Module 2

By
B N KIRAN
Assistant Professor
Department of Information Science and Engineering
NIE, Mysuru – 570 008

8086 Instructions

- Based on functionality it has been classified into 8 groups
  - Data Transfer Instructions
  - Arithmetic Instructions
  - Bit Manipulation Instructions
  - String Instructions
  - Program Execution Transfer Instructions (Branch & Loop Instructions)
  - Processor Control Instructions
  - Iteration Control Instructions
  - Interrupt Instructions

Data Transfer Instructions

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PUSH** – 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.

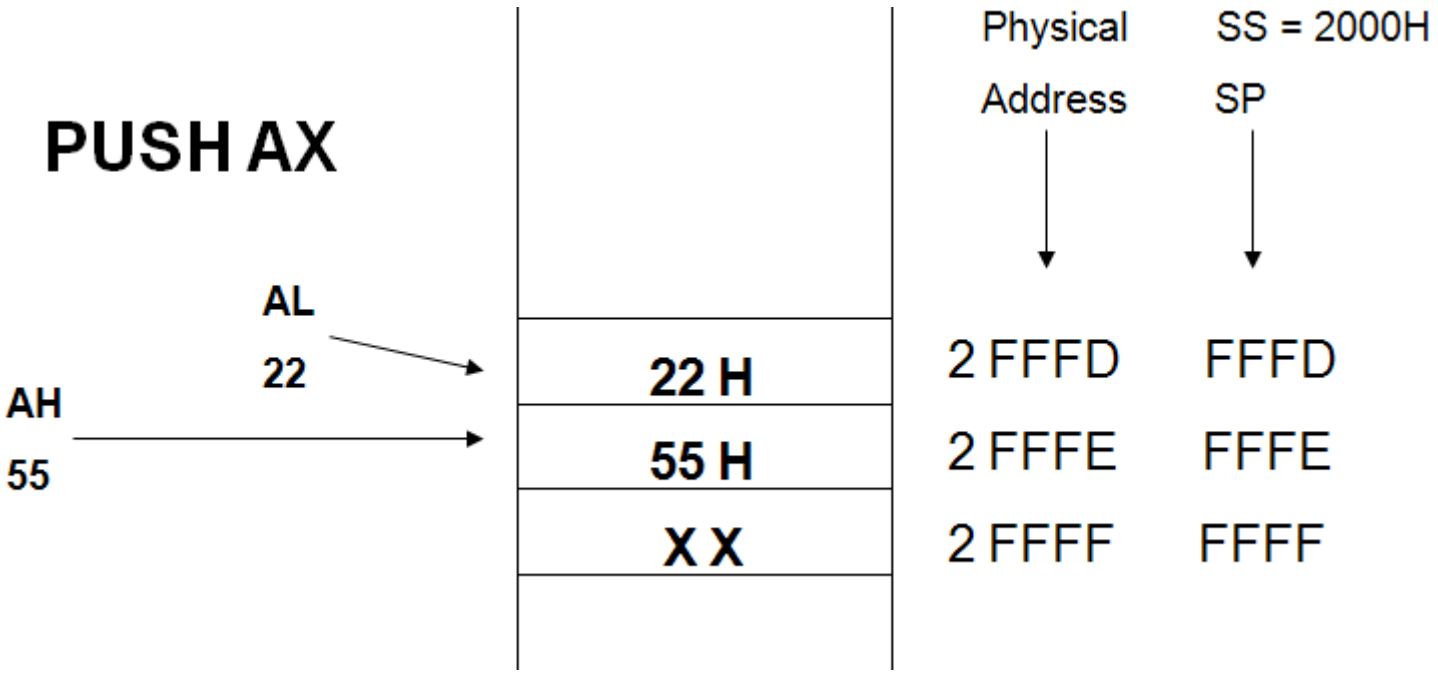
- Instructions for input and output port transfer
  - **IN** – Used to read a byte or word from the provided port to the accumulator.
  - **OUT** – Used to send out a byte or word from the accumulator to the provided port.
- Instructions to transfer the address
  - **LEA** – Used to load the address 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 Instruction

- MOV CX, 037AH ;Put immediate number 037AH to CX
- MOV BL, [437AH] ;Copy byte in DS at offset 437AH to BL
- MOV AX, BX ;Copy content of register BX to AX
- MOV DL, [BX] ;Copy byte from memory at [BX] to DL
- MOV DS, BX ;Copy word from BX to DS register
- MOV RESULT [BP], ;AX Copy AX to two memory locations; AL to the first location, AH to the second; EA of the first memory location is sum of the displacement represented by RESULTS and content of BP. Physical address = EA + SS.
- MOV ES: RESULTS [BP], ;AX Same as the above instruction, but physical address = EA + ES, because of the segment override prefix ES

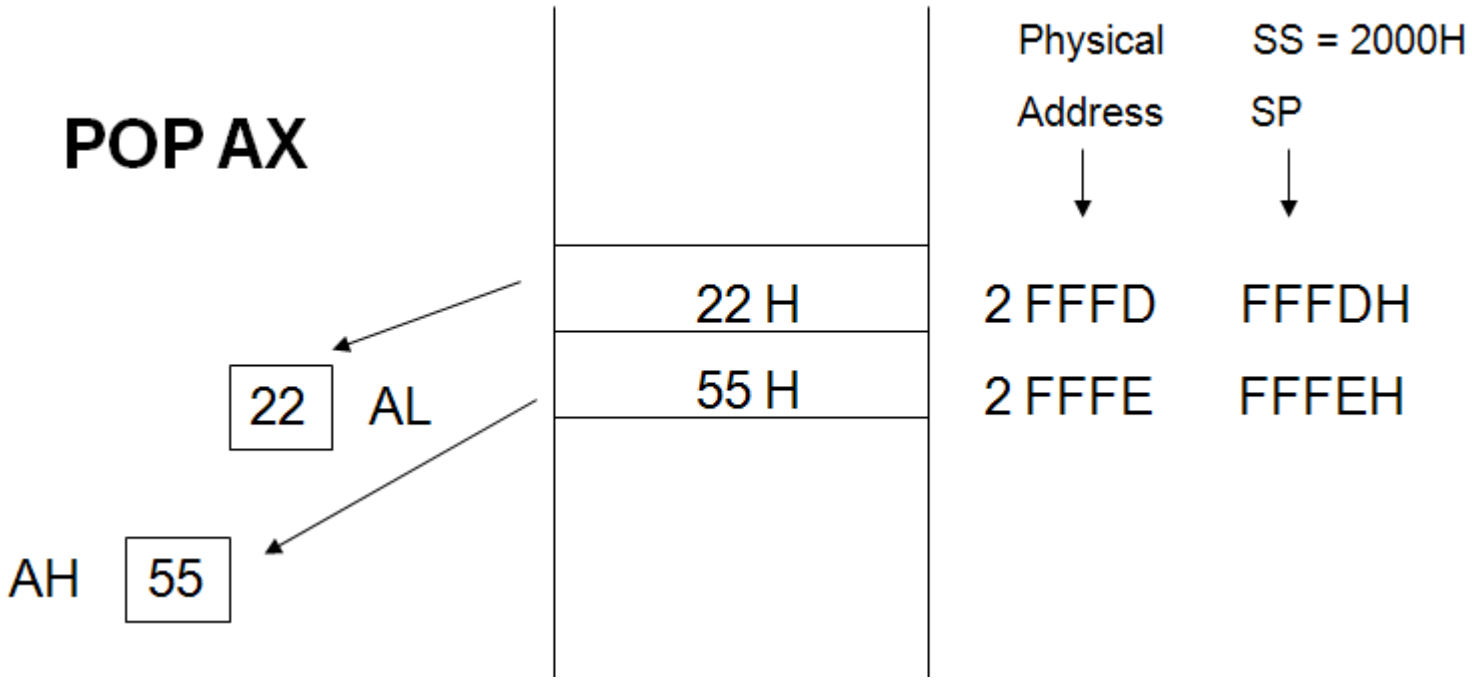
PUSH INSTRUCTION

- **PUSH: Push to Stack**
- This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.
- E.g. PUSH AX
- PUSH DS
- PUSHA
- PUSH [5000H]


PUSH AX
AL
22
AH
55
22 H
55 H
X X
Physical Address
SS = 2000H
SP
2 FFFD
FFFD
2 FFFE
FFFE
2 FFFF
FFFF

POP : Pop from Sack

- This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer.
- The stack pointer is incremented by 2 Eg. POP AX
- POP DS
- POP [5000H]


POP AX
Physical Address
SS = 2000H
SP
22 H
55 H
2 FFFD
2 FFFE
FFFDH
FFFEH
22
AL
AH
55

XCHG : Exchange byte or word

- This instruction exchange the contents of the specified source and destination operands
- Eg. XCHG [5000H], AX
- XCHG BX, AX

XLAT

- Translate byte using look-up table
- Eg. LEA BX, TABLE1
- MOV AL, 04H XLAT

IN & OUT Instructions

- Copy a byte or word from specified port to accumulator.
- Eg. IN AL,03H
- IN AX,DX
- **OUT:**
- Copy a byte or word from accumulator specified port.
- Eg. OUT 03H, AL
- OUT DX, AX

LEA & LDS

- **LEA :**
  - Load effective address of operand in specified register. [reg] offset portion of address in DS
  - Eg. LEA reg, offset
- **LDS:**
  - Load DS register and other specified register from memory. [reg] [mem]
  - [DS] [mem + 2]
  - Eg. LDS reg, mem

- **LES:**
- Load ES register and other specified register from memory. [reg] [mem]
- [ES] [mem + 2]
- Eg. LES reg, mem

LAHF & SAHF Instructions

- **Flag transfer instructions:**
- **LAHF:**
- Load (copy to) AH with the low byte the flag register.
- [AH] ← [Flags low byte]
- Eg. LAHF
- **SAHF:**
- Store (copy) AH register to low byte of flag register.
- [Flags low byte] ← [AH]
- Eg. SAHF

PUSHF & POPF Instructions

- **PUSHF:**
- Copy flag register to top of stack.
- [SP] ← [SP] – 2
- [[SP]] ← [Flags]
- Eg. PUSHF
- **POPF :**
- Copy word at top of stack to flag register.
- [Flags] ← [[SP]]
- [SP] ← [SP] + 2

- XCHG AX, DX Exchange word in AX with word in DX
- XCHG BL, CH Exchange byte in BL with byte in CH
- XCHG AL, PRICES [BX] Exchange byte in AL with byte in memory at EA = PRICE [BX] in DS.

- LEA BX, PRICES ;Load BX with offset of PRICE in DS
- LEA BP, SS: STACK_TOP ;Load BP with offset of STACK_TOP in SS
- LEA CX, [BX][DI] ;Load CX with EA = [BX] + [DI]

- LDS BX, [4326] ;Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.
- LDS SI, SPTR ;Copy content of memory at displacement SPTR and SPTR + 1 in DS to SI register. Copy content of memory at displacements SPTR + 2 and SPTR + 3 in DS to DS register. DS: SI now points at start of the desired string.

- LES BX, [789AH] ;Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH and 789DH in DS is copied to ES register.
- LES DI, [BX] ;Copy content of memory at offset [BX] and offset [BX] + 1 in DS to DI register. Copy content of memory at offset [BX] + 2 and [BX] + 3 to ES register.

Arithmetic Instructions

- Used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.
- Addition instructions
  - **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.

ADD Instruction

- The add instruction adds the contents of the source operand to the destination operand.
- It adds a byte to byte or a word to word.
- It effects AF, CF, OF, PF, SF, ZF flags.
  - Eg. ADD AX, 0100H
  - ADD AX, BX
  - ADD AX, [SI]
  - ADD AX, [5000H]
  - ADD [5000H], 0100H
  - ADD 0100H

ADC : Add with Carry

- This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.
- It adds the two operands with CF.
- It effects AF, CF, OF, PF, SF, ZF flags.
  - Eg. ADC 0100H
  - ADC AX, BX
  - ADC AX, [SI]
  - ADC AX, [5000]
  - ADC [5000], 0100H

- **ADD AL, 74H** ;Add immediate number 74H to content of AL. Result in AL
- **ADC CL, BL** ;Add content of BL plus carry status to content of CL
- **ADD DX, BX** ;Add content of BX to content of DX
- **ADD DX, [SI]** ;Add word from memory at offset [SI] in DS to content of DX
- **ADC AL, PRICES [BX]** ;Add byte from effective address PRICES [BX] plus carry status to content of AL
- **ADD AL, PRICES [BX]** ;Add content of memory at effective address PRICES [BX] to AL

INC Instructions

- **INC : Increment**
- This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.
- It increments the byte or word by one.
- The operand can be a register or memory location.
- It effects AF, OF, PF, SF, ZF flags.
- CF is not effected.
  - Eg. INC AX
  - INC [BX]
  - INC [5000H]

DEC Instructions

- **DEC : Decrement**
- The decrement instruction subtracts 1 from the contents of the specified register or memory location.
- It decrements the byte or word by one.
- The operand can be a register or memory location.
- It effects AF, OF, PF, SF, ZF flags.
- CF is not effected.
  - Eg. DEC AX
  - DEC [5000H]

- **AAA (ASCII Adjust after Addition):**
  - The data entered from the terminal is in ASCII format.
  - In ASCII, 0 – 9 are represented by 30H – 39H.
  - This instruction allows us to add the ASCII codes.
  - This instruction does not have any operand.
- **Other ASCII Instructions:**
  - **AAS (ASCII Adjust after Subtraction)**
  - **AAM (ASCII Adjust after Multiplication)**
  - **AAD (ASCII Adjust Before Division)**

- **DAA (Decimal Adjust after Addition)**
  - It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number.
  - It only works on AL register.
- **DAS (Decimal Adjust after Subtraction)**
  - It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number.
  - It only works on AL register.

Subtraction instructions

- **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.

SUB : Subtract

- The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.
- It subtracts a byte from byte or a word from word.
- It effects AF, CF, OF, PF, SF, ZF flags.
- For subtraction, CF acts as borrow flag.
  - Eg. SUB AX, 0100H
  - SUB AX, BX
  - SUB AX, [5000H]
  - SUB [5000H], 0100H

SBB : Subtract with Borrow

- The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand
- It subtracts the two operands and also the borrow from the result.
- It effects AF, CF, OF, PF, SF, ZF flags.
  - Eg. SBB AX, 0100H
  - SBB AX, BX
  - SBB AX, [5000H]
  - SBB [5000H], 0100H

NEG & CMP Instructions

- **NEG : Negate**
  - It creates 2's complement of a given number.
  - The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.
  - That means, it changes the sign of a number.
    - Eg. NEG AL
    - AL = 0011 0101 35H Replace number in AL with its 2's complement AL = 1100 1011 = CBH
- **CMP : Compare**
  - It compares two specified bytes or words.
  - The Src and Des can be a constant, register or memory location.
  - Both operands cannot be a memory location at the same time.
  - The comparison is done simply by internally subtracting the source from destination.
  - The value of source and destination does not change, but the flags are modified to indicate the result.

- SUB CX, BX CX – BX; Result in CX
- SBB CH, AL Subtract content of AL and content of CF from content of CH. Result in CH
- SUB AX, 3427H Subtract immediate number 3427H from AX
- SBB BX, [3427H] Subtract word at displacement 3427H in DS and content of CF from BX
- SUB PRICES [BX], 04H Subtract 04 from byte at effective address PRICES [BX], if PRICES is declared with DB; Subtract 04 from word at effective address PRICES [BX], if it is declared with DW.
- SBB CX, TABLE [BX] Subtract word from effective address TABLE [BX] and status of CF from CX.
- SBB TABLE [BX], CX Subtract CX and status of CF from word in memory at effective address TABLE[BX].

Multiplication instructions

- **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.

- **MUL Src:**
  - It is an unsigned multiplication instruction.
  - It multiplies two bytes to produce a word or two words to produce a double word.
  - AX = AL * Src
  - DX : AX = AX * Src
  - This instruction assumes one of the operand in AL or AX.
  - Src can be a register or memory location.
- **IMUL Src:**
- It is a signed multiplication instruction.

- MUL BH Multiply AL with BH; result in AX
- MUL CX Multiply AX with CX; result high word in DX, low word in AX
- MUL BYTE PTR [BX] Multiply AL with byte in DS pointed to by [BX]
- MUL FACTOR [BX] Multiply AL with byte at effective address FACTOR [BX], if it is declared as type byte with DB. Multiply AX with word at effective address FACTOR [BX], if it is declared as type word with DW.
  - MOV AX, MCAND_16 Load 16-bit multiplicand into AX
  - MOV CL, MPLIER_8 Load 8-bit multiplier into CL
  - MOV CH, 00H Set upper byte of CX to all 0's
  - MUL CX AX times CX; 32-bit result in DX and AX

- Imul
  - Multiplies a signed byte from source with a signed byte in AL or a signed word from some source with a signed word in AX.

- IMUL : multiplies a signed byte from source with a signed byte in AL or a signed word from some source with a signed word in AX
- Source can be a register or a memory location
- When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX.
- When a word from source is multiplied by AX, the result is put in DX and AX.
- If the magnitude of the product does not require all the bits of the destination, the unused byte / word will be filled with copies of the sign bit.
- If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0; If it contains a part of the product, CF and OF will both be 1. AF, PF, SF and ZF are undefined after IMUL.

- IMUL BH Multiply signed byte in AL with signed byte in BH; result in AX.
- IMUL AX Multiply AX times AX; result in DX and AX
- MOV CX, MULTIPLIER Load signed word in CX MOV AL, MULTIPLICAND Load signed byte in AL CBW Extend sign of AL into AH IMUL CX Multiply CX with AX; Result in DX and AX

Division instruction

- **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.

- **DIV Src:**
  - It is an unsigned division instruction.
  - It divides word by byte or double word by word.
  - The operand is stored in AX, divisor is Src and the result is stored as: AH = remainder AL = quotient
- **IDIV Src:**
  - It is a signed division instruction.

- **CBW (Convert Byte to Word):**
  - This instruction converts byte in AL to word in AX.
  - The conversion is done by extending the sign bit of AL throughout AH.
- **CWD (Convert Word to Double Word):**
  - This instruction converts word in AX to double word in DX : AX.
  - The conversion is done by extending the sign bit of AX throughout DX.

Bit Manipulation Instructions

- These instructions are used at the bit level.
  - These instructions can be used for:
  - Testing a zero bit
  - Set or reset a bit
  - Shift bits across registers

Bit Manipulation Instructions

- **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.

- **NOT Src:**
  - It complements each bit of Src to produce 1's complement of the specified operand.
  - The operand can be a register or memory location.
- **AND Des, Src:**
  - It performs AND operation of Des and Src.
  - Src can be immediate number, register or memory location.
  - DES can be register or memory location. (Both operands cannot be memory locations at the same time.)
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

- **OR Des, Src:**
  - It performs OR operation of Des and Src.
  - Src can be immediate number, register or memory location.
  - Des can be register or memory location. (Both operands cannot be memory locations at the same time.)
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

- **XOR Des, Src:**
  - It performs XOR operation of Des and Src.
  - Src can be immediate number, register or memory location.
  - Des can be register or memory location. (Both operands cannot be memory locations at the same time.)
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

Shift operations instructions

- **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.

- **SHL Des, Count:**
  - It shift bits of byte or word left, by count.
  - It puts zero(s) in LSBs.
  - MSB is shifted into carry flag.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

- **SHR Des, Count:**
  - It shift bits of byte or word right, by count.
  - It puts zero(s) in MSBs.
  - LSB is shifted into carry flag.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Rotate instructions

- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** – Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

- **ROL Des, Count:**
- It rotates bits of byte or word left, by count.
- MSB is transferred to LSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost

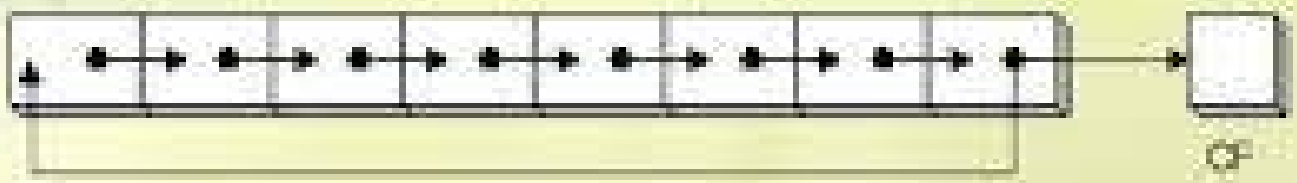

```
MOV AL,11110000b
ROL AL,1                    ; AL = 11100001b

MOV DL,3Fh
ROL DL,4                    ; DL = F3h
```

- **ROR Des, Count:**
- It rotates bits of byte or word right, by count.
- LSB is transferred to MSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost

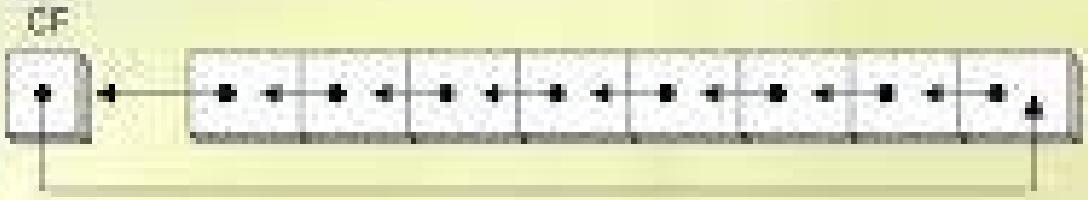



```
MOV AL,11110000b
ROR AL,1                    ; AL = 01111000b

MOV DL,3Fh
ROR DL,4                    ; DL = F3h
```

RCL Instruction

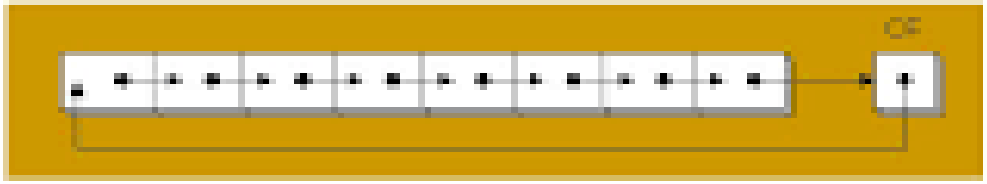
- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag




```
CLC                 ; CF = 0
MOV BL,88H          ; CF,BL = 0 10001000b
RCL BL,1            ; CF,BL = 1 00010000b
RCL BL,1            ; CF,BL = 0 00100001b
```

RCR Instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag




```
stc                     ; CF = 1
mov ah,10h              ; CF,AH = 00010000 1
rcr ah,1                ; CF,AH = 10001000 0
```

String Instructions

- String in assembly language is just a sequentially stored bytes or words.
- There are very strong set of string instructions in 8086.
- By using these string instructions, the size of the program is considerably reduced.

String Instructions

- **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.

- **CMPS Des, Src:**
  - It compares the string bytes or words.
- **SCAS/SCASB/SCASW String:**
  - It scans a string.
  - It compares the String with byte in AL or with word in AX.
- **LODS/LODSB/LODSW**
  - It loads a string.
  - It compares the String with byte in AL or with word in AX.

- **MOVS / MOVSB / MOVSW:**
- It causes moving of byte or word from one string to another.
- In this instruction, the source string is in Data Segment and destination string is in Extra Segment.
- SI and DI store the offset values for source and
- destination index.

- **REP (Repeat):**
- This is an instruction prefix.
- It causes the repetition of the instruction until CX becomes zero.
  - E.g.: REP MOVSB STR1, STR2
  - It copies byte by byte contents.
  - REP repeats the operation MOVSB until CX becomes zero.

Program Execution Transfer Instructions (Branch and Loop Instructions)

- These instructions cause change in the sequence of the execution of instruction.
- This change can be through a condition or sometimes unconditional.
- The conditions are represented by flags.
- Unconditional transfer
- **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.

- **CALL Des:**
  - This instruction is used to call a subroutine or function or procedure.
  - The address of next instruction after CALL is saved onto stack.
- **RET:**
  - It returns the control from procedure to calling program.
  - Every CALL instruction should have a RET.

- **JMP Des:**
- This instruction is used for unconditional jump from one place to another.
- **Jxx Des (Conditional Jump):**
  - All the conditional jumps follow some conditional statements or any instruction that affects the flag.

Conditional transfer

- **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

Conditional Jump Table

Mnemonic	Meaning	Jump Condition
JA	Jump if Above	CF = 0 and ZF = 0
JAE	Jump if Above or Equal	CF = 0
JB	Jump if Below	CF = 1
JBE	Jump if Below or Equal	CF = 1 or ZF = 1
JC	Jump if Carry	CF = 1
JE	Jump if Equal	ZF = 1
JNC	Jump if Not Carry	CF = 0
JNE	Jump if Not Equal	ZF = 0
JNZ	Jump if Not Zero	ZF = 0
JPE	Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JZ	Jump if Zero	ZF = 1

- **Loop Des:**
- This is a looping instruction.
- The number of times looping is required is placed in the CX register.
- With each iteration, the contents of CX are decremented.
- ZF is checked whether to loop again or not.

Processor Control Instructions

- These instructions control the processor itself.
- 8086 allows to control certain control flags that:
  - Causes the processing in a certain direction processor synchronization
  - If more than one microprocessor attached.

Processor Control Instructions

- **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.

- **STC:**
  - It sets the carry flag to 1.
- **CLC:**
  - It clears the carry flag to 0.
- **CMC:**
  - It complements the carry flag.
- **STD:**
  - It sets the direction flag to 1.
  - If it is set, string bytes are accessed from higher memory address to lower memory address.
- **CLD:**
  - It clears the direction flag to 0.
  - If it is reset, the string bytes are accessed from lower memory address to higher memory address.

Iteration Control Instructions

- **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

- **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

Executing 8086 programs

- Masm file_name.asm;
- Link file_name.obj;
- Afdebug file_name.exe

- Or
- File_name at the command prompt

datas SEGMENT
- Hello DB “Hello world $”

datas ENDS
staks SEGMENT
- DB 100 dup()

staks ENDS
codes SEGMENT
ASSUME CS: codes, DS: datas, SS: staks
Start:
- MOV AX, SEG datas
- MOV DS, AX
- MOV CX, SEG staks
- MOV SS, CX
- MOV AH, 09h
- MOV DX, OFFSET Hello
- INT 21h
- MOV AX, 4C00h
- INT 21h

codes ENDS
END START

Add two numbers 05h and 04h and display it on monitor

- #include <stdio.h>
- void main()
- {
- char aa = 05, bb = 04, sum ;
- sum = aa + bb ;
- printf(”%d”, sum);
- }

```
datas SEGMENT
    aa DB 05h
    bb DB 04h
    sum DB ?
datas ENDS
```

- stacks SEGMENT
- DB 100 DUP(0)
- stacks ENDS

```
codes SEGMENT
    ASSUME CS: codes, DS: datas,
    SS: stacks
    start:          ;; beginning address of program ({)
    MOV ax, SEG datas
    MOV ds, ax
    MOV cx, SEG stacks
    MOV ss, cx
    ; data processing
    MOV al, aa
    MOV cl, bb
    ADD al, cl
    MOV sum, al
```

```
ADD al, 30h
MOV dl, al
MOV ah, 02h
INT 21h
;; terminate program
MOV ax, 4C00h
INT 21h
codes ENDS
END start     ; (})
```