INSTRUCTION SET of 8086

Instruction set

The 8086 Instruction set is classified as

- Data transfer instructions
- Arithmetic instructions
- Logic & Bit manipulation instructions
- Branch / Control transfer instructions
- String manipulation instructions
- Processor control instructions

Data Transfer Instructions

- MOV: Copies data from source to destination.

  Source→Immediate/Reg/Mem

  Dest→ Reg/Mem

```
Ex: MOV  AX, 1234H
    MOV  AX, BX
    MOV AX,[2000H]
    MOV AX,[SI]
    MOV AX,50H[BX]
```

- PUSH: Pushes the content of source on to the stack. After the execution, SP is decremented by 2 and the source content is stored at stack top.

```
Ex:PUSH  AX
 SP ← SP-2
 [SP] ← AX
```

- POP: Pop a word from stack top to specified register

The content of stack top is moved to destination & SP is incremented by 2

Ex:POP AX

AX ← [SP]

SP ← SP+2

- XCHG : Exchange the contents of source and destination

Ex:XCHG BX,AX

XCHG [5000],AX

❑ IN: Read data from specified input port to Accumulator

Ex: IN AL, 80H ; It reads one byte of data from I/O port address 80H to AL

```
MOV DX,1234H
IN AL, DX
```

- OUT: Send data from Accumulator to specified output port
- Ex: OUT 82H, AL ; It sends one byte of data from AL to I/O port address 82H

```
 MOV DX,1234H
OUT DX, AL
```

- XLAT: Translate
- This translate instruction is used for finding out codes in code conversion problems, using lookup table technique.

Ex: XLAT: Translate byte to AL.

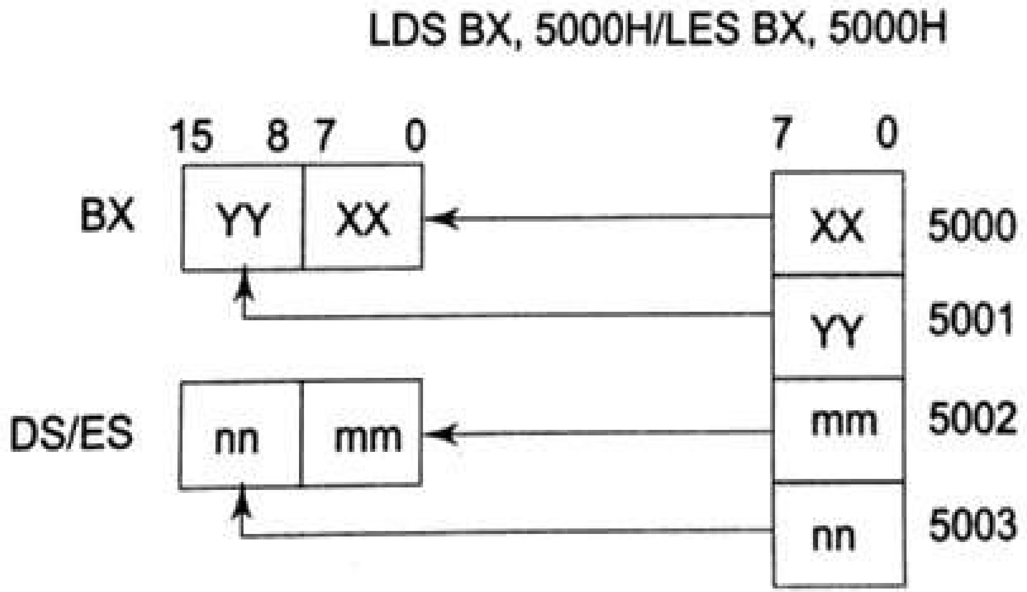
- LEA: Load Effective Address
- Loads of the effective address formed by destination operand into the specified source register.

  Ex: LEA BX,ADR

  ```
  LEA SI,ADR[BX]
  ```

- LDS/LES: Load pointer to DS or ES
- Load specified register and DS registers with contents of two words from the effective address

Ex: LDS BX,5000H/ LES BX,5000H



- LAHF: Load AH from lower byte of Flag
- SAHF: Store AH to lower byte of Flag
- PUSHF:PUSH Flags to stack
- POPF : POP Flags from stack

Arithmetic instructions

- . ADD: the content of source is added to the destination and result will be stored in destination

Ex:

```
ADD AX,0100H
ADD AX,BX
 ADD AX,[2000H]
ADD AX,[SI]
ADD AX, [BP]
```

- ADC: the content of source along with carry are added to the destination and result will be stored in destination.

- SUB:the content of source is subtracted to the destination and result will be stored in destination
- SBB:the content of source along with borrow are subtracted to the destination and result will be stored in destination.

- INC: Increases the contents of specified register or memory location by 1.
- DEC: Decreases the contents of specified register or memory location by 1.

- CMP: it compares destination and source operands.

  If the destination < source then CF is set(1).

  If the destination > source then CF is reset(0).

  If the destination = source then ZF is set(1).

```
Ex: CMP BX,0100H
    CMP [5000H],0100H
```

- NEG: Negate

  It forms the 2’s complement of the specified destination in the instruction.

- **AAA: ASCII Adjust after Addition**

  It is executed after an ADD instruction that adds two ASCII operands to give byte result in AL.

  AAA converts the result in AL into unpacked decimal digits.

- **AAS : ASCII Adjust after Subtraction**

  It is executed after SUB instruction that subtracts two ASCII operands to give byte result in AL.

  AAS converts the result in AL into unpacked decimal digits.

- **AAD : ASCII adjust before Division**

  It converts two unpacked BCD digits in AH and AL to the equivalent packed binary number in AL.

- Ex: AX = 0508

  AAD result in AL = 3AH

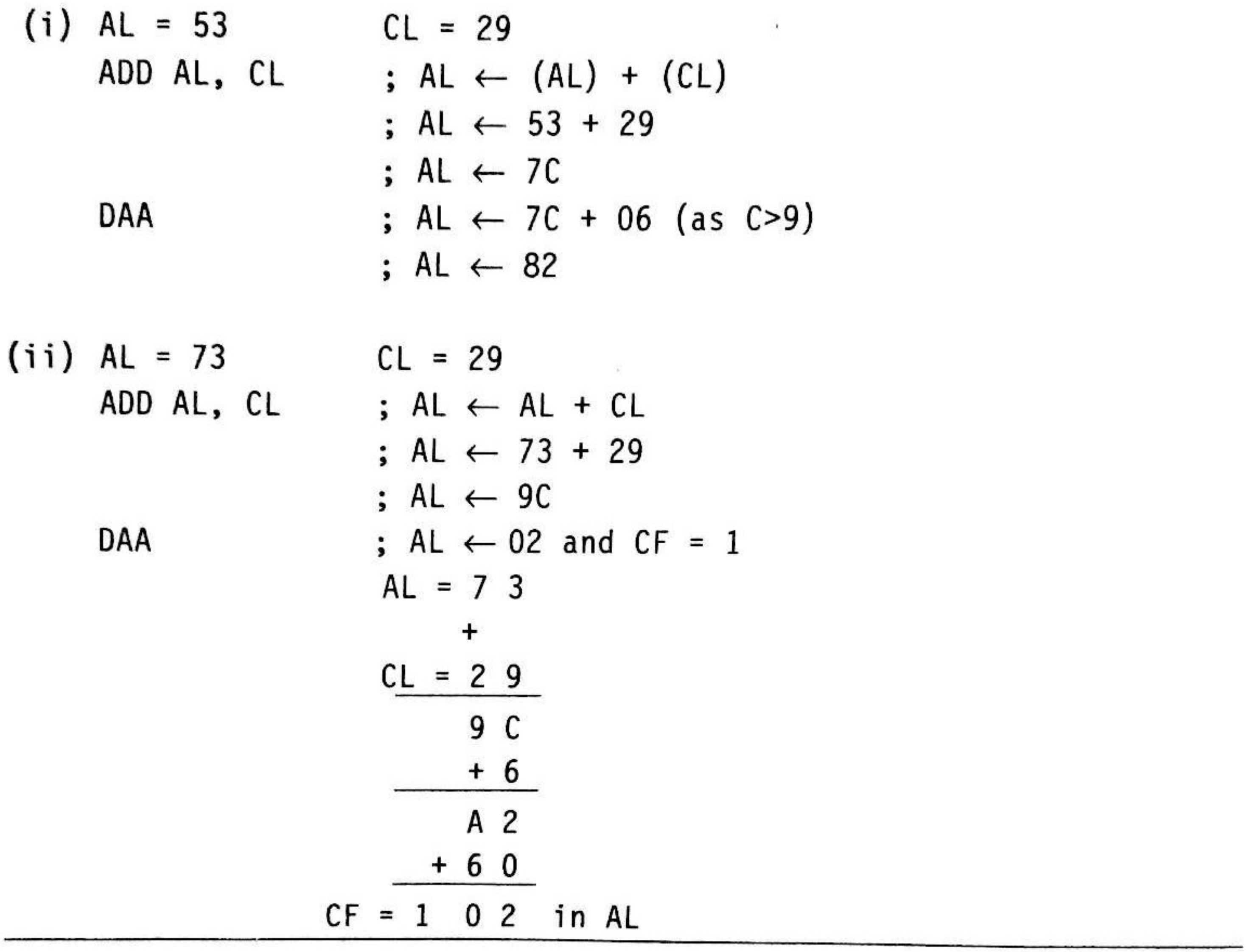
- **AAM : ASCII adjust after Multiplication**

It is executed after MUL instruction that multiplies two unpacked operands to give byte result in AL.

AAM converts the result in AL into unpacked decimal digits.

- **DAA: Decimal Adjust after Addition**
- It converts the result of addition of two packed BCD numbers to a valid BCD number. The result has to be in AL.
- If the lower nibble of AL>9 then it adds 06.
- If the higher nibble of AL>9 then it adds 60.

- Ex:

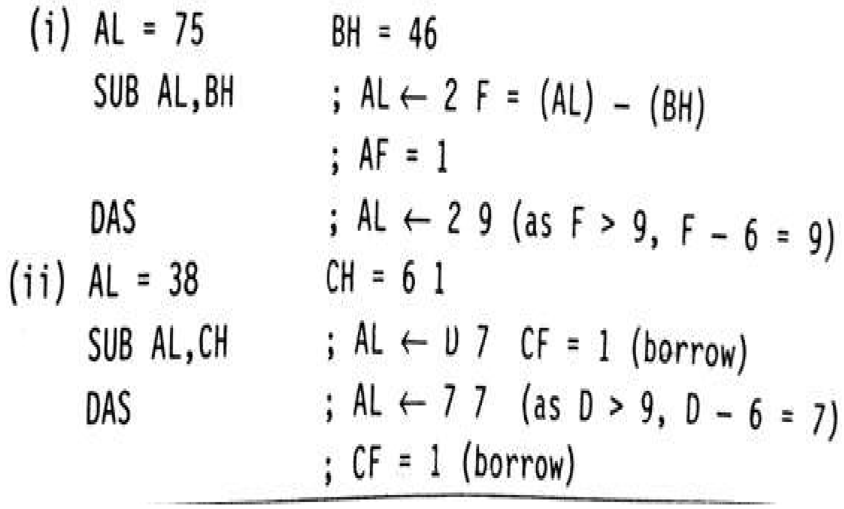


```
(i)  AL = 53          CL = 29
     ADD AL, CL       ; AL ← (AL) + (CL)
                      ; AL ← 53 + 29
                      ; AL ← 7C
     DAA              ; AL ← 7C + 06 (as C>9)
                      ; AL ← 82

(ii) AL = 73          CL = 29
     ADD AL, CL       ; AL ← AL + CL
                      ; AL ← 73 + 29
                      ; AL ← 9C
     DAA              ; AL ← 02 and CF = 1
                      AL = 7 3
                           +
                      CL = 2 9
                        ---------
                           9 C
                           + 6
                        ---------
                           A 2
                         + 6 0
                        ---------
                    CF = 1  0 2  in AL
```

DAS: Decimal Adjust after Subtraction

- It converts the result of subtraction of two packed BCD numbers to a valid BCD number. The result has to be in AL.
- If the lower nibble of AL>9 then it subtracts 06.
- If the higher nibble of AL>9 then it subtracts 60.



```
(i)  AL = 75        BH = 46
     SUB AL,BH      ; AL ← 2 F = (AL) - (BH)
                    ; AF = 1
     DAS            ; AL ← 2 9 (as F > 9, F - 6 = 9)
(ii) AL = 38        CH = 6 1
     SUB AL,CH      ; AL ← D 7  CF = 1 (borrow)
     DAS            ; AL ← 7 7  (as D > 9, D - 6 = 7)
                    ; CF = 1 (borrow)
```

- MUL: Unsigned Multiplication

Multiplies the contents of AL or AX with an unsigned byte or word .

The most significant word of the result is stored in DX and the least significant word of the result is stored AX.

Ex:

- MUL BH; (AX) ← (AL)*(BH)
- MUL CX; (DX) (AX) ← (AX)*(CX)

- IMUL: Signed Multiplication

Multiplies the contents of AL or AX with an signed byte or word .

The most significant word of the result is stored in DX and the least significant word of the result is stored AX.

Ex:

- IMUL BH; (AX) ← (AL)*(BH)
- IMUL CX; (DX) (AX) ← (AX)*(CX)

- ## CBW: Convert Byte to Word

- It converts a signed byte to a signed word. It copies the sign bit of a byte to all the bits in the higher byte of the result word.

- ## CWD: Convert Word to Double word

- It copies sign bit of AX to all the bits of the DX register.

- ### DIV: Unsigned Division:

- It divides an unsigned word or double word by a 8 bit or 16 bit operand . The dividend must be in AX for 8-bit operation and in DX:AX pair for 16-bit operation.
- The quotient will be in AL or AX and the remainder will be in AH or DX.
- Ex: DIV BL → AHAL/BL

  DIV BX → DXAX/BX

- IDIV: Signed Division
- It divides an signed word or double word by a 8 bit or 16 bit operand . The dividend must be in AX for 8-bit operation and in DX:AX pair for 16-bit operation.
- The quotient will be in AL or AX and the remainder will be in AH or DX.
- Ex: IDIV BL → AHAL/BL

  IDIV BX → DXAX/BX

LOGICAL INSTRUCTIONS

- AND:
- It performs bitwise AND operation on Source and Destination operands.

```
Ex: AND AX, 0008H
    AND  AX, BX
    AND AX,[2000H]
    AND [5000H],DX
```

- OR:
- It performs bitwise OR operation on Source and Destination operands.
- XOR:
- It performs bitwise XOR operation on Source and Destination operands.

- NOT: Logical invert
- It complements the content of a register or a memory location , bit by bit.
- Ex : NOT AX

  NOT [5000H]
- TEST: Logical AND
- It performs bit wise logical AND operation on the two operands.
- Ex: TEST AX,BX

  TEST [0500H],06H

- SHL/SAL: Shift left/Shift Arithmetic left
- These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits.

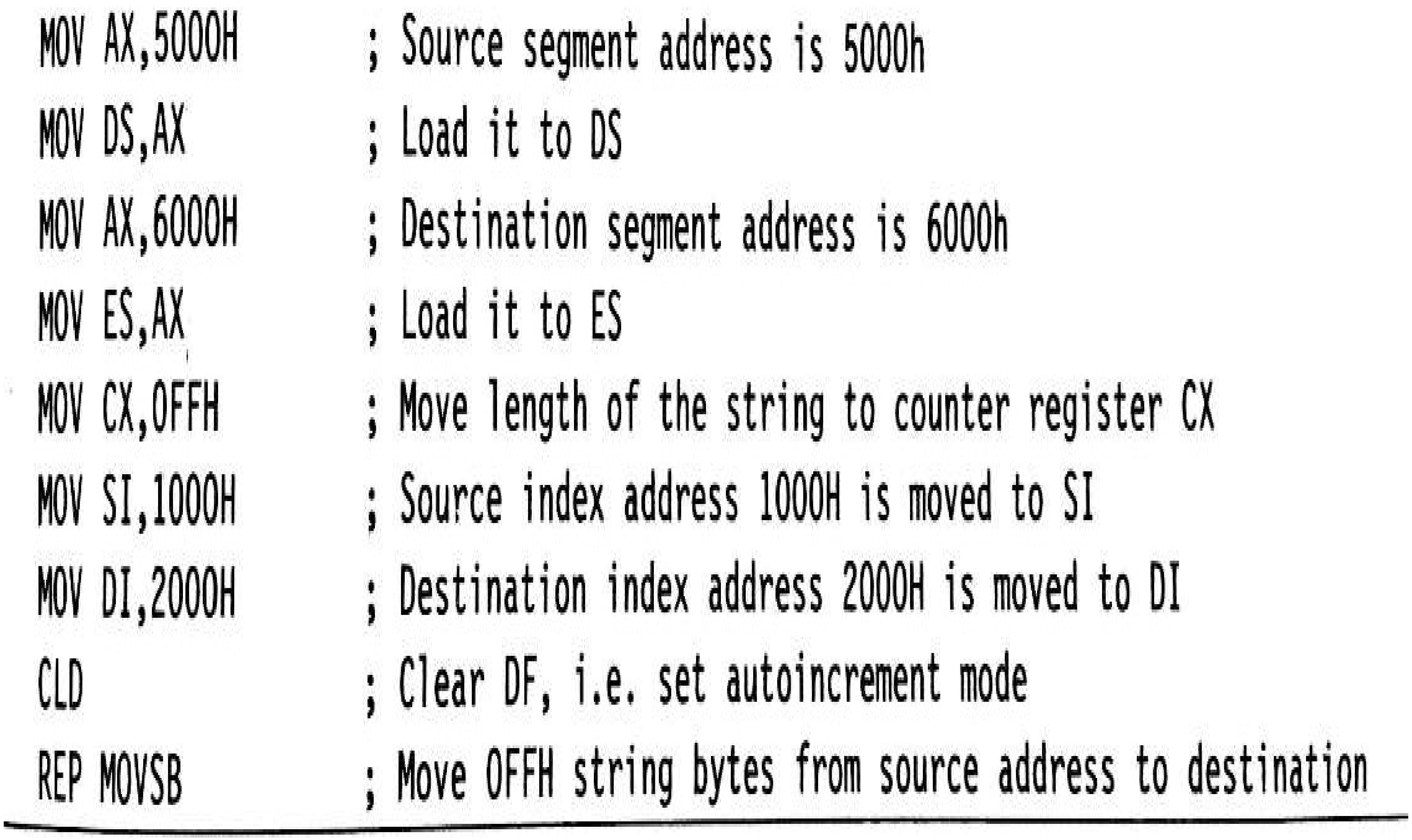
- SHR: Shift Logical Right
- This instruction shift the operand word or byte bit by bit to the right and insert zeros in the newly introduced most significant bits.

- SAR: Shift Arithmetic right
- This instruction shift the operand word or byte bit by bit to the right and it inserts most significant of the operand in the newly introduced most significant bits.

- ROL: Rotate Left without carry
- This instruction rotates the contents of destination operand to the left (bit wise) either by one or count specified in CL register, excluding carry
- ROR: Rotate Right without carry
- This instruction rotates the contents of destination operand to the right (bit wise) either by one or count specified in CL register, excluding carry
- RCL: Rotate Left through carry
- This instruction rotates the contents of destination operand to the left through carry (bit wise) either by one or count specified in CL register.
- RCR: Rotate Right through carry
- This instruction rotates the contents of destination operand to the right through carry (bit wise) either by one or count specified in CL register.

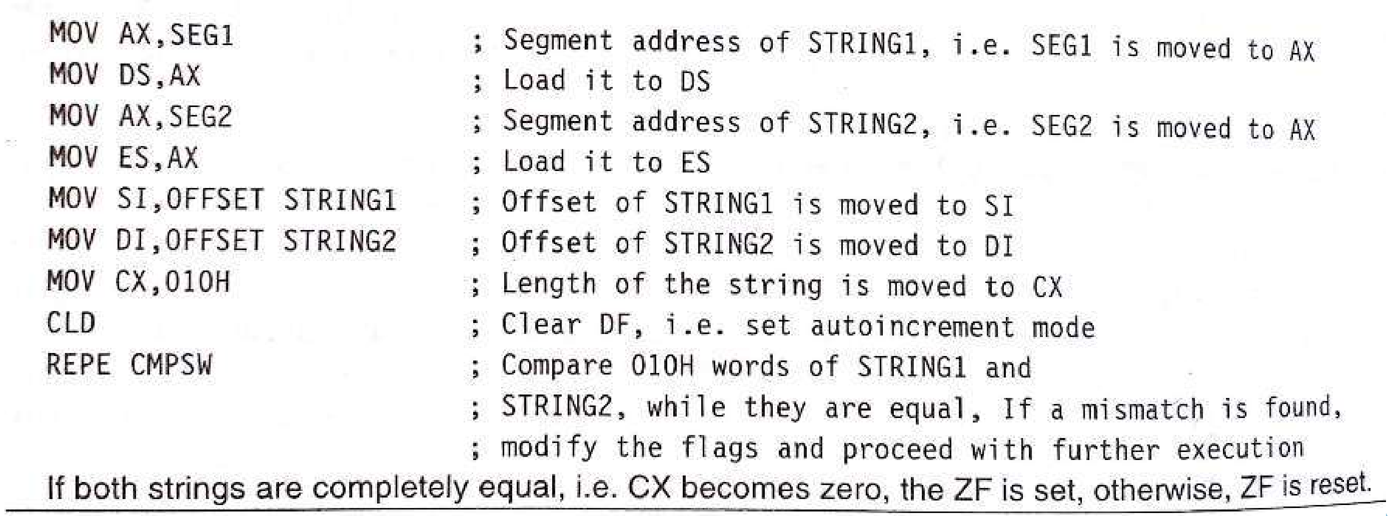
String manipulation instructions

- MOVSB/MOVSW: move string byte/word
- This instruction moves a string of bytes/words pointed by DS:SI pair to the memory location pointed by ES:DI pair.
- Each time it is executed, the index registers are automatically updated and CX is decremented.
- REP: Repeat Instruction Prefix
- It is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero.
- REPE/REPZ: Repeat operation while equal/zero
- REPNE/REPNZ: Repeat operation while not equal/not zero



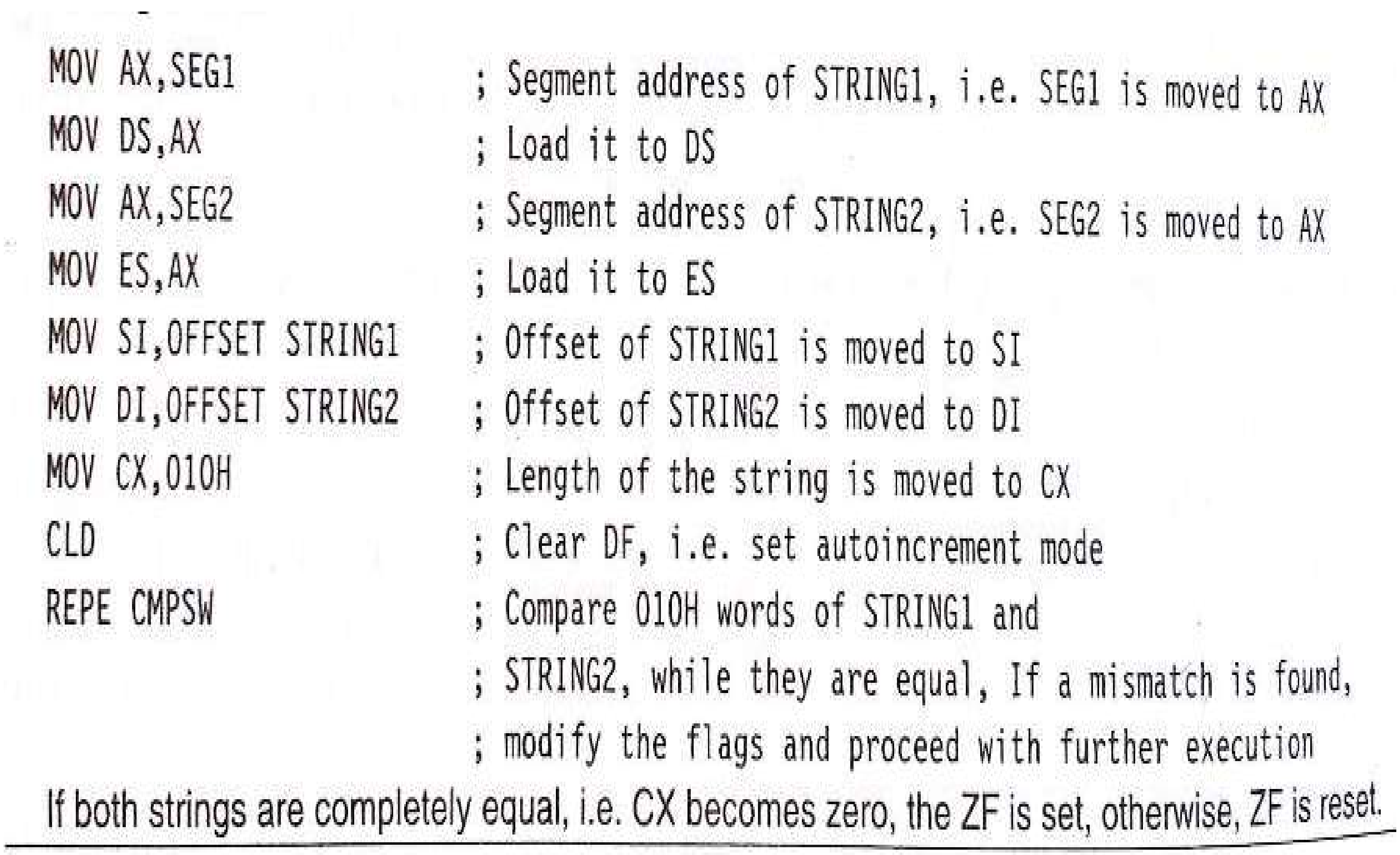
```
MOV AX,5000H        ; Source segment address is 5000h
MOV DS,AX           ; Load it to DS
MOV AX,6000H        ; Destination segment address is 6000h
MOV ES,AX           ; Load it to ES
MOV CX,0FFH         ; Move length of the string to counter register CX
MOV SI,1000H        ; Source index address 1000H is moved to SI
MOV DI,2000H        ; Destination index address 2000H is moved to DI
CLD                 ; Clear DF, i.e. set autoincrement mode
REP MOVSB           ; Move 0FFH string bytes from source address to destination
```

- CMPS: Compare String Byte or String Word
- It compares two strings stored in DS:SI and ES:DI.
- The length of the string must be stored in CX register.
- REP instruction prefix is used to repeat the operation till CX becomes zero.



```
MOV  AX,SEG1              ; Segment address of STRING1, i.e. SEG1 is moved to AX
MOV  DS,AX                ; Load it to DS
MOV  AX,SEG2              ; Segment address of STRING2, i.e. SEG2 is moved to AX
MOV  ES,AX                ; Load it to ES
MOV  SI,OFFSET STRING1    ; Offset of STRING1 is moved to SI
MOV  DI,OFFSET STRING2    ; Offset of STRING2 is moved to DI
MOV  CX,010H              ; Length of the string is moved to CX
CLD                       ; Clear DF, i.e. set autoincrement mode
REPE CMPSW                ; Compare 010H words of STRING1 and
                          ; STRING2, while they are equal, If a mismatch is found,
                          ; modify the flags and proceed with further execution
```

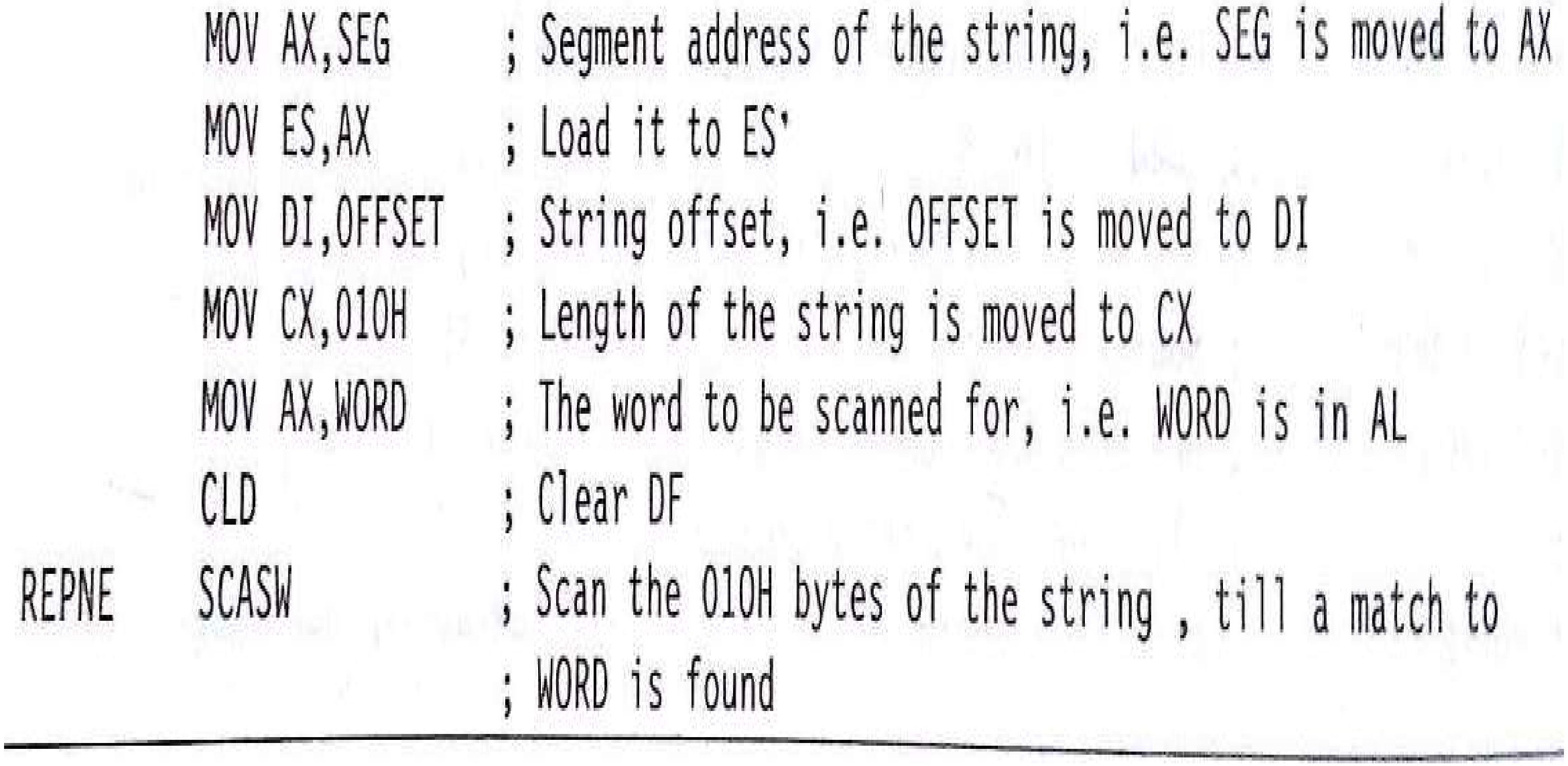
If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.



```
MOV  AX,SEG1              ; Segment address of STRING1, i.e. SEG1 is moved to AX
MOV  DS,AX                ; Load it to DS
MOV  AX,SEG2              ; Segment address of STRING2, i.e. SEG2 is moved to AX
MOV  ES,AX                ; Load it to ES
MOV  SI,OFFSET STRING1    ; Offset of STRING1 is moved to SI
MOV  DI,OFFSET STRING2    ; Offset of STRING2 is moved to DI
MOV  CX,010H              ; Length of the string is moved to CX
CLD                       ; Clear DF, i.e. set autoincrement mode
REPE CMPSW                ; Compare 010H words of STRING1 and
                          ; STRING2, while they are equal, If a mismatch is found,
                          ; modify the flags and proceed with further execution
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

- SCAS: Scan String Byte or Word
- It scans a string of bytes or words for an operand byte or word specified in the register AL or AX.
- The string is pointed by ES:DI register pair.
- If a match to the specified operand is found in the string then execution stops and zero flag is set.



```
         MOV AX,SEG       ; Segment address of the string, i.e. SEG is moved to AX
         MOV ES,AX        ; Load it to ES'
         MOV DI,OFFSET    ; String offset, i.e. OFFSET is moved to DI
         MOV CX,010H      ; Length of the string is moved to CX
         MOV AX,WORD      ; The word to be scanned for, i.e. WORD is in AL
         CLD              ; Clear DF
REPNE    SCASW            ; Scan the 010H bytes of the string , till a match to
                          ; WORD is found
```

- LODS: Load String Byte or String Word
- It loads AL/AX register by the content of a string pointed by DS:SI register pair.
- SI is modified automatically depending upon DF.

- STOS: Store String Byte or String Word

- It stores the content of AL/AX register to a location in the string pointed by ES:DI register pair.
- DI is modified automatically depending upon DF.

Control transfer/branching instructions

- CALL: Unconditional Call
- It is used to call a subroutine/procedure from a main program.
- The address of the procedure may be specified directly or indirectly depending upon the addressing mode.
- On execution ,it pushes the incremented IP and CS on to the stack and loads new CS and IP.
- NEAR CALL: the procedure lies in the same segment.
- FAR CALL: the procedure lies in the other segment.

- RET: Return to Main program
- It should be the last instruction of a procedure/subroutine.
- On execution, the previously stored content of IP and CS along with flags are retrieved and the execution of main program continues further.

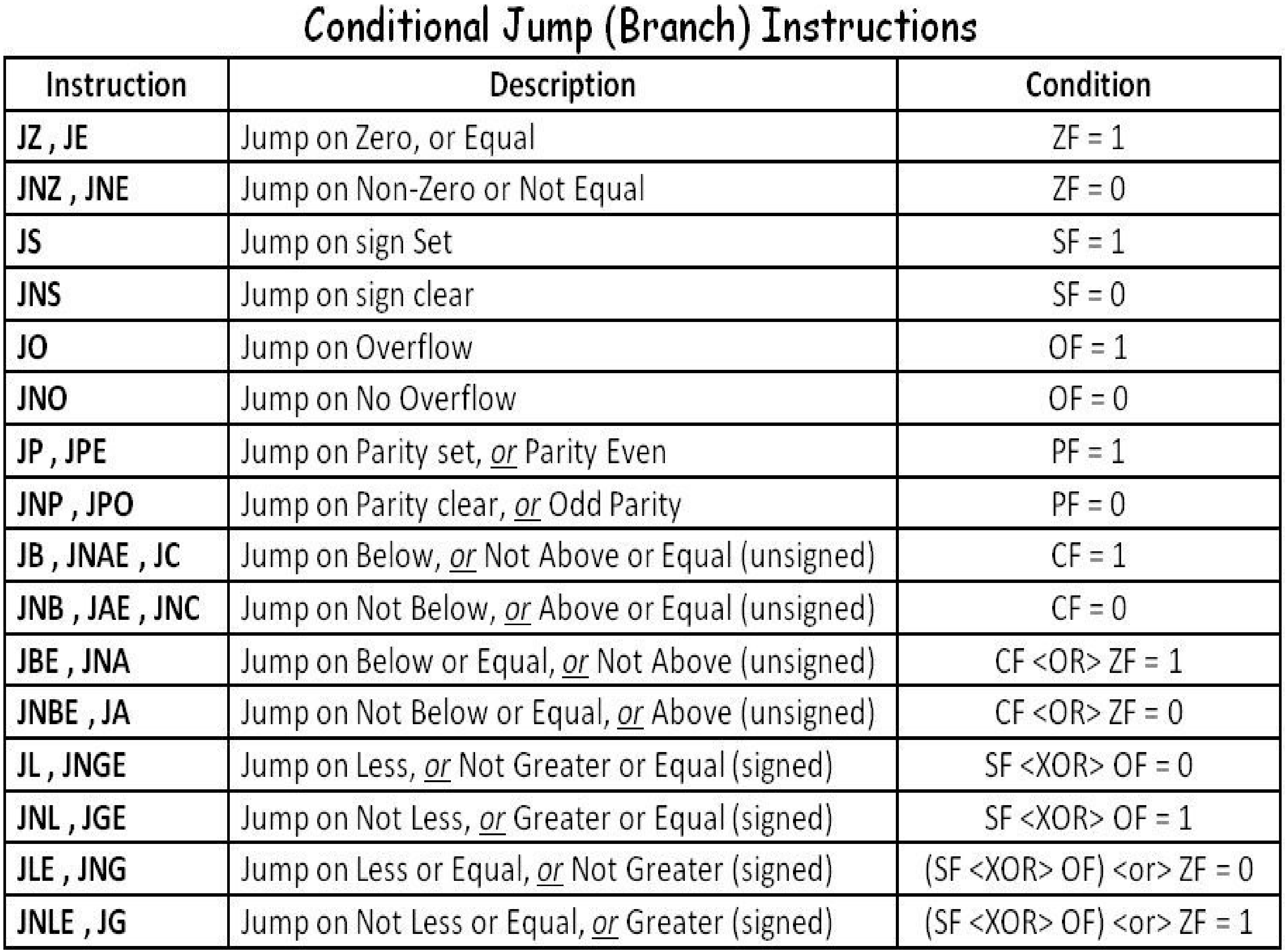
- INT N: Interrupt Type N
- When an INT instruction is executed , the control is transferred to a vector address which is obtained by multiplying Type N with 4.
- At this vector address the CS and IP values are stored.

- IRET: Return from ISR
- It appears at the end of each ISR.
- When it is executed ,the values of IP,CS and flags are retrieved from stack to continue the execution of main program.

- INTO: INTerrupt on Overflow
- It is executed, when the Overflow flag OF is set.
- The new contents of CS and IP are taken from 0000:0010 as this is equivalent to Type 4 interrupt.

- JMP: Unconditional jump
- This instruction unconditionally transfers the control of execution to the specified address using 8-bit or 16-bit displacement or CS:IP.

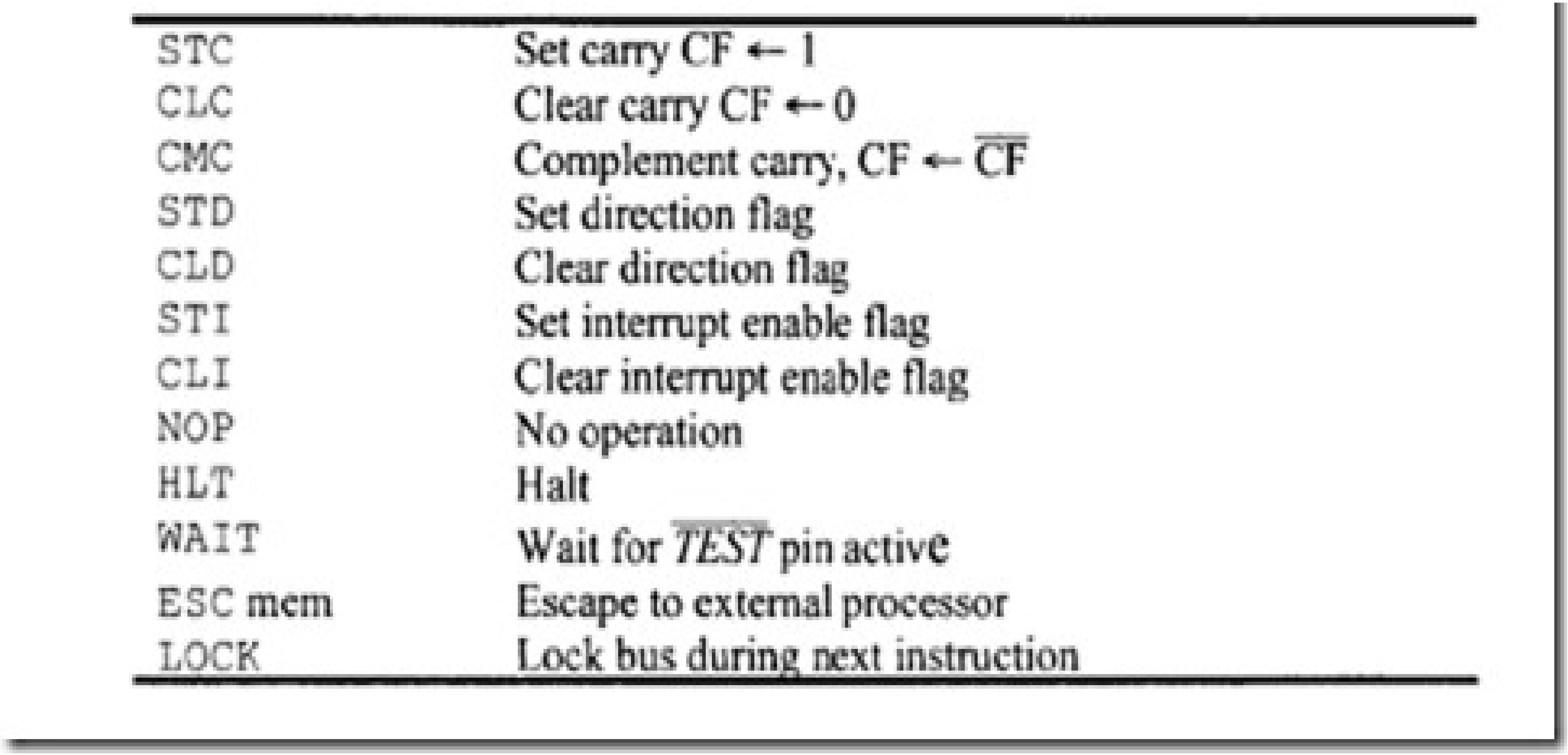
- LOOP: Loop Unconditionally
- This instruction executes a part of the program from the label or address specified in the instruction to Loop instruction ,CX number of times.
- At each iteration ,CX is decremented automatically.
- LOOP: Loop Conditionally
- LOOPZ/LOOPE: **loop** a group of **instructions** till it satisfies ZF = 1 & CX = 0.
- LOOPNZ/LOOPNE: Used to **loop** a group of **instructions** till it satisfies ZF = 0 & CX = 0.



Conditional Jump (Branch) Instructions

Instruction	Description	Condition
JZ , JE	Jump on Zero, or Equal	ZF = 1
JNZ , JNE	Jump on Non-Zero or Not Equal	ZF = 0
JS	Jump on sign Set	SF = 1
JNS	Jump on sign clear	SF = 0
JO	Jump on Overflow	OF = 1
JNO	Jump on No Overflow	OF = 0
JP , JPE	Jump on Parity set, *or* Parity Even	PF = 1
JNP , JPO	Jump on Parity clear, *or* Odd Parity	PF = 0
JB , JNAE , JC	Jump on Below, *or* Not Above or Equal (unsigned)	CF = 1
JNB , JAE , JNC	Jump on Not Below, *or* Above or Equal (unsigned)	CF = 0
JBE , JNA	Jump on Below or Equal, *or* Not Above (unsigned)	CF <OR> ZF = 1
JNBE , JA	Jump on Not Below or Equal, *or* Above (unsigned)	CF <OR> ZF = 0
JL , JNGE	Jump on Less, *or* Not Greater or Equal (signed)	SF <XOR> OF = 0
JNL , JGE	Jump on Not Less, *or* Greater or Equal (signed)	SF <XOR> OF = 1
JLE , JNG	Jump on Less or Equal, *or* Not Greater (signed)	(SF <XOR> OF) <or> ZF = 0
JNLE , JG	Jump on Not Less or Equal, *or* Greater (signed)	(SF <XOR> OF) <or> ZF = 1

Flag manipulation & machine control instructions



STC	Set carry CF ← 1
CLC	Clear carry CF ← 0
CMC	Complement carry, CF ← $\overline{CF}$
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
NOP	No operation
HLT	Halt
WAIT	Wait for $\overline{TEST}$ pin active
ESC mem	Escape to external processor
LOCK	Lock bus during next instruction

Summary

- The various types of instructions of 8086 were studied.

References

- Yu-Cheng Liu, Glenn A. Gibson, “Microcomputer Systems: The 8086 / 8088 Family -Architecture, Programming and Design”, Second Edition, Prentice Hall of India, 2007.
- Doughlas V. Hall, “Microprocessors and Interfacing, Programming and Hardware”, TMH, 2012.

Thank you