and POP. The contents of the stack are moved to certain memory locations after PUSH instruction.
- Similarly, the contents of the memory are transferred back to registers by POP instruction.

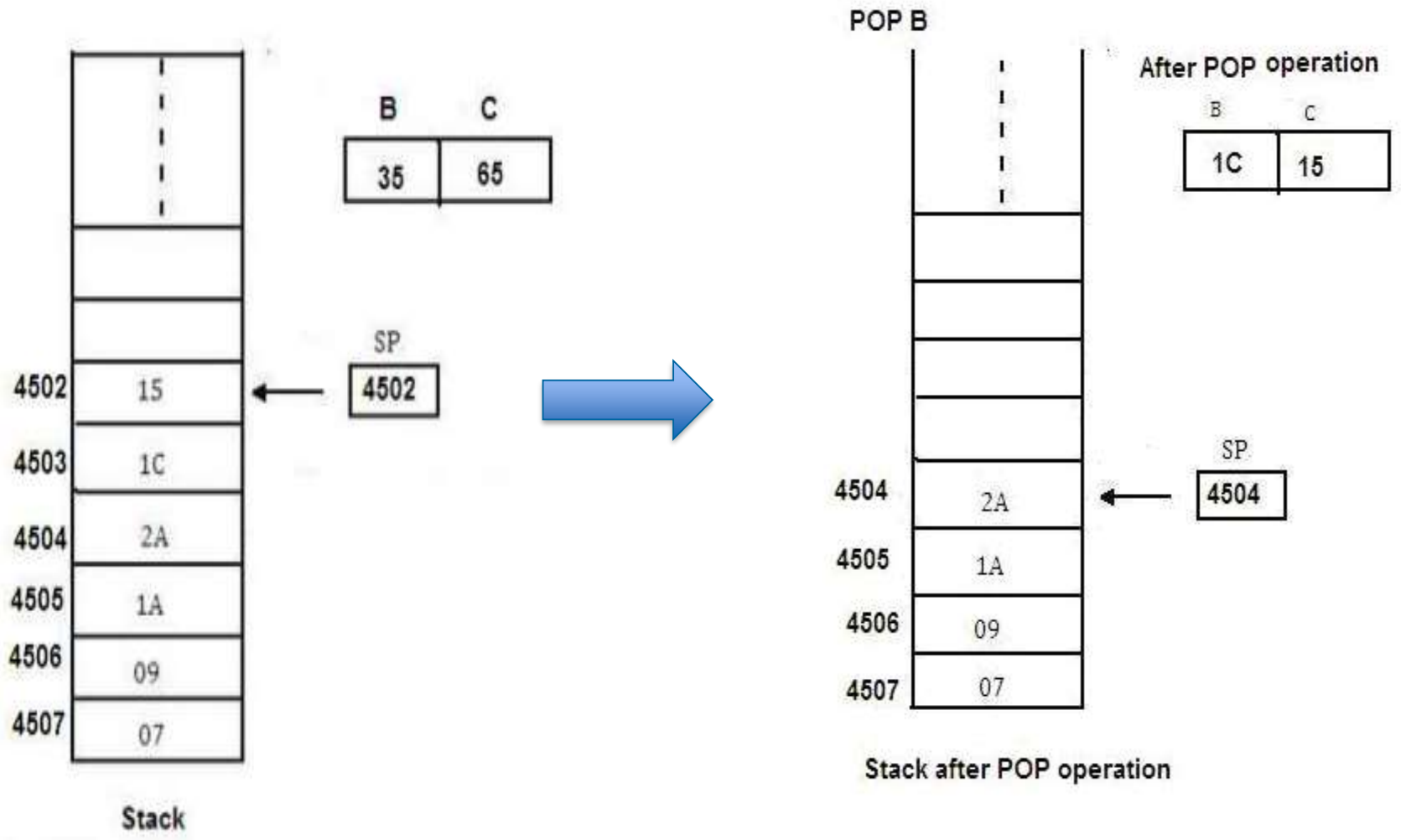
The PUSH operation of the Stack

- Let us consider two registers (register pair) B & C whose contents are 25 & 62.



Picture Courtesy :Google

The POP operation of the Stack



Picture Courtesy :Google

Example program

- **Write a program to initialize the stack pointer (SP) and store the contents of the register pair HL on stack by using PUSH instruction. Use the contents of the register pair for delay counter and at the end of the delay retrieve the contents of H-L using POP.**

Memory Location	Label	Mnemonics	Operand	Comments
8000		LXI	SP, 4506 H	Initialize Stack pointer
8003		LXI	H,2565 H	
8006		PUSH	H	
8007		DELAY	CALL	Push the contents.
. . . 8.00A		. . . POP	. . . H	

Picture Courtesy :Google

Session – 13

- Interrupt Structure and related programming

Interrupts

- **The meaning of 'interrupts' is to break the sequence of** operation
- While the CPU is executing a program, an interrupt breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR)
- After executing ISR , the control is transferred back again to the main program 8086 Interrupts and Interrupt Response

Types of Interrupt

An 8086 interrupt can come from any one of the three sources

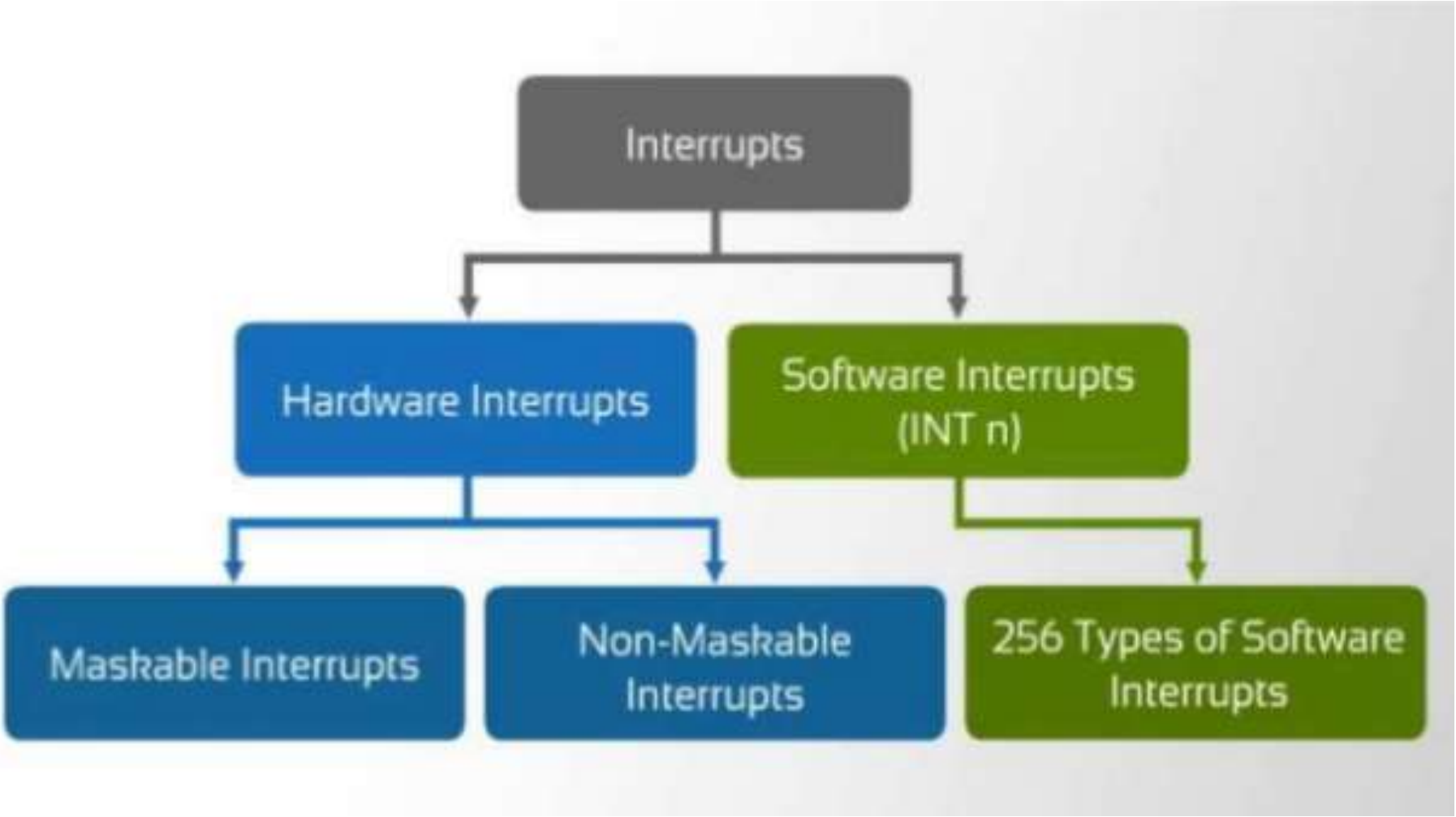
- One source is an external signal applied to the Non Maskable Interrupt (NMI) input pin or to the Interrupt (INTR) input pin
- An interrupt caused by a signal applied to one of these inputs is referred to as a hardware interrupt

A second source of an interrupt is execution of the Interrupt instruction, INT, This is referred to as Software Interrupt

The third source of an interrupt is some error condition produced in the 8086 by the execution of an instruction

- An example of this is the divide by zero error

Types of Interrupt



Picture Courtesy :Google

Types of Interrupt

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

TYPES OF INTERRUPT

- **SOFTWARE INTERRUPTS:** There are instructions in 8086 which cause an interrupt.
  - INT instructions with type number specified.
  - INT 3, Break Point Interrupt instruction.
  - INTO, Interrupt on overflow instruction.

- **HARDWARE INTERRUPTS:**

The primary sources of interrupts, however, are the PCs timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices.

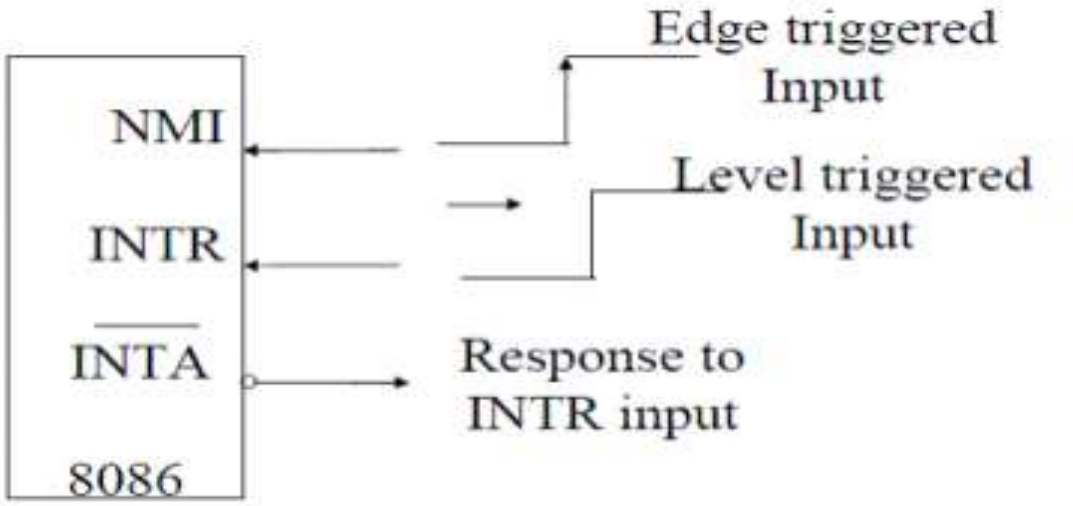
Picture Courtesy :Google

Hardware Interrupt

(i) *Hardware Interrupts (External Interrupts)-*

The Intel microprocessors 8086 **support hardware interrupts** through:

- Two pins that allow interrupt requests, -INTR and NMI
- However one pin that acknowledges, INTR is INTA.



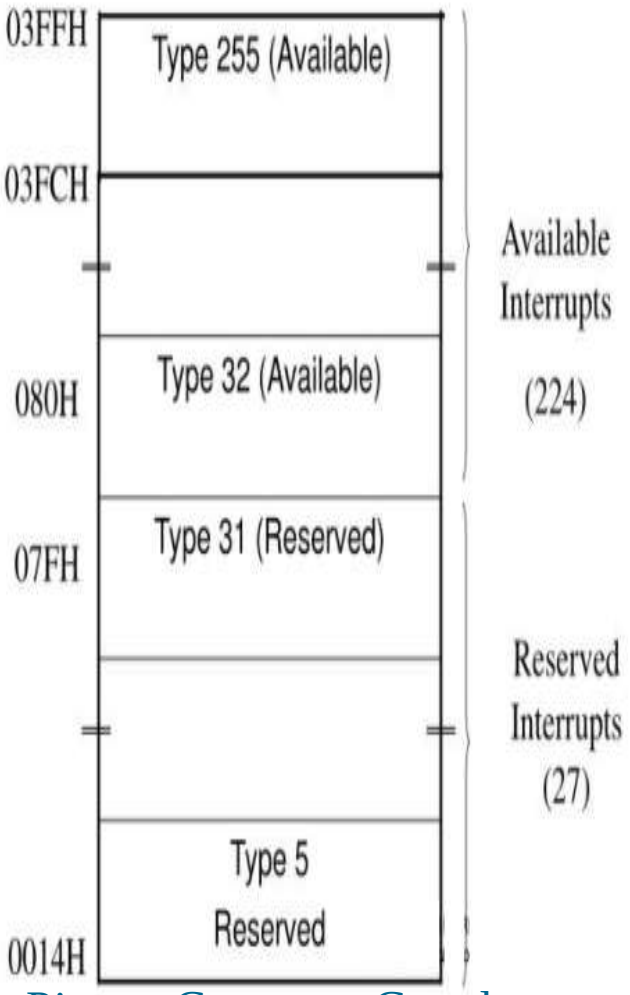
INTR and NMI

Picture Courtesy :Google

Software Interrupts

- The interrupt vector table contains 256 four byte entries, contains the CS:IP
- Interrupt vectors for each of the 256 possible interrupts. The table is used to locate the interrupt service routine addresses for each of those interrupts.
- The Interrupt vector table is located in the first 1024 bytes of memory at addresses 000000H-0003FFH.It contains the address(segment and offset)of the interrupt service provider

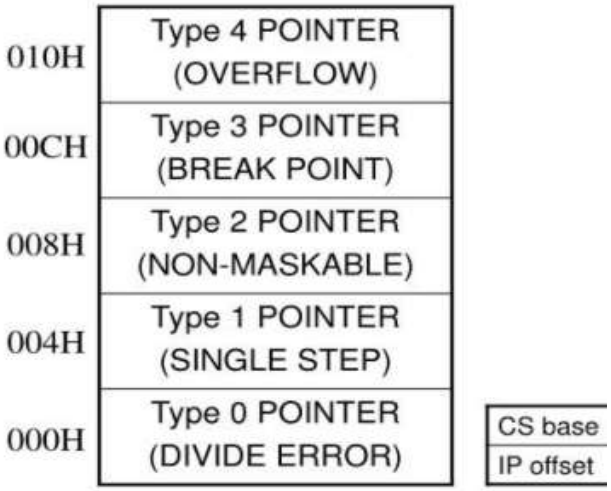
Software Interrupts

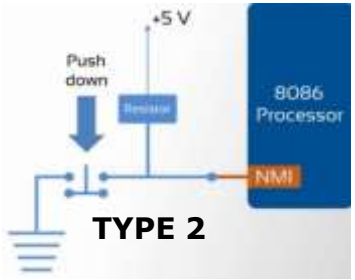


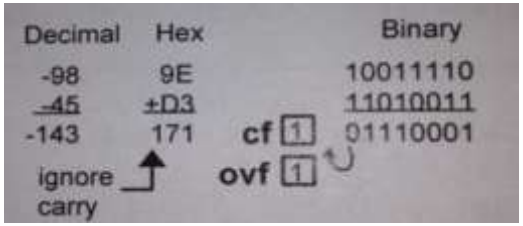
Picture Courtesy :Google

- The first five interrupt vectors are identical in all Intel processors
- Intel reserves the first 32 interrupt vectors
- The last 224 vectors are user available
- Each is four bytes long in real mode and contains the starting address of the interrupt service procedure.
- The first two bytes contain the offset address The last two contain the segment address

Types of Interrupts







TYPE 4

- TYPE 0 The divide error : whenever the results from a division overflows or an attempt is made to divide by zero.
- Type 2 The non-maskable interrupt occurs when a logic 1 is placed on the NMI input pin to the microprocessor. non- maskable—it cannot be disabled
- Type 3 A special one-byte instruction (INT 3) that uses this vector to access its interrupt-service procedure. often used to store a breakpoint in a program for debugging
- TYPE 4 Overflow is a special vector used with the INTO instruction. The INTO instruction interrupts the program if an overflow condition exists.

Picture Courtesy :Google

Types of Interrupts

- **TYPE 5 The BOUND** instruction compares a register with boundaries stored in the memory. If the contents of the register are greater than or equal to the first word in memory and less than or equal to the second word, no interrupt occurs because the contents of the register are within bounds. if the contents of the register are out of bounds, a type 5 interrupt ensues as reflected by the overflow flag (OF)

- **Type 7** The coprocessor not available interrupt occurs when a coprocessor is not found, as dictated by the machine status word (MSW or CR0) coprocessor control bits. if an ESC or WAIT instruction executes and no coprocessor is found, a type 7 exception or interrupt occurs

Types of Interrupts

- Type 8 A double fault interrupt is activated when two separate interrupts occur during the same instruction.
- In computing, a double fault is a serious type of error that occurs when a central processing unit (CPU) cannot adequately handle a certain type of system event that requires the CPU's immediate attention. Double faults may cause computer crashes and error messages, automatic restarting of the machine, and the loss of any unsaved data. They are often caused by problems in the computer's hardware such as a bad memory module or overheating CPU.

Types of Interrupts

- Type 9 The coprocessor segment overrun occurs if the ESC instruction (coprocessor opcode) memory operand extends beyond offset address FFFFH in real mode.
- Type 10 An invalid task state segment interrupt occurs in the protected mode if the TSS is invalid because the segment limit field is not 002BH or higher. usually because the TSS is not initialized
- Type 11 The segment not present interrupt occurs when the protected mode P bit (P = 0) in a descriptor indicates that the segment is not present or not valid.

Types of Interrupts

- Type 12 A stack segment overrun occurs if the stack segment is not present (P = 0) in the protected mode or if the limit of the stack segment is exceeded.
- Type 13 The general protection fault occurs for most protection violations in 80286–Core2 in protected mode system. These errors occur in Windows as general protection faults. A list of these protection violations follows.

Types of Interrupts

- Type 13 PROTECTION VIOLATIONS
  - (a) Descriptor table limit exceeded
  - (b) Privilege rules violated
  - (c) Invalid descriptor segment type loaded
  - (d) Write to code segment that is protected
  - (e) Read from execute-only code segment
  - (f) Write to read-only data segment
  - (g) Segment limit exceeded
  - (h) CPL = IOPL when executing CTS, HLT, LGDT, LIDT, LLDT, LMSW, or LTR o (i) CPL > IOPL when executing CLI, IN, INS, LOCK, OUT, OUTS, and STI (cont.)

Types of Interrupts

- Type 14 Page fault interrupts occur for any page fault memory or code access in 80386, 80486, and Pentium–Core2 processors. Type 16 Coprocessor error takes effect when a coprocessor error (ERROR = 0) occurs for ESCape or WAIT instructions for 80386, 80486, and Pentium–Core2 only.
- Type 17 Alignment checks indicate word and double word data are addressed at an odd memory location (or incorrect location, in the case of a double word). interrupt is active in 80486 and Pentium–Core2
- Type 18 A machine check activates a system memory management mode interrupt in Pentium– Core2.

Summary

- The addressing modes in 8086 are classified into Register, Immediate, Data memory, Stack memory and program memory addressing modes.
- The different instructions in 8086 are classified into Data Transfer, Arithmetic, Logical, Shift, Rotate, Flag manipulation, Control Transfer, String and Machine Control instructions.
- The assembly language programming of 8086 can be done with line assembler or assembler.
- Assembler directives are used while writing assembly language program of 8086 which is to be assembled by using an assembler.

Key terms in Module

- Addressing mode is the way by which the microprocessor addresses the operands while fetching data during execution of an instruction or the way by which the microprocessor calculates the memory address, from where the next instruction to be executed is taken, in the case of jump or call instructions.
- Instruction set of 8086 consists of Data Transfer, Arithmetic, Logical, Shift, Rotate, Flag manipulation, Control Transfer, String and Machine Control Instructions.
- Assembler is software which is used to convert assembly language program into machine language program.
- Line assembler converts each line in assembly language program into corresponding machine language program immediately as it is entered in the system.
- Assembler directives are commands to the assembler to give various details in a program such as the required storage class for a particular constant or a variable (byte, word or double word), logical name of the segments (CODE or STACK or DATA segment), type of different procedures or routines (FAR or NEAR or PUBLIC or EXTRN), end of a segment (ENDS) and defining a macro (MACRO, ENDM), etc.

Learning Resources

1. K. M. Bhurchandi and A. K. Ray, *Advanced Microprocessors and Peripherals-with ARM and an Introduction to Microcontrollers and Interfacing*, 3rd edition, Tata McGraw Hill, 2015.

2. Yu-cheng Liu, Glenn A. Gibson, *Microcomputer systems: The 8086/8088 family-Architecture, programming and design,* 2nd edition, Prentice Hall of India, 2007.

Thank You