VIT-AP UNIVERSITY

Computer Organization & Architecture

ECE 2002

DR.S SARITHA

ASSISTANT PROFESSOR

SENSE

26-07-2022

Lecture - 0

Course Objectives

- Introduction and overview of basic computer organization. Computer arithmetic: binary, hexadecimal and decimal number conversions, binary number arithmetic and IEEE binary floating point number standard.
- To learn basic computer logic: gates, combinational circuits, sequential circuits, adders, ALU, SRAM and DRAM.
- To learn basic assembly language programming, basic Instruction Set Architecture (ISA), and the design of single cycle CPU.

Outcomes Expected

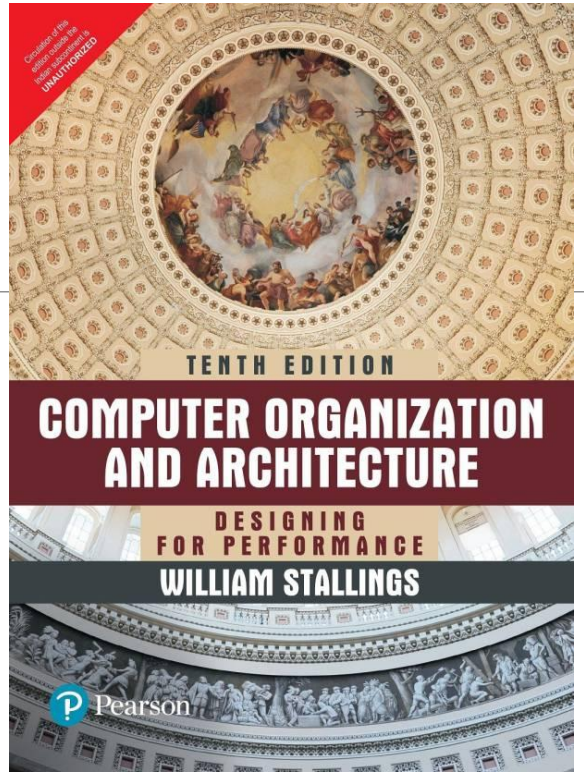
- Apply different formats of data representation and number systems
- Use Boolean algebra as related to designing computer logic, including solving Karnaugh maps
- Design and evaluate combinational and sequential logic circuits with multiple inputs and outputs
- Design simple combinational and sequential logic circuits, using a small number of logic gates
- Assemble a simple computer with hardware design including data format, instruction format, instruction set, addressing modes, bus structure, input/output, memory, Arithmetic/Logic unit, control unit, and data, instruction and address flow
- Design simple assembly language programs that make appropriate use of a registers and memory

Module 3: I/O Organization

- Microprocessors
- Instruction format
- Instruction set
- Addressing modes
- Assembly Language Programming
- Stack
- Subroutine
- Interrupt
- Accessing I/O devices
- Standard I/O Interfaces- RS-232C, IEEE-488, USB
- Interfacing techniques

Text Book:

William Stallings, Computer Organization and Architecture: Designing for Performance, Pearson Education, Tenth Edition,2013



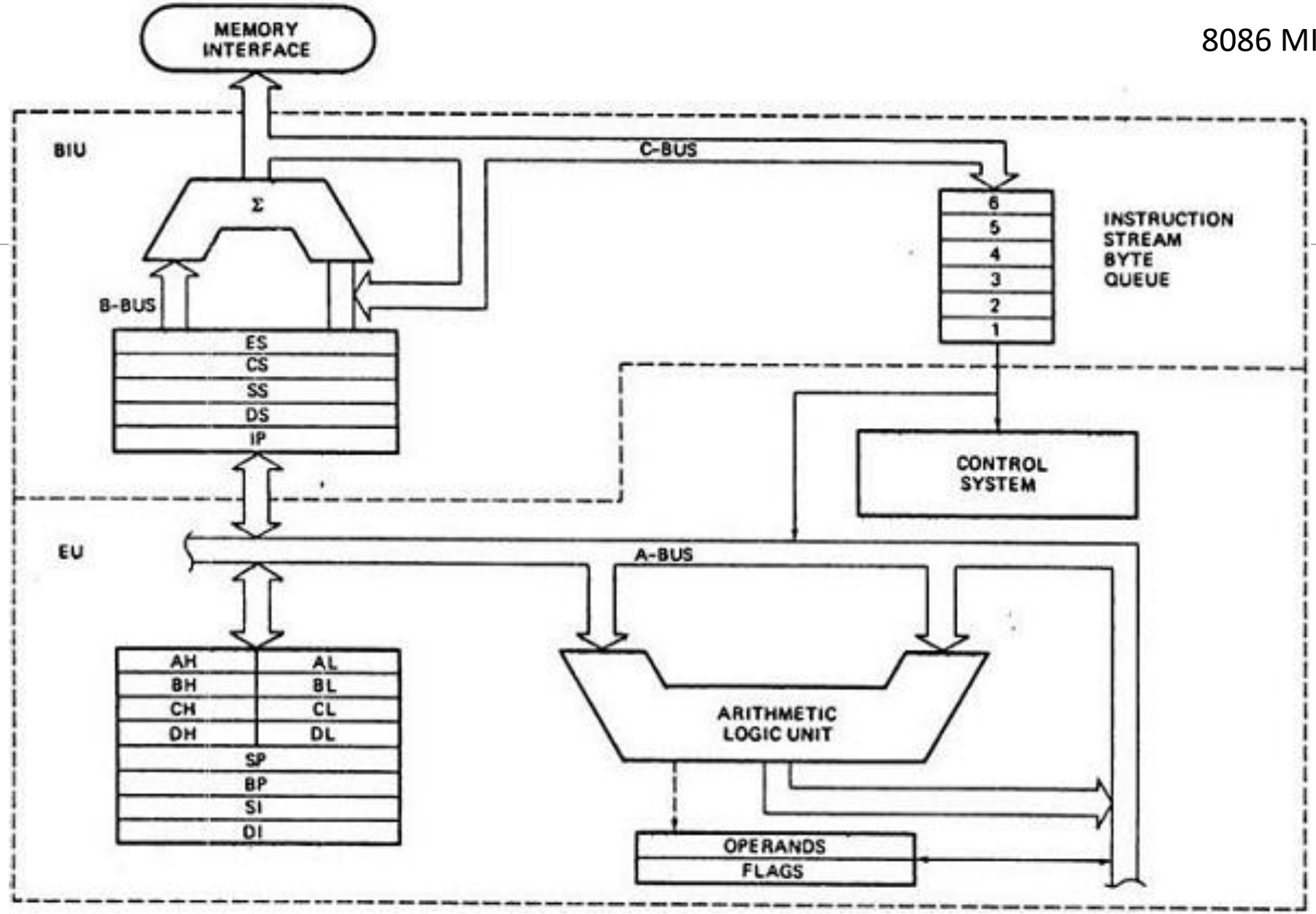
Books:

1. M. Morris Mano, Rajib Mall, Computer System Architecture, Pearson Education Third Edition,2017.
2. Carl Hamacher, Zvonkovranesic, Safwat Zaky , Computer Organization, McGraw Hill, Fifth Edition,2011.

8086 microprocessor

- The 8086 Microprocessor has two units

1. Bus Interface Unit
2. Execution Unit


MEMORY INTERFACE
BIU
C-BUS
Σ
B-BUS
ES
CS
SS
DS
IP
6
5
4
3
2
1
INSTRUCTION STREAM BYTE QUEUE
CONTROL SYSTEM
EU
A-BUS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
ARITHMETIC LOGIC UNIT
OPERANDS
FLAGS

Bus interface Unit

- The Bus Interface Unit consists of different units such as the Instruction Queue, Segment Registers, Instruction Pointer, Address adder.
- It interfaces the processor to the outside → performing all the all external bus operations like fetch, read, write, input and output of data.
- The BIU uses instruction queue for pipelined instructions → 6-byte First-in-First-out register.
- It provides a full 16-bit bidirectional data bus and 20-bit address bus.
- Specifically, it performs Instruction fetch, Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement a pipeline architecture.

Execution Unit

- The Execution Unit consists of the following units such as Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.
- The EU decodes and executes the instructions fetched by the BIU.
- It extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- If the BIU is already in the process of fetching an instruction when the EU requests it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.
- It also tests the status and control flags and updates these flags based on the results of the instruction.

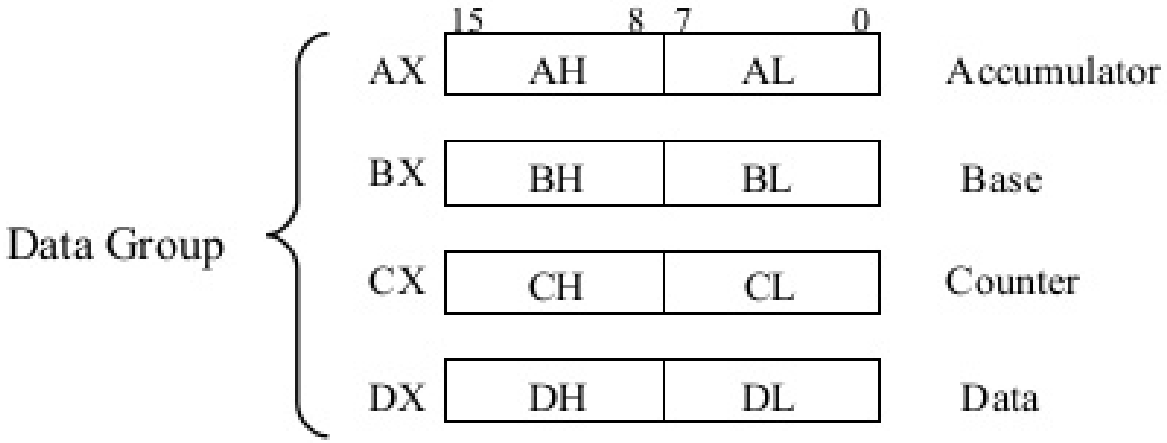
Register organization

- 8086 has a powerful set of registers of 16-bit each
- The registers are categorized into 4 groups

1. General data registers
2. Segment registers
3. Pointer and index registers
4. Flag register

General Registers

- 8086 contains 4 general data registers AX, BX, CX, and DX.
- They are used to hold data, variables, results etc., temporarily for faster operation.



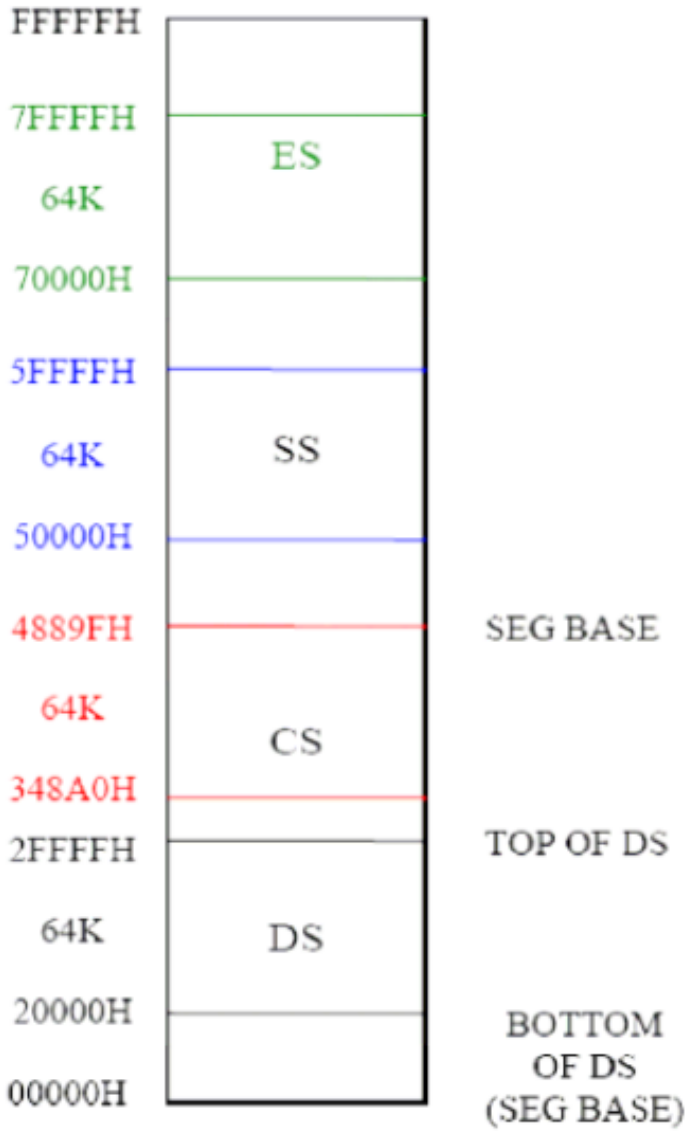
General Registers

- AX is used as a 16-bit accumulator, with the lower 8-bits designated as AL and higher 8-bits as AH for 8-bit operations.
- It performs all the arithmetic and logic operations and it is also used to store the result of any operation.
- BX register is used as a general purpose register as well as to store the offset for forming physical address in certain addressing modes.
- CX register is used as a default counter in case of string and loop instructions.
- It is also used for the count of the number of bits by which the contents of an operand must be shifted or rotated during the execution of the multibit shift or rotate instructions.
- DX register is used in I/O operations to hold the address of the I/O port.
- DX register also holds the remainder after a word division and holds the high-order bits (MSB) of the result after a word multiplication (32-bit).

Segment Registers

- There are 4 segment registers

1. Code segment
2. Data segment
3. Stack segment
4. Extra segment



Segment Registers

- Code segment (CS) register is used to address a memory location in the code segment of memory where the executable program or instructions are stored
- Stack segment (SS) register is used for addressing stack segment of memory which is used to store stack data.
- The data segment (DS) register points to the data segment of the memory where the data is stored
- The extra segment (ES) register points to the extra segment of the memory. This is used as another data segment for extra data storage.

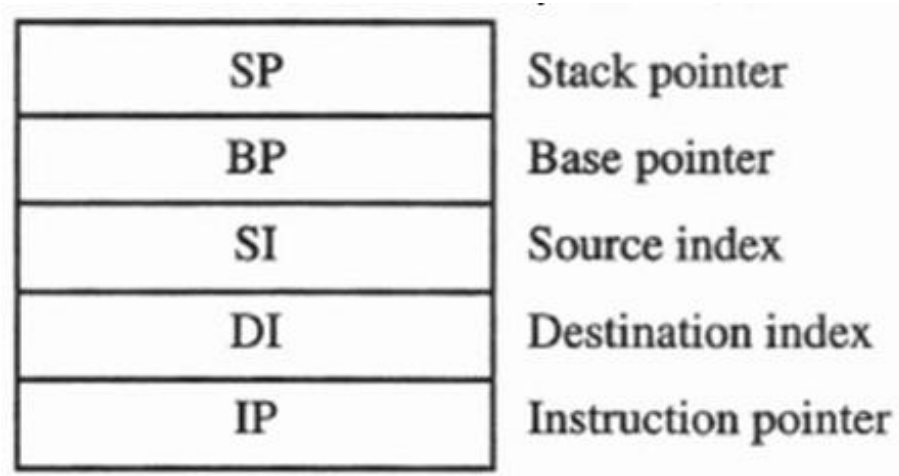
Pointers and index Registers

- There are 3 pointers in 8086

1. Instruction pointer (IP)
2. Base pointer (BP)
3. Stack pointer (SP)

- There are two Index registers

1. Source index (SI)
2. Destination index (DI)



Pointers and index Registers

- The function of IP is similar to a program counter, but it contains the offset address instead of the actual address of the next instruction.
- It contains the offset address within the code segment and the IP is combined with the CS register to generate the actual address of the next instruction to be executed.
- Stack pointer also contains the offset value which is added to the SS register to obtain the actual address of the stack segment.
- Base pointer is similar to the SP since it also contains the offset value pointing to the stack segment.

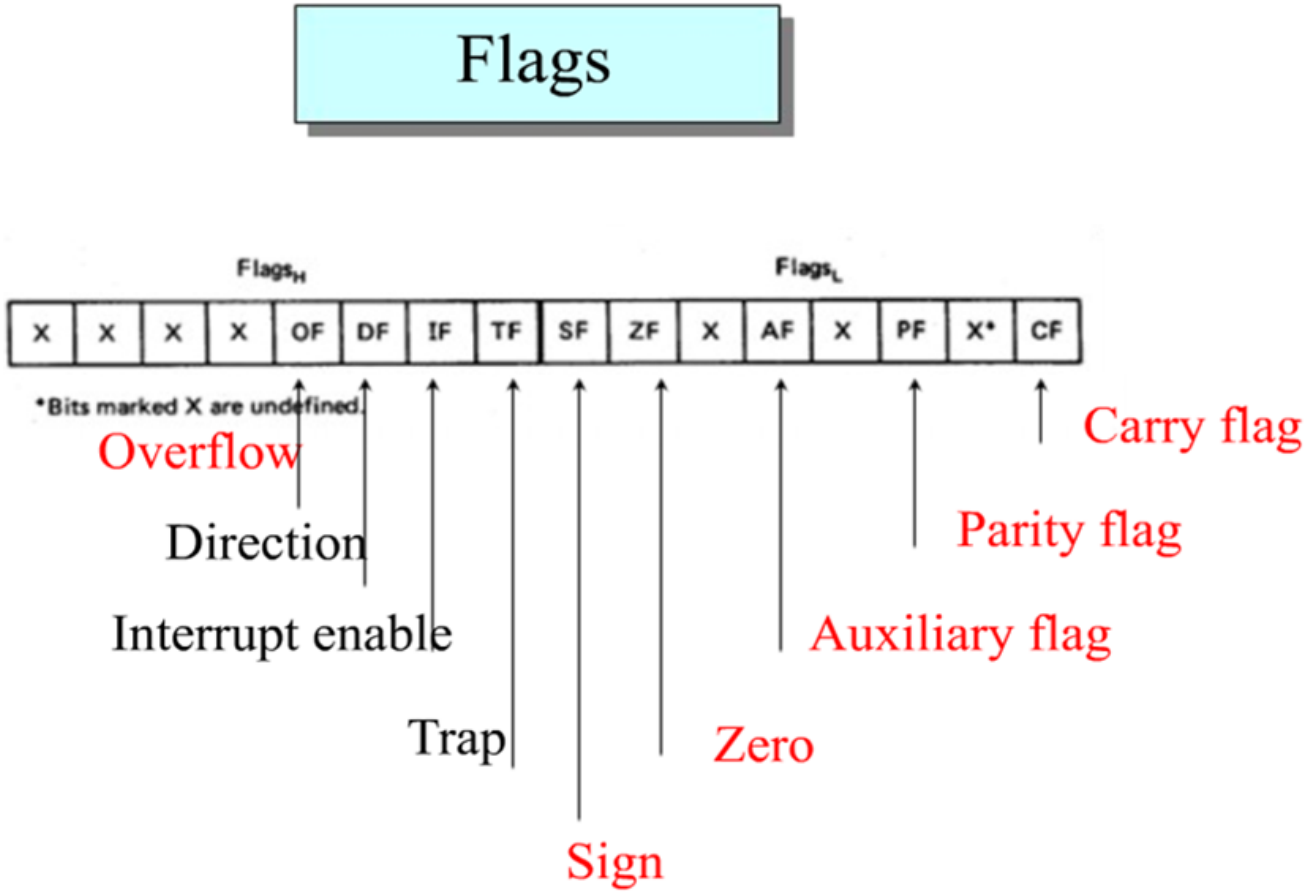
Pointers and index Registers

- The index registers are used as general purpose registers as well as for offset storage purpose.
- The source index register is used to store the offset of the source data in the data segment.
- And the destination index register is used to store the offset of the destination data in the data or extra segment.

Default segment and offset Registers

Segment	Offset
CS	IP
SS	SP or BP
DS	BX, SI or DI
ES	DI

FLAG Register



FLAG Registers

- CF is set when there is carry from the MSB in case of addition or a borrow in case of subtraction i.e. carry from D15 bit for 16-bit operations and D7 bit for 8-bit operation.
- Parity flag is set when the ALU output has an even parity and is reset when the ALU output has odd parity.
- Auxiliary carry flag is set when there is carry after addition and borrow after subtraction between D3 and D4 bit positions (for 8-bit data) and carry from D7 to D8 (for 16-bit data).
- Zero flag shows the result of the ALU operation is zero or non-zero.
- The sign flag is set when the result of the ALU operation is negative i.e. sign bit D15 is 1 (for 16-bit operations) or sign bit D7 is 1 (for 8-bit operations).

FLAG Registers

- Overflow flag is based on the (n-1)th bit carry of the ALU result.
- Overflow occurs when signed numbers are added or subtracted.
- For 8-bit operation, if there is carry from the D6 bit to the D7 bit, then the overflow flag is set
- Similarly, for 16-bit operation, if there is carry from the D14 bit to the D15 bit, then the overflow flag is set
- Trap flag is when the processor enters into single step mode or else it is reset.
- In single step mode, the processor executes one instruction at a time and it is useful for debugging programs.

FLAG Registers

- Interrupt flag is set when an maskable interrupt or INTR is received by the processor.
- Direction flag is used for string manipulation instructions i.e. the direction flag selects the increment or decrement mode for DI and SI registers in string instructions
- If DF = 1; then the registers are automatically decremented, or else DF = 0 then the registers are incremented.

Addressing modes

- Addressing mode is the way of locating the data or operands, the types of operands used and the way they are accessed for executing an instruction.
- Based on the flow of instructions, the instructions in 8086 can be categorized as

1. Sequential control flow instructions
2. Control transfer instructions

- Sequential control flow are the instructions in which after execution, the control is transferred to the next instruction appearing immediately after it in the program. Eg. Arithmetic instructions, logical, data transfer, and processor control instructions.
- Control transfer instructions transfer their control to some predefined address after their execution. Eg. INT, CALL, RET, and JUMP instructions.

Addressing modes

1. Register addressing mode
2. Immediate addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Register relative addressing mode
6. Indexed addressing mode
7. Based indexed addressing mode
8. Relative based indexed addressing mode

1. Intrasegment direct addressing mode
2. Intrasegment indirect addressing mode
3. Intersegment direct addressing mode
4. Intersegment indirect addressing mode

Sequential Control flow modes

1. Register addressing mode
2. Immediate addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Register relative addressing mode
6. Indexed addressing mode
7. Based indexed addressing mode
8. Relative based indexed addressing mode

Immediate addressing mode

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example

```
MOV CX, 4929 H, ADD AX, 2387 H,  MOV AL, FFH
```

Register addressing mode

It means that the register is the source of an operand for an instruction.

Example

```
MOV CX, AX   ; copies the contents of the 16-bit AX register into
          ; the 16-bit CX register),
ADD BX, AX
```

Register addressing mode

- In this mode, both the operands are specified by registers.
- Eg. MOV AX, BX
- All the registers can be used in this mode.
- Both the source and destination registers should be of the same size
- A segment to segment movement of data is not allowed

Immediate addressing mode

- In this mode, the source operand is specified as immediate data byte or word and the destination is either a register or a memory location.
- Eg. MOV AL, 22H; MOV BX, 3456H
- All the registers can be used in this mode.
- Both the source and destination should be of the same size

Direct addressing mode

The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example

```
MOV AX, [1592H], MOV AL, [0300H]
```

Register indirect addressing mode

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example

```
MOV AX, [BX]  ; Suppose the register BX contains 4895H, then the contents
              ; 4895H are moved to AX
ADD CX, {BX}
```

Direct addressing mode

- In this mode, the source is a memory location and the destination is a register.
- Eg. MOV AL, [1234H]; MOV BX, [5000H]
- Here, a 16-bit memory address i.e. the offset address is directly specified in the instruction as a part of it.
- The content of the physical address which is formed from the offset address is the source data.

Register indirect addressing mode

- Register indirect addressing mode allows data to be addressed at any memory location using the offset registers: BP, BX, DI or SI
- DS is the default segment when the registers BX, DI or SI are used.
- SS is the default segment when the register BP is used.
- Eg. MOV AX, [BX]

Register relative addressing mode

- In this mode, the data in a segment of memory are addressed by adding an 8-bit or 16-bit displacement to the contents of a base register (BX or BP) or an index register (SI or DI).
- ES and DS are the default segments when the registers BX, DI or SI are used.
- SS is the default segment when the register BP is used.
- Eg. MOV AX, 1000H [BX]

Indexed addressing mode

- In this mode, the offset address of the operand is stored in one of the index registers like SI or DI.
- ES and DS are the default segments for DI or SI
- Eg. MOV AX, [SI]

Based-Indexed addressing mode

- In this mode, one base register (BX or BP) and one index register (SI or DI) are used to indirectly address memory.
- The effective address is formed by adding contents of a base register to the contents of the index register.
- Eg. MOV AX, [BX] [DI]

Relative Based-Indexed addressing mode

- It is similar to the based indexed mode, but it adds a displacement along with the base register and index register to form the memory address
- The effective address is formed by adding the 8-bit or 16-bit displacement with the addition result of the base register and the index register.
- Eg. MOV AX, 2000H [BX] [DI]

Based-index addressing mode

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

Example

```
ADD CX, [AX+SI], MOV AX, [AX+DI]
```

Based indexed with displacement mode

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement.

Example

```
MOV AX, [BX+DI+08], ADD CX, [BX+SI+16]
```

Question 1

The value of Code Segment (CS) Register is 4042H and the value of different offsets is as follows:
BX: 2025H , IP: 0580H , DI: 4247H
Calculate the effective address of the memory location pointed by the CS register.

Answer

The offset of the CS Register is the IP register.
Therefore, the effective address of the memory location pointed by the CS register is calculated as follows:

Effective address= Base address of CS register X 10_H + Address of IP

$= 4042_H \text{ X } 10_H + 0580_H$

$= (40420 + 0580)_H$

$= 409A0_H$

Question 2

Calculate the effective address for the following register:

SS: 3860H, SP: 1735H, BP: 4826H

Answer

Both SP and BP are the offsets for Stack Register (SS). The address calculated when BP is taken as the offset gives the starting address of the stack. The address when SP is taken as the offset denotes the memory location where the top of the stack lies.

Therefore, the effective address for both these cases is:

```
(SS X 10H) + SP = 3860H X 10H + 1735H
                = 38600H + 1735H
                = 39D35H

(SS X 10H) + BP = 3860H X 10H + 4826H
                = 38600H + 4826H
                = 3CE26H
```

Question 3

The value of the DS register is 3032H. And the BX register contains a 16 bit value which is equal to 3032H. 0008H is added to BX.

ADD BX, 0008H

The register AX contains some value which needs to be stored at a location as follows:

MOV [BX], AX

Calculate the address at which the value of the AX will be stored.

Q3: Answer

After executing the first instruction, the value of BX Register is as follows:

BX = 3040H

The BX register is an offset of the Data Segment (DS) register. So, the location at which the value of the AX register will be stored is calculated as follows:

```
(DS X 10H) + BX = 3032H X 10H +(3032+0008)H
               = 30320H + 303AH
               = 3335AH
```

Question 4

You are provided the following values:

DS: 3056H, IP: 1023H, BP: 2322H and SP: 3029H

Can you calculate the effective address of the memory location as per the DS register?

Q4: Answer

No, the effective address of the DS register cannot be calculated from the given values because none of the given offset is an offset of the DS Register. This can be done only in the case of segment override prefix, but as it is not mentioned here, we will not follow that.

Segment	Offset
CS	IP
SS	SP or BP
DS	BX, SI or DI
ES	DI

Control transfer instruction modes

1. Intrasegment direct addressing mode
2. Intrasegment indirect addressing mode
3. Intersegment direct addressing mode
4. Intersegment indirect addressing mode

- If the address location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode.
- If the destination lies in the same segment, the mode is called intrasegment mode

Intrasegment direct mode

- In this mode, the effective branch address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and it appears directly in the instruction as an immediate displacement value.
- The effective branch address is the sum of an 8-bit or 16-bit displacement in the current contents of IP.
- Eg. JMP [02](Eff. offset Addr = [IP] + [02])

Intrasegment indirect mode

- In this mode, the effective branch address is the contents of the register or memory location that is accessed using any one of the data addressing modes.
- The contents of the IP will be replaced by the effective branch address
- Eg. JMP BX (Eff. offset Addr = [IP] + [BX])
- In this instruction, the control is jumped to an address specified by the 16-bit register.
- The value of IP+BX is copied into IP with CS value unchanged.
- Then the physical address of the next instruction is obtained using the current content of CS and new value of IP

Intersegment direct mode

- This addressing mode is used to provide means of branching from one segment to another segment.
- Eg. JMP 2000H: 3000H
- i.e. the JMP instruction loads CS with 2000H and loads IP with 3000H

Intersegment indirect mode

- This addressing mode replaces the contents of the IP and CS with the contents of two consecutive words in memory that are addressed using indirect addressing.
- Eg. JMP [5000H] or JMP [BX or SI or DI]
- i.e. the contents of [5000H] & [5000H+1] in DS is loaded into IP and loads the contents of [5000H +2] & [5000H +3] in DS into CS.

8086 instructions

- Instructions are classified on the basis of the functions they perform

1. Data transfer instructions
2. Arithmetic and logical instructions
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag manipulation instructions
7. Shift and rotate instructions
8. String instructions

Data transfer instructions

- All the instructions which perform data movement comes under this group
- They are move, load, store, exchange, input, output, push and pop instructions
- The source of the data maybe a register, memory location, port etc. and the destination maybe a register, memory location or port
- Eg: MOV, XCHG, XLAT, PUSH, POP, LDS, LEA, LES, IN, OUT.

Types:

a) Instruction to transfer a word
b) Instructions for input and output port transfer
c) Instructions to transfer the address
d) Instructions to transfer flag registers

Instruction to transfer a word

- **MOV** - Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** - Used to put a word at the top of the stack.
- **POP** - Used to get a word from the top of the stack to the provided location.
- **PUSHA** - Used to put all the registers into the stack.
- **POPA** - Used to get words from the stack to all registers.
- **XCHG** - Used to exchange the data from two locations.
- **XLAT** - Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** – Used to read a byte or word from the provided port to the accumulator.
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.
- **LDS** – Used to load DS register and other provided register from the memory
- **LES** – Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

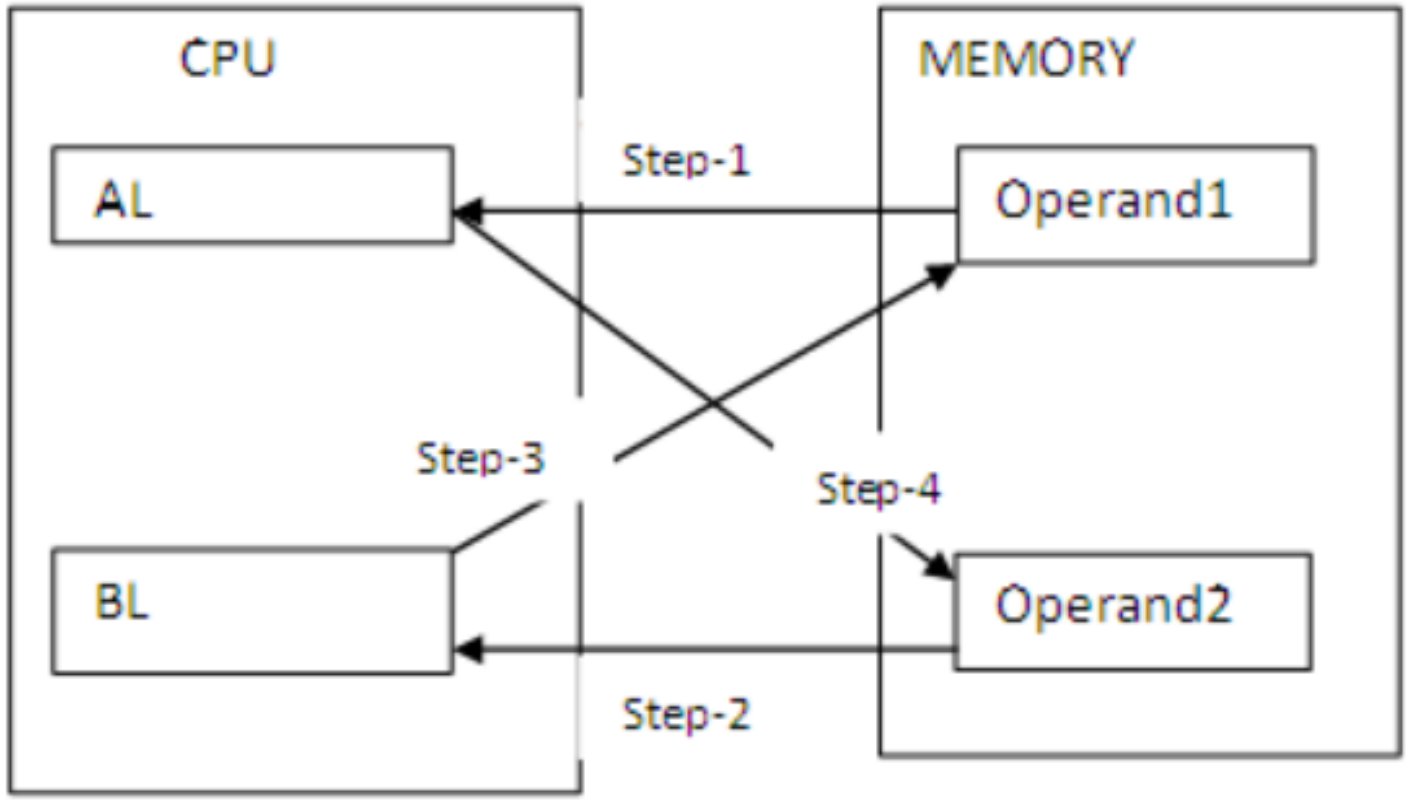
- **LAHF** – Used to load AH with the low byte of the flag register.
- **SAHF** – Used to store AH register to low byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.
- **POPF** – Used to copy a word at the top of the stack to the flag register.

MOV

- Move a byte or word to register or memory location
- The source can be a register, memory location or an immediate number, the destination can be a register or memory location.
- But both the source and destination cannot be memory locations.

Instruction	Format	Examples	Comments

MOV	MOV dest., src	MOV CX, 023Bh
		MOV AX, BX
		MOV AX, [SI]
		[BX]
		MOV BX, array
			Initialize CX with immediate number
			Transfers the content of BX into AX



Example Code

```
MOV AL, Operand1
MOV BL, operand2
MOV operand1,BL
MOV operand2,AL
```

MOV: Examples

- MOV AX, 1234H
- MOV AX, BX
- MOV AX, [SI]
- MOV AX, [1000H]
- MOV AX, 1234H [BX] [DI]

XCHG

- XCHG instruction exchanges two operands that are of same type i.e. byte type or word type.
- This instruction cannot exchange directly two memory locations, and also operands cannot be segment registers.

Instruction	Format	Examples	Comments
XCHG	XCHG dest., src	XCHG DX, AX XCHG AL,BL	Exchanges the content of AX and DX Exchanges the content of BL and AL

XCHG: Examples

- XCHG [5000H], AX
- XCHG BX, AX

XLAT

- This instruction is used to translate the byte in AL using a table in memory. i.e. it replaces the content of AL with the content from the top of the lookup table.
- DSx10H + BX + AL → AL

LEA

- This instruction is used to load the effective address in the register specified in the instruction.

Instruction	Format	Examples	Comments
LEA	LEA dest., src	LEA SI, ARRAY LEA BX, MyName	Load the 16 bit offset of array in reg. SI Loads 16 bit offset of MyName in reg. SI This instruction is equivalent to the instruction: MOV SI, offset ARRAY or MOV BX, offset MyName

LEA: Examples

- LEA BX, [DI]
- LEA BX, 6000H [SI]
- LEA SI, 44H [BP]
- LEA BX, [SI]
- LEA CX, 5000H [BX] [SI]

LES/LDS

- These two instructions are used to load the Data Segment(DS) or the Extra Segment (ES) registers and the register specified in the instruction from memory.

Instruction	Format	Examples	Comments
LDS	LDS dest., src	LDS SI, ARRAY	Loads DS and SI from locations starting from offset ARRAY: Load SI from offset ARRAY and ARRAY+1 and then loads DS from offset ARRAY+2 and ARRAY+3
LES	LES dest., src	LES BX, MyName	Loads ES and BX from locations starting from offset MyName: Load SI from offset MyName and MyName +1 and then loads DS from offset MyName +2 and MyName +3

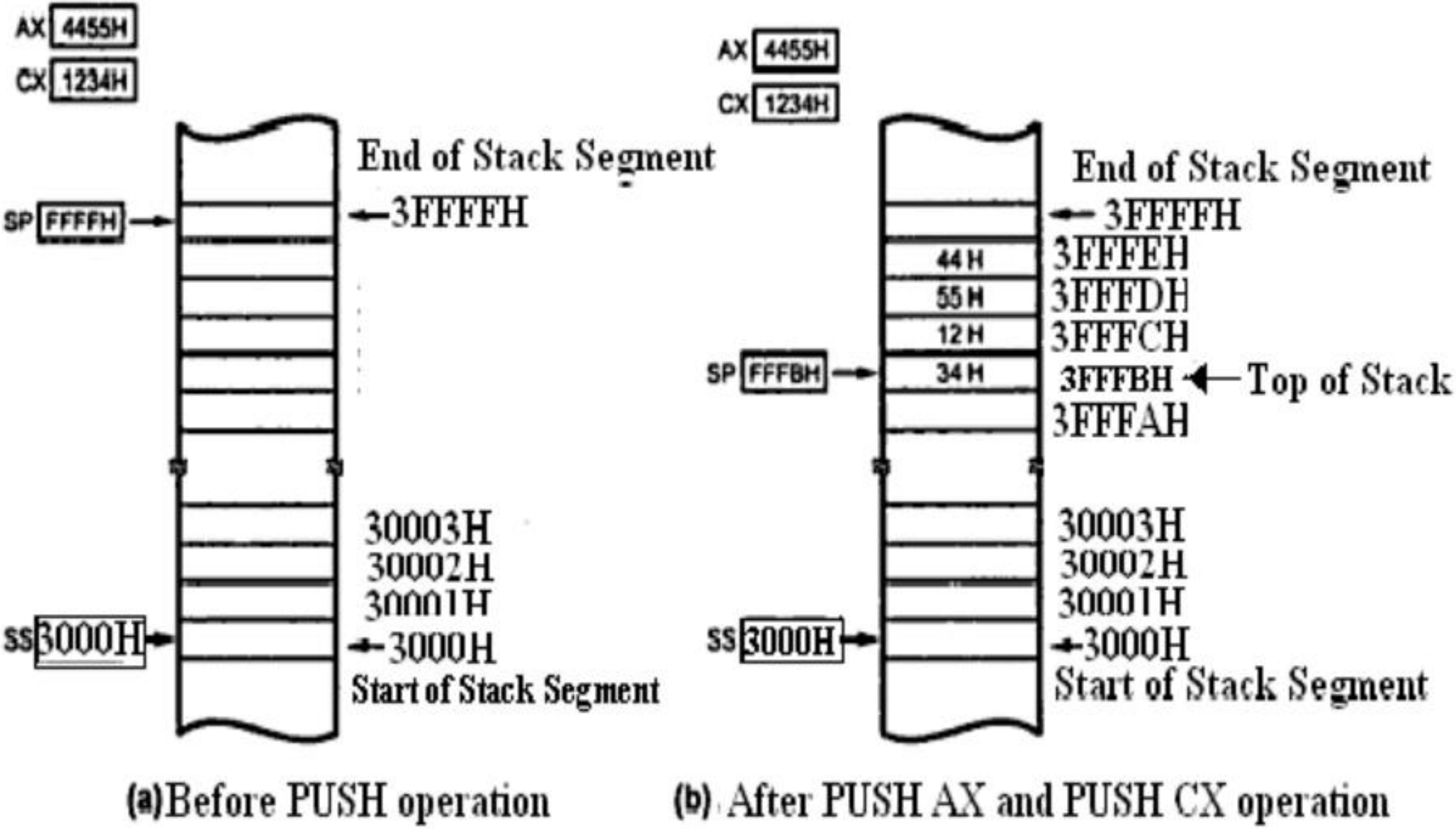
Stack operations

- Data is placed on the stack using the PUSH instruction and removed from the stack using the POP instruction.
- When an item is pushed onto the stack, the processor decrements the SP register, then writes the item at the new top of stack.
- When an item is popped off the stack, the processor reads the item from the top of stack, then increments the SP register.
- Therefore, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

PUSH/POP

- These instructions are used to copy a word on top of the stack or remove the word from top of the stack in the register specified.
- The operand in both (PUSH and POP) instructions can be a general purpose register, segment register(except CS) or a memory location.

Instruction	Format	Examples	Comments
PUSH	PUSH operand	PUSH BX	Copies the BH at SP-1 and BL at SP-2. Thus after the complete execution of PUSH instruction SP is decremented by 2, this new value (SP-2) is the new top of stack.
POP	POP operand	POP CX	Copies byte from the top of stack in CL and sets SP to SP+1, copies the byte from this location to CH and sets the SP to SP+1. Thus after the complete execution of POP instruction SP is increments by 2, this new value (SP+2) is the new top of stack.


AX 4455H
CX 1234H
End of Stack Segment
SP FFFFH
3FFFFH
30003H
30002H
30001H
SS 3000H
3000H
Start of Stack Segment
(a) Before PUSH operation
AX 4455H
CX 1234H
End of Stack Segment
3FFFFH
44H
3FFFEH
55H
3FFFDH
12H
3FFFCH
SP FFFBH
34H
3FFFBH
Top of Stack
3FFFAH
30003H
30002H
30001H
SS 3000H
3000H
Start of Stack Segment
(b) After PUSH AX and PUSH CX operation

IN

- It transfers data from a port to accumulator for 8-bit, AL reg and 16-bit, AX reg.

Eg: IN AL, 8-bit port address IN AX, 16-bit port address

IN AL, DX IN AX, DX

OUT

- It transfers data from accumulator to an I/O port.

Eg: OUT 8-bit port address, AL OUT 16-bit port address, AX

OUT DX, AL OUT DX, AX

Additional instructions

- **LAHF:** loads the AH register with the low byte of the Flag. The instruction does not require any operand.
- **SAHF:** Stores the AH register content to the low byte of the flag register. As it copies into the low byte of flag register the five flags are affected.
- **PUSHF:** This instruction copies the flag register to top of stack.
- **POPF:** This instruction copies the top of stack into the flag register, thus this instruction affects all the flags.

Arithmetic instructions

- The instructions of this group perform addition, subtraction, multiplication, division, increment, decrement, comparison, ASCII and decimal adjust.
- These instructions and their operations can be performed on numbers expressed in a variety of numeric data formats.
- They include signed or unsigned binary bytes or words, packed or unpacked BCD bytes or ASCII numbers.

Instructions to perform addition

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **NPG** – Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.
- **AAM** – Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** – Used to divide the signed word by byte or signed double word by word.
- **AAD** – Used to adjust ASCII codes after division.
- **CBW** – Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** – Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Instructions to perform logical operation

- **NOT** – Used to invert each bit of a byte or word.
- **AND** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** – Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive–OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** – Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group –

- **REP** – Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** – Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** – Used to compare two string bytes/words.
- **INS/INSB/INSW** – Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** – Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** – Used to store the string byte into AL or string word into AX.

Branch and Loop Instructions

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition:

- **CALL** − Used to call a procedure and save their return address to the stack.
- **RET** − Used to return from the procedure to the main program.
- **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions –

- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.
- **JC** – Used to jump if carry flag CF = 1
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** – Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0
- **JNO** – Used to jump if no overflow flag OF = 0
- **JNP/JPO** – Used to jump if not parity/parity odd PF = 0
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JP/JPE** – Used to jump if parity/parity even PF = 1
- **JS** – Used to jump if sign flag SF = 1

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group –

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group –

- **LOOP** – Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** – Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** – Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** – Used to jump to the provided address if CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

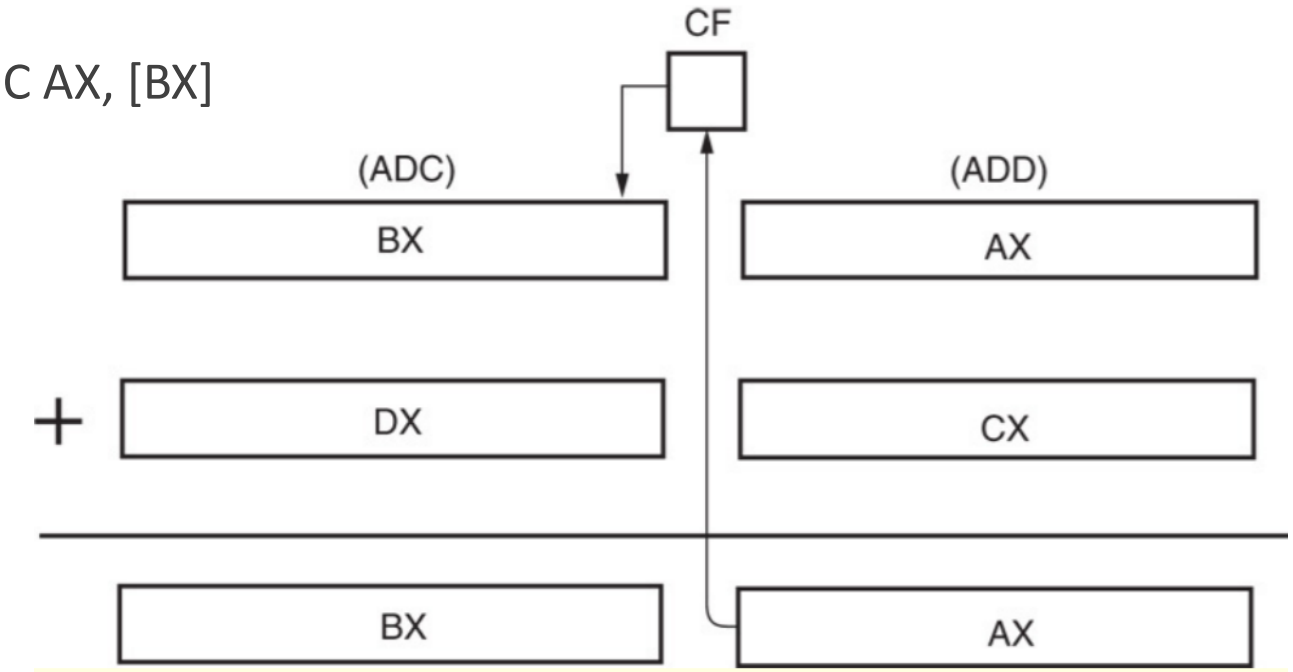
- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program

ADD

- This instruction adds the source with the destination, byte to byte or word to word.
- The result is stored in destination register
- It affects the AF, CF, OF, PF, SF, ZF
- Eg: ADD AL, 7AH ADD DX, AX ADD AX, [BX]

ADC

- This instruction adds the source with the destination along with the carry flag.
- The result is stored in destination register
- It affects the AF, CF, OF, PF, SF, ZF
- Eg: ADC AL, 7AH ADC DX, AX ADC AX, [BX]

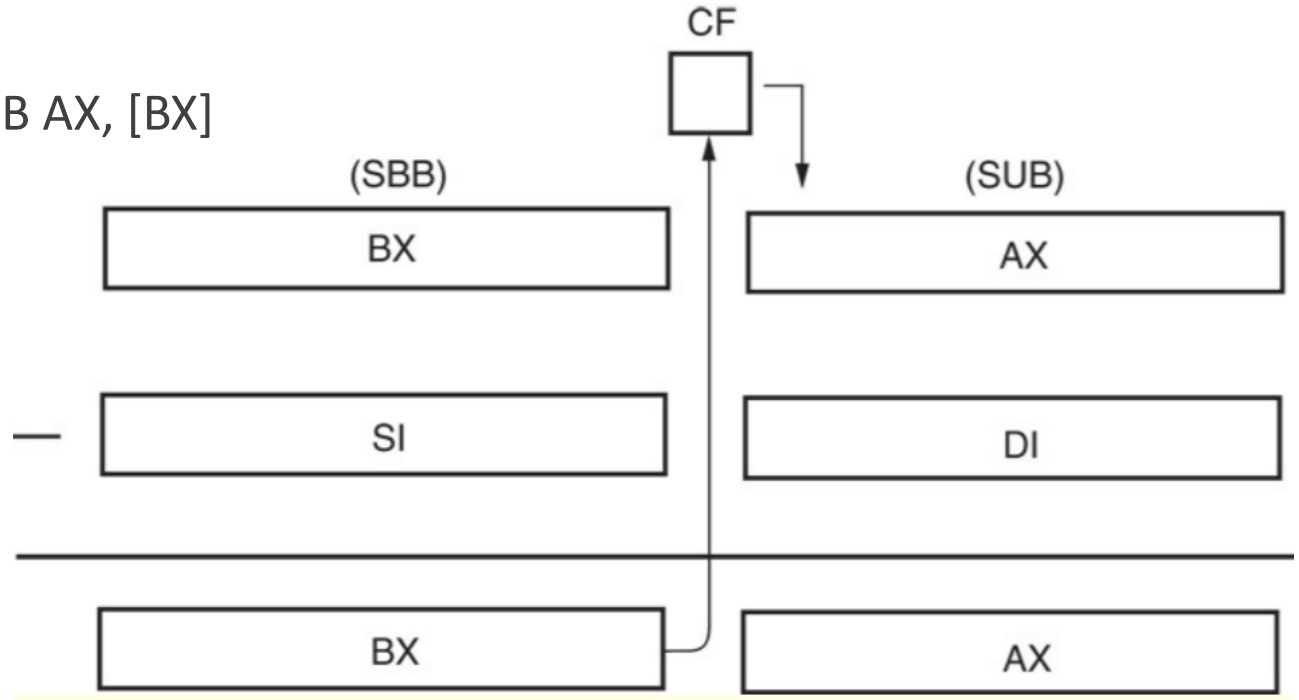


SUB

- This instruction subtracts the source from the destination byte to byte or word to word.
- The result is stored in destination register
- It affects the AF, CF, OF, PF, SF, ZF
- Eg: SUB AL, 7AH SUB DX, AX SUB AX, [BX]

SBB

- This instruction subtracts the source from the destination along with the borrow.
- The result is stored in destination register
- It affects the AF, CF, OF, PF, SF, ZF
- Eg: SBB AL, 7AH SBB DX, AX SBB AX, [BX]



NEG / INC / DEC

- NEG instruction negates the register value and stores the result in the register itself.

  Eg: NEG AX

- INC instruction adds ‘one’ to the register value and stores the result in the register itself.

  Eg: INC BX

- DEC instruction subtracts ‘one’ from the register value and stores the result in the register itself.

  Eg: DEC DX

MUL / IMUL

- MUL instruction multiplies an unsigned byte or word by the contents of AL or AX, respectively.

  Eg: MUL BH MUL CX MUL [SI]

- IMUL instruction multiplies a signed byte or word by the signed contents AL or AX, respectively.

  Eg: IMUL BH IMUL CX IMUL [SI]

- For 8-bit operations, the result is stored in AX; for 16-bit operations the result is stored in {DX,AX}

DIV / IDIV

- DIV instruction divides an unsigned 16-bit word in AX by the 8-bit contents of the register specified in the instruction; or divides an unsigned 32-bit word in {DX,AX} by the 16-bit contents of the register specified in the instruction.

  Eg: DIV BH DIV CX DIV [SI]

- IDIV instruction divides signed data in the similar manner as above.

  Eg: IDIV BH IDIV CX IDIV [SI]

- For 8-bit operations, the quotient is stored in AL and remainder in AH; for 16-bit operations the quotient is stored in AX and remainder in DX

CMP

- This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location

Eg: CMP BX, 0100H

CMP AX, 0100H

CMP [5000H], 0100H

CMP BX, [SI]

CMP BX, CX

CBW / CWD

- CBW instruction copies the sign of a byte in AL to all the bits in AH.
- AH is then said to be sign extension of AL.

  Eg: CBW

  (AX= 0000 0000 1001 1000; Result in AX = 1111 1111 1001 1000)
- This instruction copies the sign of a byte in AX to all the bits in DX.

  Eg: CWD

  (AX= 0000 0000 1001 1000;

  Result in DX = 0000 0000 0000 0000 : AX = 0000 0000 1001 1000)

DAA

- This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

Eg: AL = 53 CL = 29

```
ADD AL, CL          ; AL <– (AL) + (CL)
                    ; AL 53 + 29
                    ; AL 7C
DAA                 ; AL 7C + 06 (as C>9)
                    ; AL 82
```

DAS

- This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Eg: AL = 75, BH = 46

```
SUB AL, BH          ; AL   2F = (AL) – (BH)
                    ; AF = 1
DAS                 ; AL 29 (as F>9, F – 6 = 9)
```

AAA

- The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL.
- The AAA instruction converts the resulting contents of AL to unpacked decimal digits.

```
Eg: ADD CL, DL  ; [CL] = 32H = ASCII for 2
                ; [DL] = 35H = ASCII for 5
                ; Result [CL] = 67H
    MOV AL, CL  ; Move ASCII result into AL, since AAA adjusts only [AL]
    AAA         ; [AL]=07, unpacked BCD for 7
```

Logical instructions

- Instructions of this group perform Logical AND, OR, XOR, NOT, rotate, shift etc.

AND/OR/XOR/NOT

- Bit-by-bit logical AND of two operands

  Eg: AND AL, 05h

  AND AX, 1234h

  AND AL, BL

  AND AX, BX

- NOT is the 1’s complement of the specified mem/reg

TEST

- It performs bit-by-bit AND operation on the two operands and the result affects the Flag register without affecting the destination operand

Shift / Rotate Instructions

- These instructions manipulate the binary nos. at the bit level.
- It has 4 different shift instructions and 4 Rotate Instructions

1. SHL (Shift Logical Left)
2. SAL (Shift Arithmetic Left)
3. SHR (Shift Logical Right)
4. SAR (Shift Arithmetic Right)
5. ROL (Rotate Left)
6. ROR (Rotate Right)
7. RCL (Rotate Left through Carry)
8. RCR (Rotate Right through Carry)

Control Transfer Instructions

1. Unconditional
2. Conditional

Unconditional instructions

- CALL instruction calls a subroutine and saves the return address on the stack

  Eg: CALL 2050

- RET instr returns from the subroutine to the main program
- JUMP instr transfers the control of execution to the specified address

  Eg: JUMP 2050

- LOOP instr loops through a sequence of instructions until CX=0

  Eg: LOOP 2050

Conditional instructions

- JC instr jump if CF = 1
- JNC instr jump if CF = 0
- JZ instr jump if ZF = 1
- JNZ instr jump if ZF = 0
- JO instr jump if OF = 1
- JNO instr jump if OF = 0
- JP instr jump if PF = 1
- JNP instr jump if PF = 0

Conditional instructions

- JS instr jump if SF = 1
- JNS instr jump if SF = 0

Conditional Instructions

JA	Above	CF = 0 and ZF = 0
JAE	Above or equal	CF = 0
JB	Bellow	CF
JBE	Bellow or equal	CF or ZF
JC	Carry	CF
JE	Equality	ZF
JG	Greater(s)	ZF = 0 and SF = OF
JGE	Greater of equal(s)	SF = OF
JL	Less(s)	SF ≠ OF
JLE	Less equal(s)	ZF or SF ≠ OF
JNA	Not above	CF or ZF
JNAE	Neither above nor equal	CF
JNB	Not bellow	CF = 0
JNBE	Neither bellow nor equal	CF = 0 and ZF = 0
JNE	Not equal	ZF = 0
JNG	Not greater	ZF or SF ≠ OF
JNGE	Neither greater nor equal	SF ≠ OF
JNL	Not less	SF = OF
JNLE	Not less nor equal	ZF = 0 and SF = OF

String Manipulation Instructions

- A series of data bytes or words available in the memory at consecutive locations are called byte strings or word strings
- Length of the string is stored in CX reg
- For referring to a string, starting/ending address of the string and length of the string is required

LODS/STOS

- Load string byte or word

Eg: LODSB → Loads a byte into AL

AL = DS : [SI]

LODSW → Loads a word into AX

AX = DS : [SI]

Eg: STOSB → Stores a byte in AL

ES : [DI] = AL

STOSW → Stores a word in AX

ES : [DI] = AX

MOVS

- Moves a string byte or word in mem to another mem location

Eg: MOVSB → Moves string byte

ES : [DI] = DS : [SI]

MOVSW → Moves string word

ES : [DI] = DS : [SI]

MOVS

- Moves a string byte or word in mem to another mem location

Eg: MOVSB → Moves string byte

ES : [DI] = DS : [SI]

MOVSW → Moves string word

ES : [DI] = DS : [SI]

CMPS

- Compares two strings

Eg: CMPSB → Compares string bytes

ES : [DI] ↔ DS : [SI]

CMPSW → Compares string words

ES : [DI] ↔ DS : [SI]

SCAS

- It scans the string of bytes (SCASB) or word (SCASW) for an operand byte or word specified in the register AL or AX

Eg: SCASB → Compares AL with byte in memory

AL ↔ ES : [DI]

SCASW → Compares string words

AX ↔ ES : [DI]

SCAS

- It scans the string of bytes (SCASB) or word (SCASW) for an operand byte or word specified in the register AL or AX

Eg: SCASB → Compares AL with byte in memory

AL ↔ ES : [DI]

SCASW → Compares string words

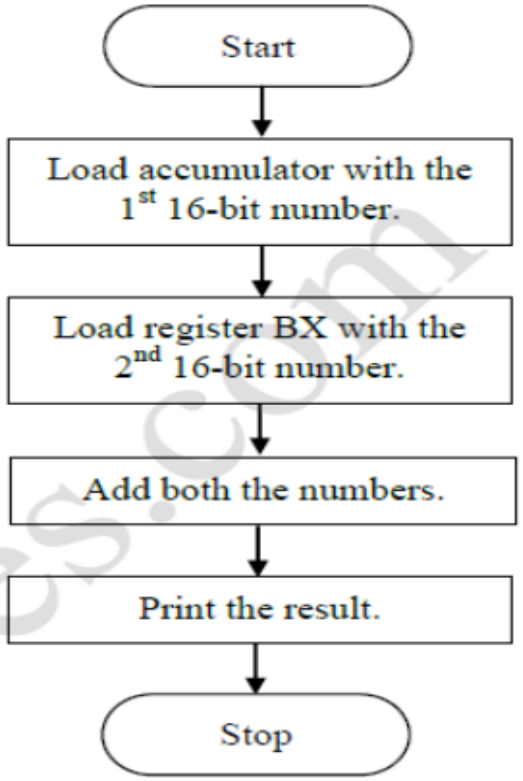
AX ↔ ES : [DI]

REP

Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0

Add two 16 bit numbers

Flowchart:



Program:

Instructions	Comments
include "emu8086.inc"	
ORG 100h	
MOV AX, 0005H	Move 1st 16-bit number to AX.
MOV BX, 0003H	Move 2nd 16-bit number to BX.
ADD AX, BX	Add BX with AX.
CALL PRINT_NUM	Print the result.
RET	Return.
DEFINE_PRINT_NUM	Declare function.
END	

Explanation:

- This program adds two 16-bit numbers.
- The program has been developed using ***emu8086*** emulator available at: www.emu8086.com.
- ORG 100h is a compiler directive. It tells compiler how to handle the source code.
- It tells compiler that the executable file will be loaded at the offset of 100h (256 bytes).
- The 1st 16-bit number 0005H is moved to accumulator AX.
- The 2nd 16-bit number 0003H is moved to register BX.
- Then, both the numbers are added and the result is stored in AX.
- The result is printed on the screen.

ALGORITHM:

I. Initialize the SI register to input data memory location
II. Initialize the DI register to output data memory location
III. Initialize the CL register to zero for carry
IV. Get the 1st data into accumulator.
V. Add the accumulator with 2nd data
VI. Check the carry flag, if not skip next line
VII. Increment carry(CL Reg)
VIII. Move the result from accumulator to memory.
IX. Also store carry register
X. Halt

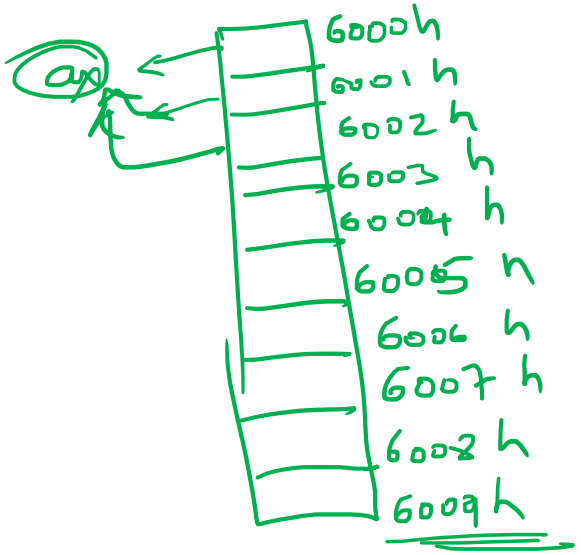
Program

```
MOV SI, 2000H
MOV DI, 3000H
MOV CL, 00H
MOV AX, [SI]
ADD AX, [SI+2]
JNC STORE
INC CL
MOV [DI], AX
MOV [DI+2], CL
INT 3
```

Q: Write a program to find the addition of 10 nos. at offset location 6000h.

Sol:-

```
        org 100h
        mov SI, 6000h
        mov ax, [SI]
        inc SI
        add ax, [SI]
next:   inc SI
        adc ax, [SI]
        cmp SI, 6009h
        jnz next
        ret
```



Q:- WAP to multiply two no.s at two labels 'operand1' & 'operand2'. Send the result to a port of 8086 having address 6791h.

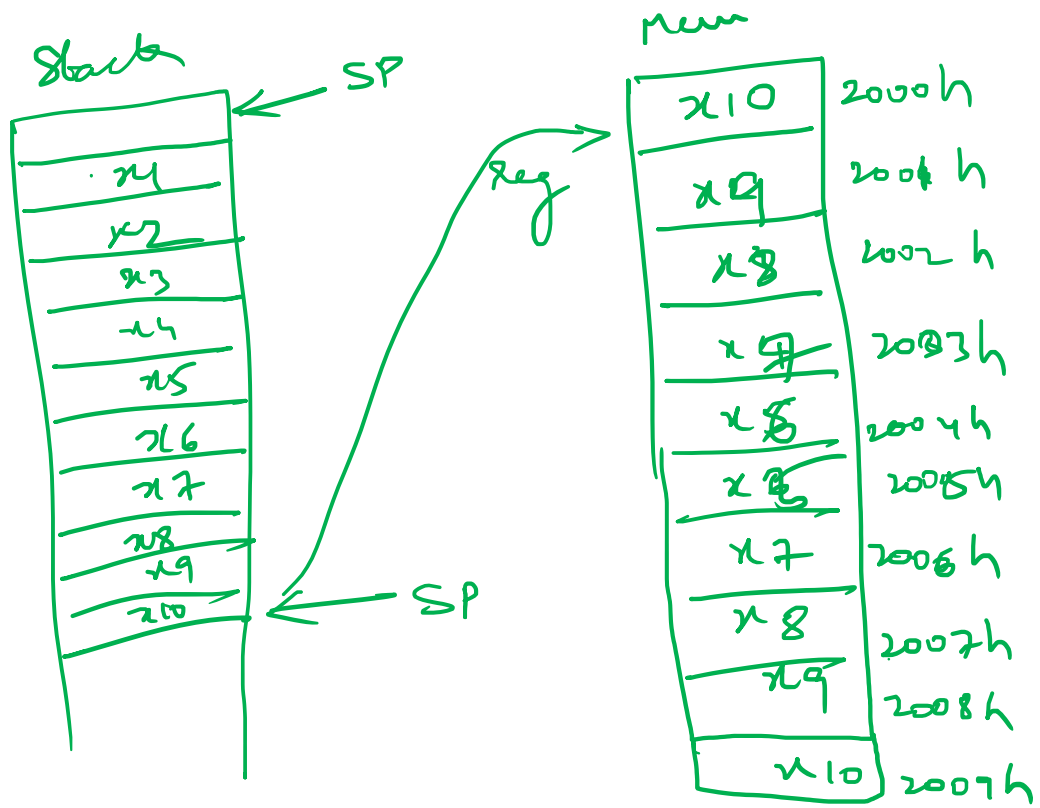
Sol:-

```
lea  SI, operand1
mov  al, [SI]
lea  SI, operand2
mov  bl, [SI]
mul  bl
mov  dx, 6791h
out  dx, ax
```

Q:- WAP to reverse an array ~~and~~ having 10 values using stack operations.

Sol:-

```
        org 100h
        mov SI, 2000h
next:   mov al, [SI]
        push al
        inc SI
        cmp SI, 200Ah
        JNZ next
        mov SI, 2000h
next1:  pop al
        mov [SI], al
        inc SI
        cmp SI, 200Ah
        JNZ next1
        ret
```



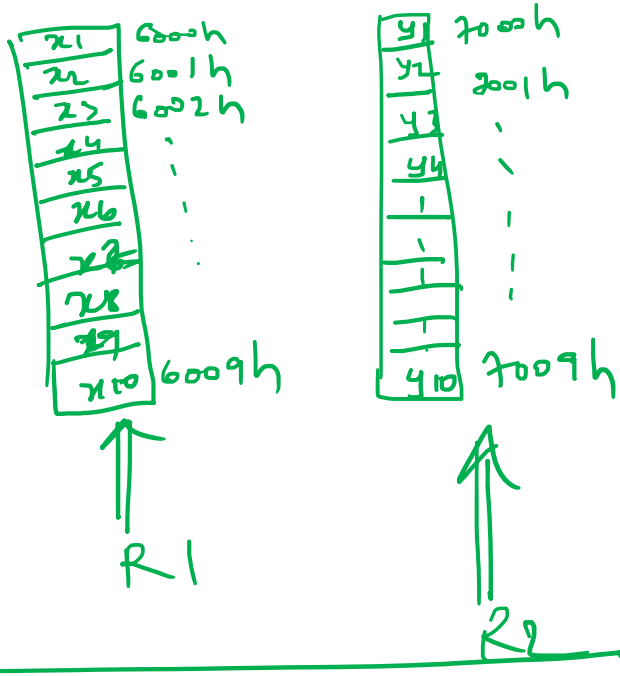
WAP to add a series of 8 bit numbers stored in memory locations starting from 1200 to 1209. Store the result in 120A

```
        MOV SI,1200H
        MOV AL,00H
        MOV CL,0AH
L1:     ADD AL,[SI]
        INC SI
        DEC CL
        JNZ L1
        MOV [SI],AL
        HLT
```

Q. WAP to add 10 nos. at 6000h & 7000h location seperately and subtract the lower one. Store the result in 8000h.

Sol:-

```
        org 100h
        mov SI, 6000h
        mov. al, [SI]
next:   inc SI
        add al, [SI]
        cmp SI, 6009h
        jnz next
        mov [9000h], al
        mov SI, 7000h
        mov al, [SI]
next1:  inc SI
        add al, [SI]
        cmp SI, 7009h
        JNZ next
        mov bl, [9000h]
        cmp al, bl
        JG code1
        sub bl, al
        mov [8000h], bl
        jmp exit
code1:  sub al, bl
        mov [8000h], al
exit:   ret
```



WAP to find the greatest number in a given set of data array

Memory address(offset	Data (Input)
1500	05
1501	25
1502	35
1503	20
1504	30
1505	15

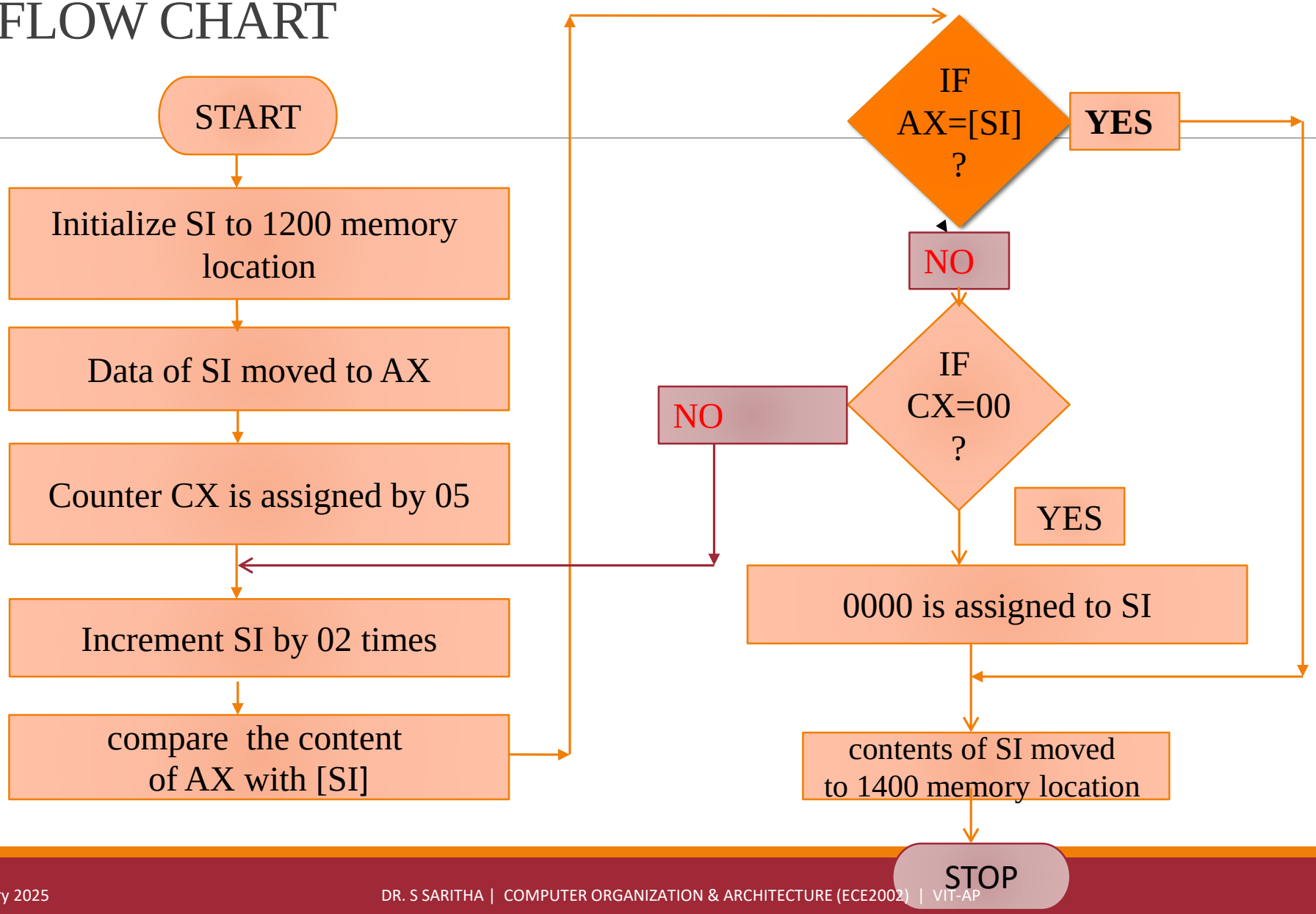
Memory address(offset	Data (Output)
1600	35

	MNEMONICS	COMMENT	
	MOV SI, 1500H	SI<-1500	
	MOV CL, [SI]	CL<-[SI]	
	INC SI		SI<-SI+1
	MOV AL, [SI]	AL<-[SI]	
	DEC CL		CL<-CL-1
	INC SI		SI<-SI+1
L1:	CMP AL, [SI]	AL-[SI]	
	JNC L2		JUMP TO L2 IF CY=0
	MOV AL, [SI]	AL<-[SI]	
L2:	INC SI		SI<-SI+1
	LOOP L1		**CX<-CX-1 & JUMP TO L1 IF CX NOT 0**
	MOV [1600], AL	AL->[1600]	
	HLT		END

How the program will be modify to write for finding the smallest number?

Searching the existence of a certain data in a given data array

FLOW CHART


START
Initialize SI to 1200 memory location
Data of SI moved to AX
Counter CX is assigned by 05
Increment SI by 02 times
compare the content of AX with [SI]
IF AX=[SI] ?
YES
NO
IF CX=00 ?
NO
YES
0000 is assigned to SI
contents of SI moved to 1400 memory location
STOP

PROGRAM

MEMORY ADDRESS	DATA IN
1200	AB96
1202	89CD
1204	AB96
1206	4EDF
1208	9197
120A	9600

Memory address	label	mnemonics
1000		MOV SI,1200H
		MOV AX,[SI]
		MOV CX,0005H
	GG	INC SI
		INC SI
		CMP AX,[SI]
		JE SS
		DEC CX
		JNZ GG
		MOV SI,0000H
	SS	MOV [1400], SI
		HLT

MEMORY ADDRESS	DATA IN
1200	AB96
1202	89CD
1204	AB96
1206	4EDF
1208	9197
120A	9600

MEMORY ADDRESS	DATA OUT
1400	1204

ALP to find the Sum of cubes of an array of size 10 by using 8086.

```
        MOV SI,0200 H
        MOV DI,0220H
        MOV CL,0AH
Up:     MOV AL,[SI]
        MOV BL,AL
        MUL BL
        MUL BL
        MOV [DI],AX
        INC SI
        INC DI
        INC DI
        DEC CL
        JNZ Up
        HLT
```

To separate odd and even numbers in a given data array

Address	Program	Explanation
	MOV CL, 06	Set counter in CL register
	MOV SI, 1600	Set source index as 1600
	MOV DI, 1500	Set Destination index as memory address 1500
Loop1:	MOV AL,[SI]	Load data from source memory
	ROR AL, 01	Rotate AL once to right
	JNC Loop1	If bit is one Jump to Loop1
	ROL AL,01	Rotate AL once to left
	MOV [DI], AL	Move result to Destination
	INC SI	Increment Destination index
	INC DI	Increment Destination index
	DEC CL	Decrement the count
	JNZ Loop1	Jump if CL not 0 to Loop1
	HLT	Stop the program

INPUT & OUTPUT:

INPUT	1600	2
	1601	5
	1602	7
	1603	6
	1604	12
	1605	15
OUTPUT	1500	5
	1501	7
	1503	15

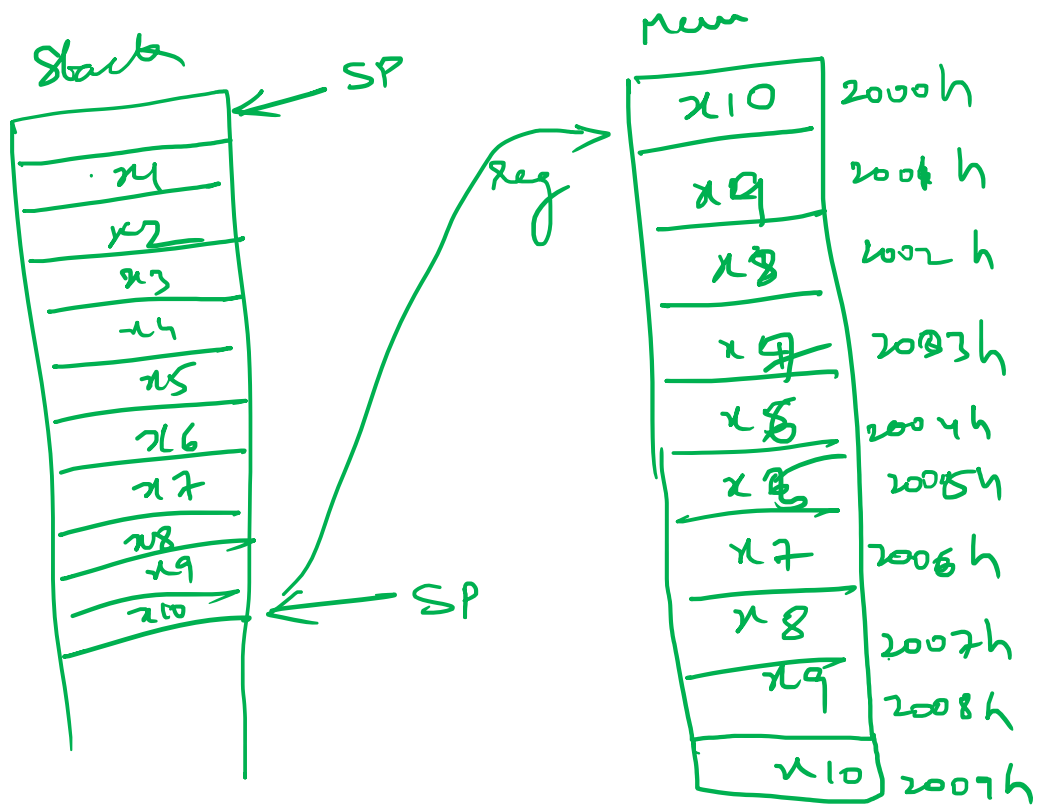
Similarly write a program to separate –ve numbers from +ve numbers in a given set of data

WAP to count the number of odd and even numbers in a given set of data array

Q:- WAP to reverse an array ~~and~~ having 10 values using stack operations.

Sol:-

```
        org 100h
        mov SI, 2000h
next:   mov al, [SI]
        push al
        inc SI
        cmp SI, 200Ah
        JNZ next
        mov SI, 2000h
next1:  pop al
        mov [SI], al
        inc SI
        cmp SI, 200Ah
        JNZ next1
        ret
```



Write a program to get Factorial of 10 numbers stored from the starting location 4000H:1000H. The results should be stored in 4000H:2000H

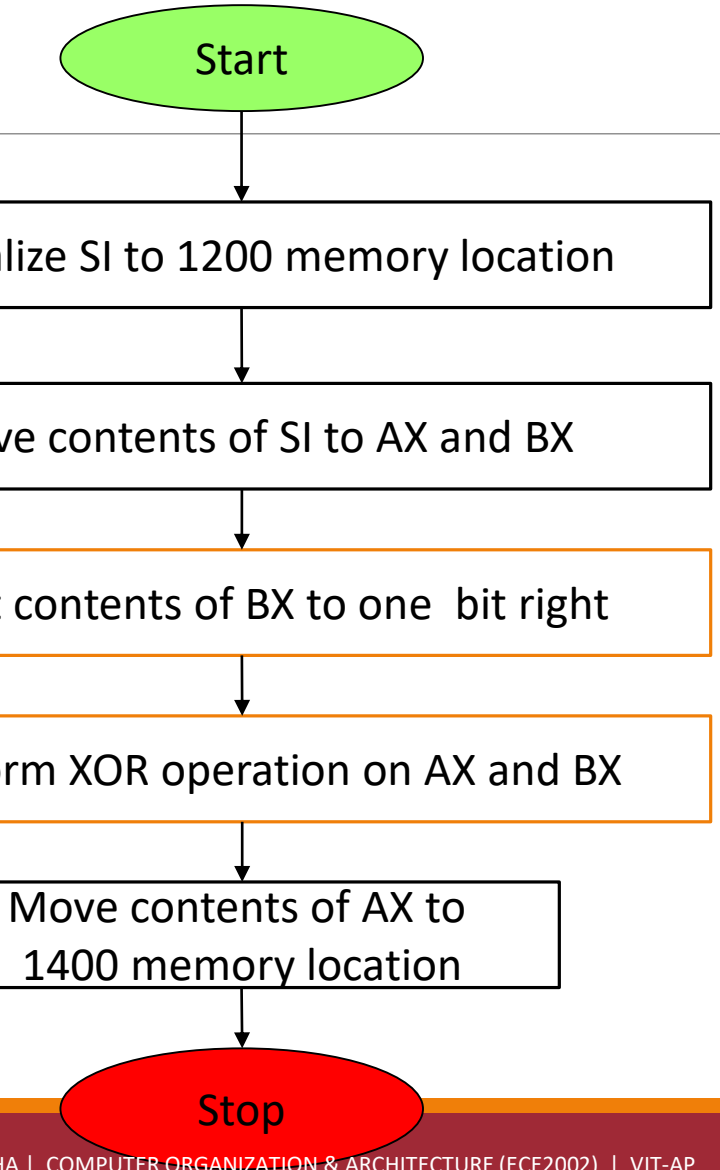
```
      MOV AX,4000H
      MOV DS, AX
      MOV SI,1000H
      MOV DI, 2000H
      MOV CL,0AH
      MOV AL, 01H
Next: MOV BL,[SI]
LOOK: MUL BL
      DEC BL
      JNZ LOOK
      MOV [DI], AL
      INC SI
      INC DI
      LOOP NEXT
      HLT
```

Write a program to sort a 8 bit data array in ascending order. The array consists of 5 numbers starting from location 3000H:4000H.

```
        MOV AX,3000H
        MOV DS,AX
        MOV CH, 04H
L3:     MOV CL,04H
        MOV SI, 4000H
L2:     MOV AL, [SI]
        MOV AH, [SI+01H]
        CMP AL, AH
        JB L1
        JZ L1
        MOV [SI+1], AL
        MOV [SI], AH
L1:     INC SI
        DEC CL
        JNZ L2
        DEC CH
        JNZ L3
        HLT
```

Address offset	Initial data	Ext loop 1	Ext loop 2	Ext loop 3	Ext loop 4
4000	55	45	35	25	15
4001	45	35	25	15	25
4002	35	25	15	35	35
4003	25	15	45	45	45
4004	15	55	55	55	55

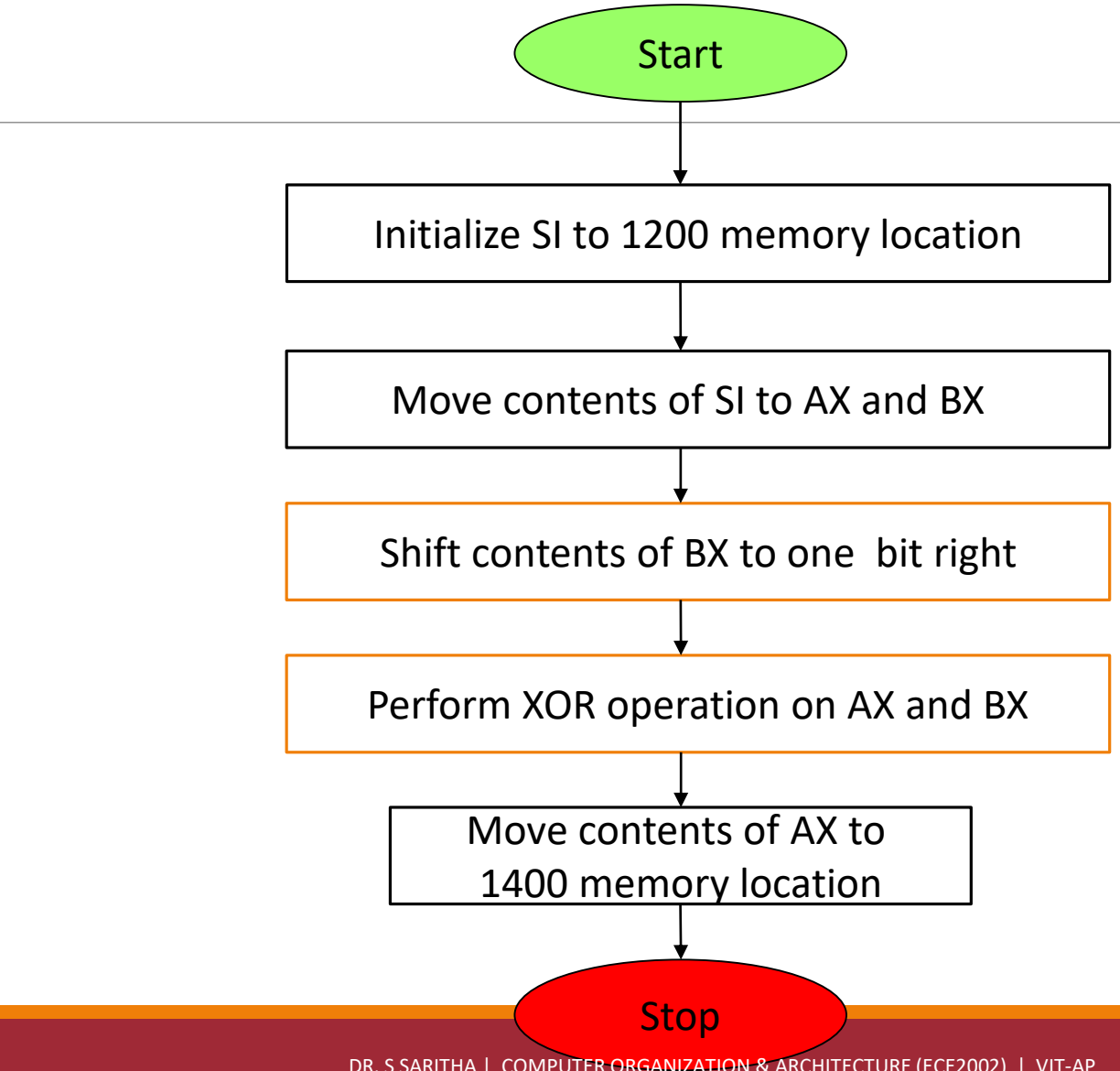
Write an ALP to convert binary number to gray code



Program

```
MOV SI,  1200
MOV AX,  [SI]
MOV BX,  [SI]
SHR BX,  01
XOR AX,  BX
MOV  [1400], AX
HLT
```

Write an ALP to convert binary number to gray code



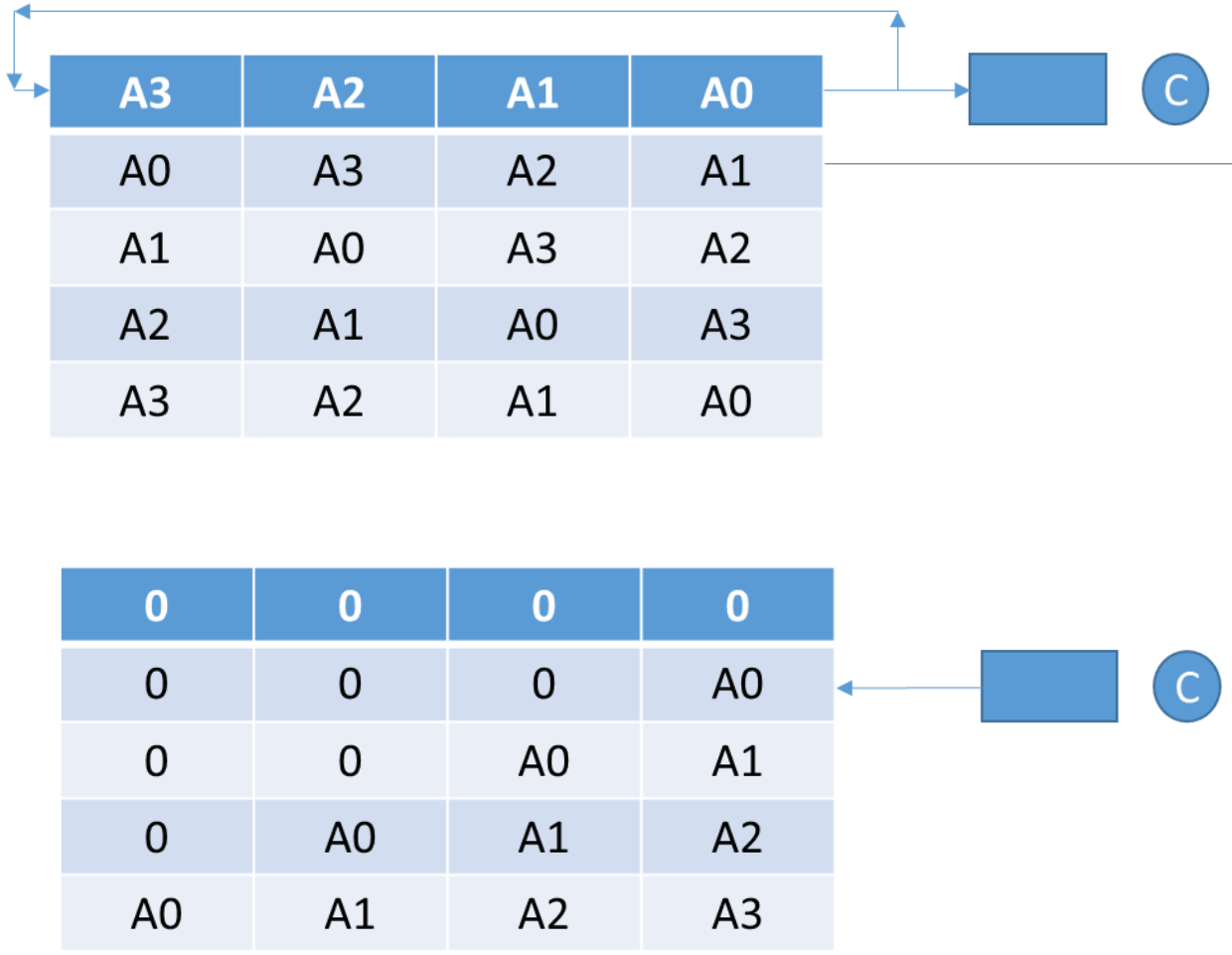
A3	A2	A1	A0
0	A3	A2	A1

A3XOR0	A2XORA3	A1XORA2	A0XORA1

Program

```
MOV SI,  1200
MOV AX,  [SI]
MOV BX,  [SI]
SHR BX,  01
XOR AX,  BX
MOV  [1400], AX
HLT
```

Program to check a number for bit wise palindrome. If palindrome place FFH at 2500H or place 00H at 2500H



A3	A2	A1	A0
A0	A3	A2	A1
A1	A0	A3	A2
A2	A1	A0	A3
A3	A2	A1	A0

C

0	0	0	0
0	0	0	A0
0	0	A0	A1
0	A0	A1	A2
A0	A1	A2	A3

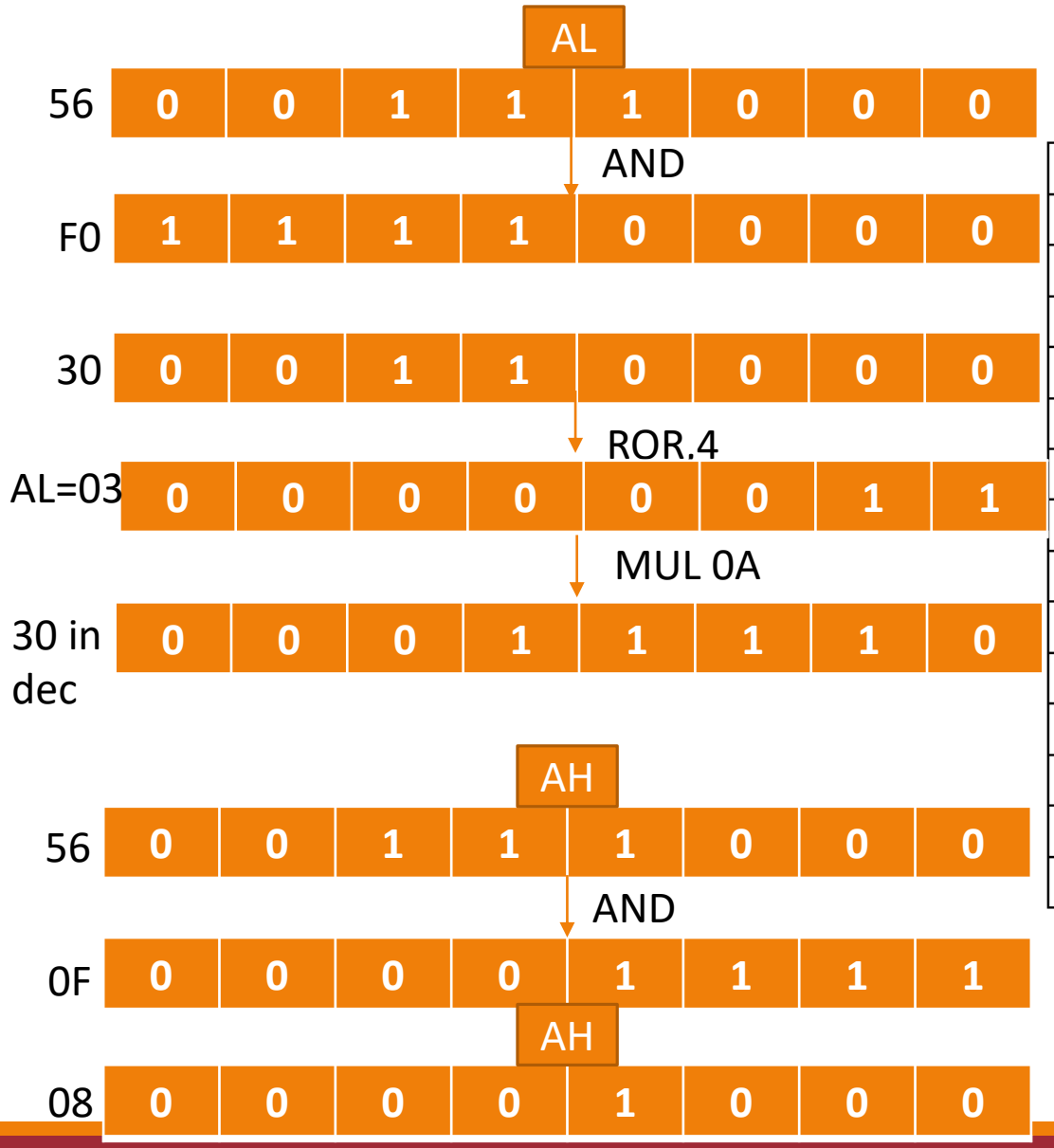
C

```
        MOV AX, [2300H]
        MOV CL,10H ;Initialize the counter 10.
UP:     ROR AX,1 ;Rotate right one time.
        RCL DX,1 ;Rotate left with carry one time.
        DEC CL
        JNZ UP ;Loop the process.
        CMP AX,DX ;Compare AX and DX.
        JNZ DOWN ;If no zero go to DOWN label.
        MOV [2500H], FFH ;Declare as a PALINDROME.
        JMP EXIT ;Jump to EXIT label.
DOWN:   MOV [2500H], 00H ; Declare as not a PALINDROME
EXIT:   HLT
```

Program to get the square root of a number

M0V AX, [0500H]	move the data from offset 500 to register AX
MOV CX, 0000H	move 0000 to register CX
MOV BX, FFFFH	move FFFF to register BX
L1: ADD BX, 0002H	add BX and 02
INC CX	increment the content of CX by 1
SUB AX, BX	subtract contents of BX from AX
JNZ L1	jump to address 040A if zero flag(ZF) is 0
MOV [0600], CX	store the contents of CX to offset 600
HLT	end the program

CONVERSION OF BCD TO HEXADECIMAL



Program	Explanation
MOV SI, 1600	Set source index as 1600
MOV DI, 1500	Set Destination index as memory address 1500
MOV AL, [SI]	Load BCD number into AL register
MOV AH, AL	Move content of AL to AH
AND AH, 0F	Mask MSD of BCD number
MOV BL, AH	Save LSD in BL register
AND AL, F0	Mask LSD of BCD number
MOV CL, 04	Load 04 to counter
ROR AL, CL	rotate AL by counter times
MOV BH, 0A	move 0A to BH register
MUL BH	Multiply BH register
ADD AL, BL	Add AL, BL
MOV [DI], AL	Move result to Destination
HLT	Stop

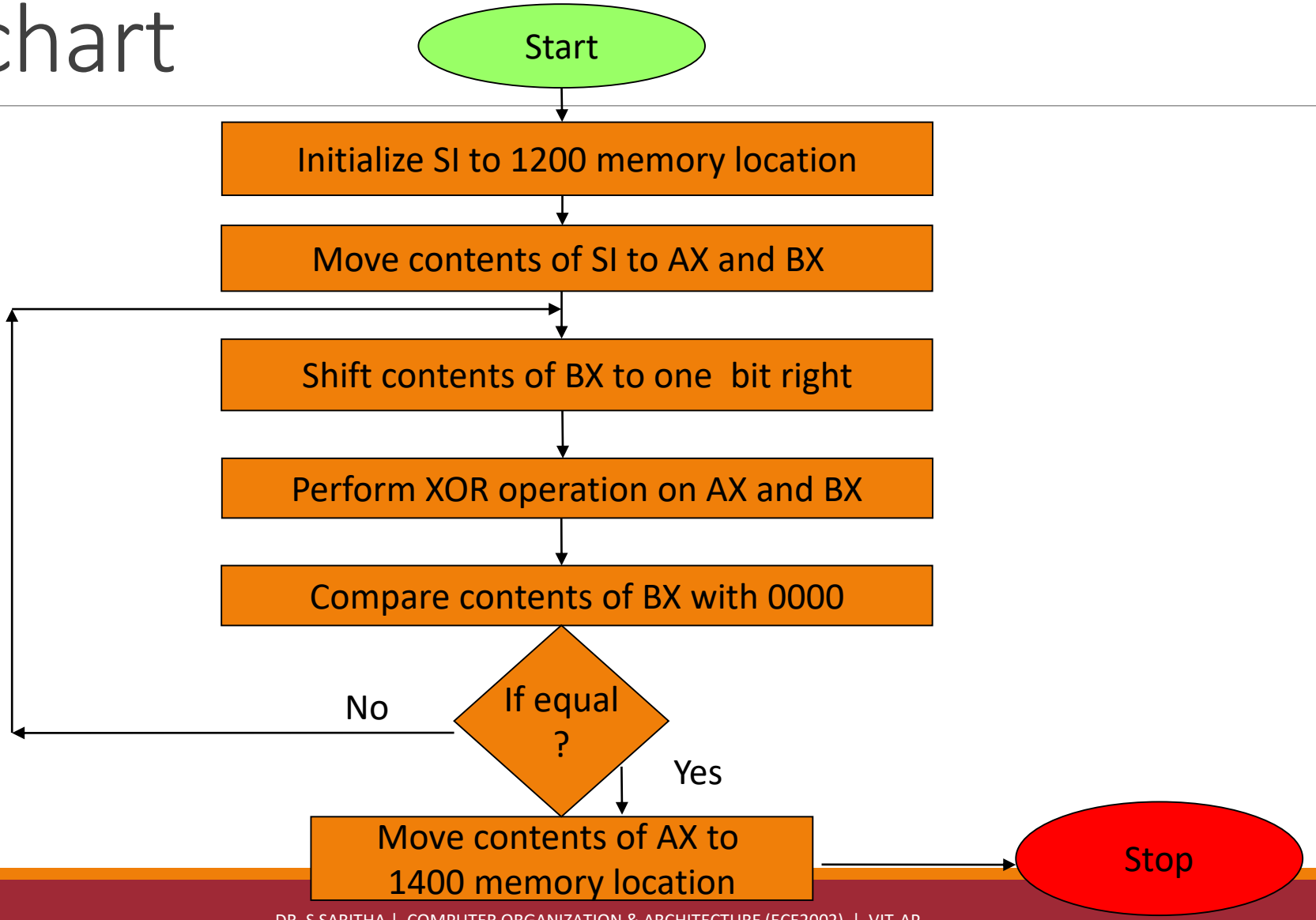
INPUT & OUTPUT:

INPUT	1600	56
OUTPUT	1500	38

Write a program to get Factorial of 10 numbers stored from the starting location 4000H:1000H. The results should be stored in 4000H:2000H

```
      MOV 4000H, AX
      MOV DS, AX
      MOV SI,1000H
      MOV DI, 2000H
      Mov CL,0AH
Next: MOV AL, 01H
      MOV BL,[SI]
LOOK: MUL BL
      DEC BL
      JNZ LOOK
      MOV [DI], AL
      INC SI
      INC DI
      LOOP NEXT
      HLT
```

Flowchart


Start
Initialize SI to 1200 memory location
Move contents of SI to AX and BX
Shift contents of BX to one bit right
Perform XOR operation on AX and BX
Compare contents of BX with 0000
If equal ?
No
Yes
Move contents of AX to 1400 memory location
Stop

Program

	Mnemonics
	MOV SI, 1200
	MOV AX, [SI]
	MOV BX, [SI]
LOOP 1	SHR BX, 01
	XOR AX, BX
	CMP BX, 0000
	JE LOOP 2
	JMP LOOP 1
LOOP 2	MOV [1400], AX
	HLT

Program to find Greatest common divisor (GCD) of given numbers

```
        MOV SI, 2300H           ;Store Offset address 2300h in SI
        MOV DI, 2400H           ;Store offset address 2400h in DI
        ---------------------------------------------------------
        MOV AX, [SI]    ;Move the first number to AX.
        MOV BX, [SI+1]          ;Move the second number to BX.
UP:     CMP AX, BX      ;Compare the two numbers.
        JE EXIT                 ;If equal, go to EXIT label.
        JB EXCG                 ;If first number is below than second, go to EXCG label.
UP1:    MOV DX,0000H            ;Initialize the DX.
        DIV BX                  ;Divide the first number by second number.
        CMP DX,0000H            ;Compare remainder is zero or not.
        JE EXIT                 ;If zero, jump to EXIT label.
        MOV AX,DX       ;If non-zero, move remainder to AX.
        JMP UP                  ;Jump to UP label.
EXCG:   XCHG AX,BX      ;Exchange the remainder and quotient.
        JMP UP1                 ;Jump to UP1.
EXIT:   MOV [DI], BX    ;Store the result in DI.
        HLT                     ; Stop
```

Program to find least common multiple (LCM) of a given numbers

```
MOV SI, 2300H          ;Store Offset address 2300h in SI
MOV DI, 2400H          ;Store offset address 2400h in DI
MOV DX,0000H           ;Initialize the DX
MOV AX,[SI]            ;Move the first number to AX
MOV BX,[SI+2]          ;Move the second number to AX
PUSH AX                ;Store the quotient/first number STACK
PUSH DX                ;Store the remainder STACK
DIV BX                 ;Divide the first number by second number
CMP DX,0000H           ;Compare the remainder.
JE EXIT                ;If remainder is zero, go to EXIT label
POP DX                 ;If remainder is non-zero, ;Retrieve the remainder from stack
POP AX                 ;Retrieve the quotient from stack
ADD AX,[SI]            ;Add first number with AX
JNC DOWN               ;If no carry jump to DOWN label
INC DX                 ;Increment DX
DOWN:   JMP UP         ;Jump to Up label
EXIT:   MOV [DI], AX ;If remainder is zero, store the value of LCM at destination
HLT                    ;Stop
```

Some important programs

Program to count logical 1’s and 0’s in a given data

Program for getting square of array of numbers

Program to find LCM of a given numbers

Program to find GCD of a given numbers

Program to check a number is Bit wise palindrome or not

Program to check a 16 bit number is Nibble wise palindrome or not

Program to reverse a string

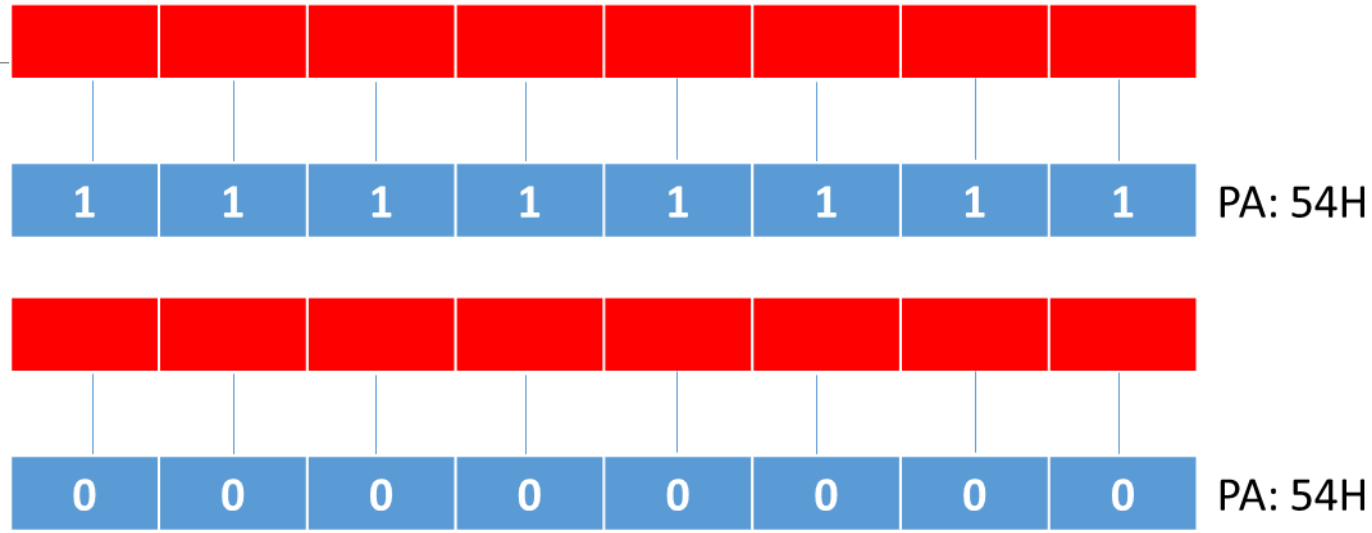
Program to search for a character in a string

Write a program to swap the nibbles of 10 data stored in the memory which starts from 2000

```
        MOV CL, 0A
        MOV SI, 2000
        MOV DI, 3000
L1:     MOV AL, [SI]
        ROR AL, 04
        MOV [DI], AL
        INC SI
        INC DI
        DEC CL
        JNZ L1
        HLT
```

Consider all the 8 bits of an output port having address 54H are connected to 8 LEDs. Write a program to blink all the LEDs ON and OFF with a time gap of 5 seconds. Consider the time taken by any instruction to be executed is 1 Second.

```
L1:     MOV AL, FFH
        OUT 54H
        CALL DELAY
        MOV AL, 00H
        OUT 54H
        CALL DELAY
        JMP L1
```




```
DELAY:  MOV CL, 02H
L2:     DEC CL
        JNZ L2
        RET
```

Write a program to add the contents of the memory location 0500H to register BX and CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using offset=0600H. Store the result of the addition in 0700H. Assume data segment's starting physical address is 20000H.

```
MOV AX, 2000H

MOV DS, AX

ADD BX, [0500H]

ADD CX, BX

MOV DL, 05H

ADD DL, [0600]

MOV [0700], DL

HLT
```

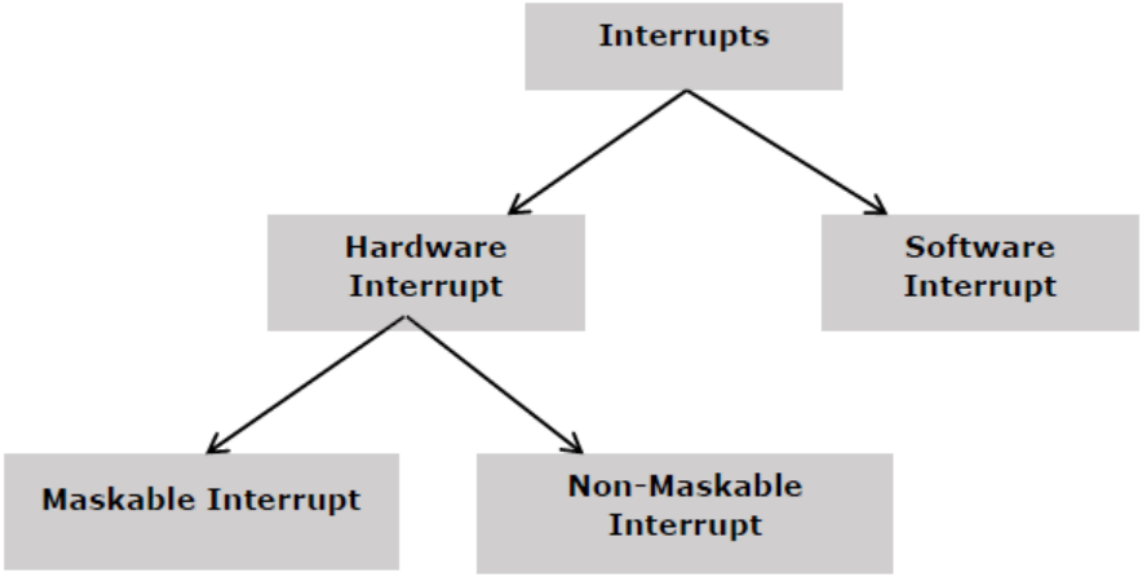
WAP to reverse a string of 10 bytes using stack. Check whether the string is palindrome or not. If it is palindrome display FFH on a display unit whose address is 52H else display 00H.

```
        MOV SI, 2300H
        MOV DI, 2500H
        MOV CL, 0AH
L1:     MOV AL, [SI]
        PUSH AL
        INC SI
        DEC CL
        JNZ L1
        MOV CL, 0AH
L2:     POP AL
        MOV [DI], AL
        INC DI
        DEC CL
        JNZ L2
```

SI		STACK		DI	
2300	11	FFF1	AA	2500	AA
2301	22	FFF2	99	2501	99
2302	33	FFF3	88	2502	88
2303	44	FFF4	77	2503	77
2304	55	FFF5	66	2504	66
2305	66	FFF6	55	2505	55
2306	77	FFF7	44	2506	44
2307	88	FFF8	33	2507	33
2308	99	FFF9	22	2508	22
2309	AA	FFFA	11	2509	11

Interrupts

- An interrupt is used to cause a temporary halt in the execution of program.
- Microprocessor responds to the interrupt with an interrupt service routine → short program or subroutine that instructs the microprocessor on how to handle the interrupt.



Interrupts

- A non-maskable interrupt requires an immediate response by microprocessor → usually used for serious circumstances like power failure
- A maskable interrupt is an interrupt that the microprocessor can ignore depending upon some predetermined condition defined by status register

Interrupt can divide to five groupe:

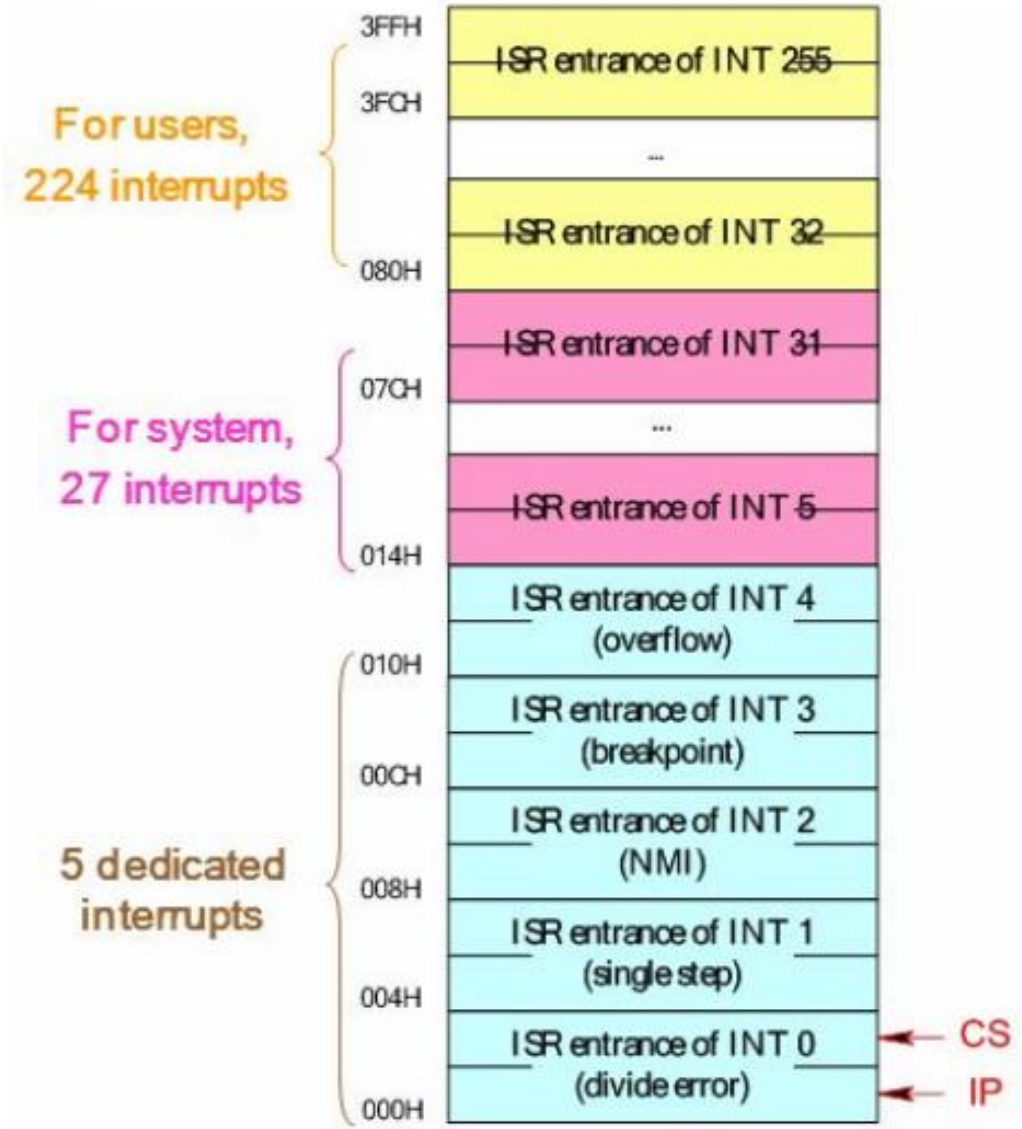
1. hardware interrupt
2. Non-maskable interrupt
3. Software interrupt
4. Internal interrupt
5. Reset

Interrupts

- Hardware, software and internal interrupt are serviced on priority basis.
- Each interrupt is given a different priority level by assigning it to a type number.
- Type 0 identifies the highest-priority and type 255 identifies the lowest-priority interrupt.
- The 8086 chips allow up to 256 vectored interrupts → it can have up to 256 different sources for an interrupt and the 8086 will directly call the service routine for that interrupt without any software processing.
- This is in contrast to non-vectored interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

Interrupts

- The 8086 provides a 256 entry interrupt vector table beginning at address 0:0 in memory.
- This is a 1K table containing 256 4-byte entries.
- Each entry in this table contains a segmented address that points at the interrupt service routine in memory.
- The lowest five types are dedicated to specific interrupts such as the divide by zero interrupt and the non maskable interrupt.
- The next 27 interrupt types, from 5 to 31 are reserved by Intel for use in future microprocessors.
- The upper 224 interrupt types, from 32 to 255, are available to use for hardware and software interrupts.



- 256 interrupts
  - 0 ~ 4 dedicated
  - 5 ~ 31reserved for system use
    - 08H ~ 0FH : 8259A
    - 10H ~ 1FH : BIOS
  - 32 ~ 255 reserved for users
    - 20H ~ 3FH : DOS
    - 40H ~ FFH : open

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.

Vector No.	Mnemonic	Description	Source
9		CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		(Intel reserved. Do not use.)	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19-31		(Intel reserved. Do not use.)	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Interrupts

- When an interrupt occurs, regardless of source, the 8086 does the following:

1. The CPU pushes the flag register onto the stack.
2. The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
3. The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from the IVT.
4. The CPU transfers control to the routine specified by the interrupt vector table entry.
5. When the interrupt service routine wants to return control, it must execute an IRET (interrupt return) instruction.
6. The interrupt return pops the far return address and the flags off the stack.

Interrupts

- When an interrupt occurs, regardless of source, the 8086 does the following:

1. The CPU pushes the flag register onto the stack.
2. The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
3. The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from the IVT.
4. The CPU transfers control to the routine specified by the interrupt vector table entry.
5. When the interrupt service routine wants to return control, it must execute an IRET (interrupt return) instruction.
6. The interrupt return pops the far return address and the flags off the stack.

Reset

- Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin.
- The 8086 RESET is required to be HIGH for greater than 4 CLK cycles.
- The 8086 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH.
- The low-going transition of RESET triggers an internal reset sequence for approximately 10 CLK cycles.
- After this interval the 8086 operates normally beginning with the instruction in absolute location FFFF0H.

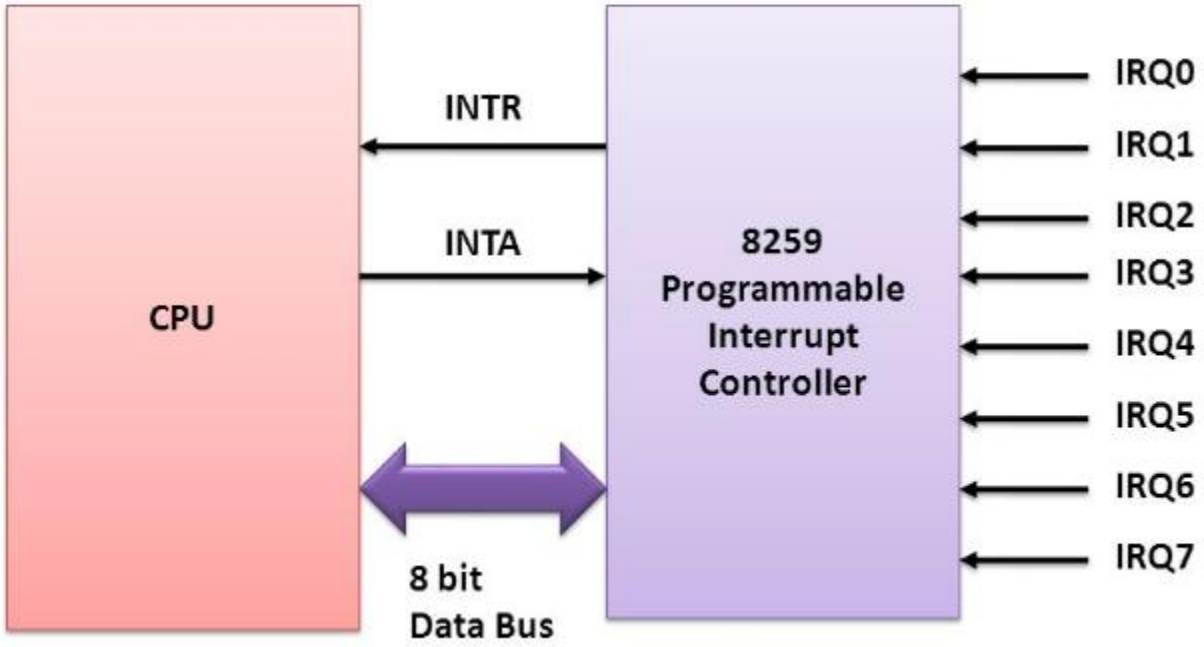
Non-Maskable interrupt

- The processor provides a single non-maskable interrupt pin (NMI) which has higher priority than the maskable interrupt request pin (INTR).
- A typical use would be to activate a power failure routine.
- The NMI is edge-triggered on a LOW-to-HIGH transition.
- The activation of this pin causes a type 2 interrupt.
- NMI is required to have a duration (in the HIGH state) of greater than two CLK cycles, but is not required to be synchronized to the clock.

Hardware Interrupt

- The primary sources of interrupts are the PCs timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices.
- These devices connect to an Intel 8259A programmable interrupt controller (PIC) that prioritizes the interrupts and interfaces with the 8086 CPU.
- The 8259A chip adds considerable complexity to the software that processes interrupts.

8259: Programmable Interrupt Controller



Maskable interrupt

- Whenever an external signal activates the INTR pin, the microprocessor will be interrupted only if interrupts are enabled using set interrupt Flag instruction (STI).
- If the interrupts are disabled using clear interrupt Flag instruction (CLI), the microprocessor will not get interrupted even if INTR is activated.
- That is, INTR can be masked.
- INTR is a non vectored interrupt, which means, the 8086 does not know where to branch to service the interrupt.
- The 8086 has to be told by an external device like a Programmable Interrupt controller regarding the branch.
- Whenever the INTR pin is activated by an I/O port, if Interrupts are enabled and NMI is not active at that time, the microprocessor finishes the current instruction that is being executed and gives out a '0' on INTA pin twice.

Maskable interrupt

- When INTA pin goes low for the first time, it asks the external device to get ready.
- In response to the second INTA the microprocessor receives the 8 bit, say N, from a programmable Interrupt controller.
- The action taken is as follows:

1. Complete the current instruction.
2. Activates INTA output, and receives type Number, say N
3. Flag register value, CS value of the return address & IP value of the return address are pushed on to the stack.
4. IP value is loaded from contents of word location N x 4.
5. CS is loaded from contents of the next word location.
6. Interrupt Flag and trap Flag are reset to 0.

Maskable interrupt

- At the end of the ISR, there will be an IRET instruction.
- This performs popping off from the stack top to IP, CS and Flag registers.
- Finally, the register values which are also saved on the stack at the start of ISR, are restored from the stack and a return to the interrupted program takes place using the IRET instruction.

INT 21h

- **INT 21h** / **AH=1** - read character from standard input, with echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 1
int 21h
```

- **INT 21h** / **AH=2** - write character to standard output.
  entry: **DL** = character to write, after execution **AL = DL**.

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h

- **INT 21h** / **AH=5** - output character to printer.
  entry: **DL** = character to print, after execution **AL = DL**.

```
mov ah, 5
mov dl, 'a'
int 21h
```

- **INT 21h** / **AH=6** - direct console input or output.
  parameters for output: **DL** = 0..254 (ascii code)
  parameters for input: **DL** = 255
  for output returns: AL = DL
  for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
  AL = character read; buffer is cleared.

```
mov ah, 6
mov dl, 'a'
int 21h
```

```
mov ah, 6
mov dl, 255
int 21h
```

INT 21h

- **INT 21h** / **AH=1** - read character from standard input, with echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 1
int 21h
```

- **INT 21h** / **AH=2** - write character to standard output.
  entry: **DL** = character to write, after execution **AL = DL**.

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h

- **INT 21h** / **AH=5** - output character to printer.
  entry: **DL** = character to print, after execution **AL = DL**.

```
mov ah, 5
mov dl, 'a'
int 21h
```

- **INT 21h** / **AH=6** - direct console input or output.
  parameters for output: **DL** = 0..254 (ascii code)
  parameters for input: **DL** = 255
  for output returns: AL = DL
  for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
  AL = character read; buffer is cleared.

```
mov ah, 6
mov dl, 'a'
int 21h
```

```
mov ah, 6
mov dl, 255
int 21h
```

INT 21h

- **INT 21h** / **AH=7** - character input without echo to AL.
  if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 7
int 21h
```

- **INT 21h** / **AH=9** - output of a string at **DS:DX**. String must be terminated by '**$**'.

```
org 100h
mov dx, offset msg
mov ah, 9
int 21h
ret
msg db "hello world $"
```

INT 21h

- **INT 21h** / **AH=0Ah** - input of a string to **DS:DX**, first byte is buffer size, second byte is number of chars actually read. this function does **not** add '$' in the end of string. to print using **INT 21h** / **AH=9** you must set dollar character at the end of it and start printing from address **DS:DX+2**.

```
org 100h
mov dx, offset buffer
mov ah, 0ah
int 21h
jmp print
buffer db 10,?, 10 dup(' ')
print:
xor bx, bx
mov bl, buffer[1]
mov buffer[bx+2], '$'
mov dx, offset buffer + 2
mov ah, 9
int 21h
ret
```