Instruction set of 8086

- The instruction set of 8086 can be classified into following groups

1.Data transfer instructions

2. Arithmetic instructions

3. Bit manipulation instructions

4. String instructions

5.Program execution transfer instructions

6. Processor control instructions

- The data transfer instructions can be classified into following categories

1. General purpose byte / word transfer instructions

2.Simple input and output port transfer instructions

3. Special address transfer instructions

4. Flag transfer instructions

Data transfer instructions

- *The functions of data transfer instructions is*

1. Copy the contents of a register to another register.
2. Copy the contents of a register to memory or vice-versa.
3. Load the immediate operand to memory/register
4. Copy the contents of a register/ memory to segment registers (excluding CS register) or vice-versa.
5. Exchange the contents of two registers or register and memory.
6. Copy the contents of accumulator to port or vice-versa.
7. Load the effective address in segment registers.

- The data transfer instructions generally involve two operands i.e. Source operand and Destination operand.

Data transfer instructions

- The Source and Destination operand should be of same size i.e. both the operands should be either 8-bits (byte) or 16-bit (word).
- Only 8-bit word can be moved to a 8-bit register/memory or a 16-bit word can be moved to 16-bit register/memory.
- Moving the contents of 8-bit register to 16-bit register/ memory or vice-versa is illegal.
- The Source can be a register or a memory location or an immediate data.
- The destination can be a register or a memory location.
- In double operand instructions the source and destination cannot refer to memory locations in the same instruction.

Data transfer instructions

- Copying the contents of one memory location to another memory location in a single instruction is not possible (except push instruction).
- The data transfer instructions (except POPF & SAHF instructions) do not affect any flags of 8086.
- While executing the POPF instruction, the previously stored status of flag is restored in the flag register.
- The instruction SAHF is used to modify the content of flag register.
- For XCHG instruction, at least one of the operands must be a register, but neither operand can be a segment register.

Data transfer instructions

- General purpose byte or word transfer instructions:
  - **1) MOV**
  - **2) PUSH**
  - **3) POP**
  - **4) XCHG**
  - **5) XLAT**
- MOV: Copy byte or word from specified source to specified destination.

  Format: MOV <dest>, <source>

  Operation: (dest) ← (source)

  Examples: MOV Reg2, Reg1

  MOV AL, Mem

  MOV Seg reg, Mem

Data transfer instructions

General purpose byte or word transfer instructions

- PUSH: Copy specified word to top of the stack

  Format: PUSH < source >

  Operation: (SP) ← (SP) – 2

  $(PA)_{stack} = (SS) \times 16_{10} + (SP)$

  (PA; PA+1) ← source

  The stack point is decremented by 2 and the contents of the source are transferred to stack memory pointed by stack pointer

  Examples: PUSH Reg16

  PUSH Mem

  PUSH Seg reg

Data transfer instructions

General purpose byte or word transfer instructions

- POP: Copy word from top of the stack to specified location

  Format: POP < destination >

  Operation: (PA) = (SS) X 16_{10} + (SP)

  destination ← (PA; PA+1)

  (SP) ← (SP) + 2

  The contents of 16-bit stack memory pointed by SP is moved to destination and the stack pointer is incremented by 2

  Examples: POP Reg16

  POP Mem

  POP Seg reg

Data transfer instructions

General purpose byte or word transfer instructions

- XCHG: Exchange bytes or words between 2 registers or a register and memory location

  Format: XCHG < destination > , < source >

  Operation: (destination) ⇔ (Source)

  Examples: XCHG Reg1, Reg2

  XCHG Mem, Reg

  XCHG AX, Reg16

Data transfer instructions

General purpose byte or word transfer instructions

- XLAT: Translate a byte in AL, using a table in memory

  Format: XLAT

  Operation: PA = DS X 16_{10} + (BX) + (AL)

  (AL) ← (PA)

  This instruction is used to translate a byte from one code to another code

  The instruction replaces a byte in the AL register with a byte pointed to by BX register in a look up table in memory

  Before executing XLAT instruction, the look up table is to be put into memory and the starting address of the look up table has to be loaded into BX register

  Examples: ASCII value of 0-9 is 30-39 and EBCDIC is 0-9. Hence to convert EBCDIC code in ASCII, the ASCII values of the 0-9 has to be stored say from 2000H, then save 2000H in BX.

Data transfer instructions

Simple input and output port transfer instructions

IN & OUT

- IN : Copy a byte or word from specified port to accumulator.

  Format : IN <Accumulator>, <Source>

  Example : IN AL/AX,[DX]

  (AL/AX) ← (Port)

  the contents of 8-bit port whose address is specified by DX register is transferred to 8-bit accumulator (AL/AX)

  IN AL/AX, addr8

  (AL/AX) ← (addr8)

  The contents of 8-bit port whose address is given in the instruction is transferred to accumulator (AL/AX)

Data transfer instructions

Simple input and output port transfer instructions

- OUT : copy a byte or word from accumulator to specified port.

  Format : OUT <Destination>, AL/AX

  (port) ← AL/AX

  Example: OUT [DX],AL/AX

The contents of accumulator is transferred to the specified port whose address is given in DX register.

OUT addr8,AL/AX

The contents of accumulator are transferred to the port whose address is specified in the instruction.

Data transfer instructions

Special Address transfer instructions

Special address transfer instructions

LEA, LDS & LES

- LEA : Load effective address of operand into specified register.

  LEA Reg16,Mem

  (Reg16) ← EA

The 16-bit register is loaded with the effective address (EA) of the memory location specified by the instruction.

Data transfer instructions

Special Address transfer instructions

- LDS : Load DS register and the other specified register from memory.

  LDS Reg16,Mem

  (Reg16) ← (Mem)

  (DS) ← (Mem+2)

Copies a word from two memory locations into the register specified in the instruction,it then copies a word from the next two memory locations into the DS register.

- LES : Load ES register and the other specified register from memory.

  LES Reg16,Mem

  (Reg16) ← (Mem)

  (ES) ← (Mem+2)

Copies a word from two memory locations into the register specified in the instruction,it then copies a word from the next two memory locations into the ES register.

Data transfer instructions

Flag Transfer instructions

Flag Transfer Instructions
LAHF,SAHF,PUSHF&POPF

- LAHF: Load AH wit6h the low byte of the flag register.

  (AH) ← (lower byte of flag register)

  The contents of the lower byte of flag register is transferred to the higher byte register of the accumulator.

- SAHF: Store (copy) AH register to Low byte of flag register.

  (Lower byte of flag register) ← (AH)

  The content of the higher byte register of the accumulator is moved to lower byte flag register.

Data transfer instructions

Flag Transfer instructions

- PUSHF: Push (copy) the flag register to the top of the stack.

  (sp) ← (sp)-2

  MAs = (ss) X 16_{10} + (sp)

  (MAs; MAs + 1) ← (flags)

  the stack pointer is decremented by two and the contents of the 16-bit flag register is pushed to stack memory to pointed by the SP.

- POPF:

  MAs = (ss) X 16_{10} + (sp)

  (flags) ← (MAs; MAs + 1)

  (sp) ← (sp) + 2

  The contents of (16-bit) stack memory pointed by the SP is moved to flag register and the stack pointer is incremented by 2.

ARITHMETIC INSTRUCTIONS

- The Arithmetic instructions can be classified into following categories
  1. Addition instructions
  2. Subtraction instructions
  3. Multiplication instructions
  4. Division instructions
- The arithmetic operands involve two operands i.e., source and destination.
- The result of the arithmetic operations is stored in destination register or memory location except in the case of comparison. (in comparison the result is used only to update the flags and then it is discarded).
- Performing arithmetic operations directly on two operands in two memory locations is not possible.
- In immediate addressing mode arithmetic operations is the data/operand is not matching with the size of register then the operation will be performed on sign extended data.

ARITHMETIC INSTRUCTIONS

Addition instructions

- The arithmetic instructions alter the flags, the result is used to update the flags.

Addition instruction

ADD,ADC,INC,AAA,DAA

- ADD: Add

  ADD Reg1,Reg2 ➔ Reg1 = Reg1. + Reg2.

  the contents of two registers or a immediate data to register or contents of one memory location to a register contents are added.

  The ADD instruction affects all conditional flags depending on the result of operation.

  Example: ADD AX,BX

ARITHMETIC INSTRUCTIONS

Addition instructions

- ADC: Add with carry

  ADC Reg1. ,Reg2 ➔ Reg1 + Reg2 + [CY]

  the contents of two registers or a immediate data to register or contents of one memory location to a register contents along with carry flag contents are added.

  The ADC instruction affects all conditional flags depending on the result of operation.

  Example: ADC AX,BX

- INC : Increment

  INC <source>

  This instruction increases the contents of the specified register or memory location i.e., source by 1.

  This instruction affects all conditional flags except the carry flag.

  This instruction adds 1 to the contents of the operand.

  Immediate data cannot be operand of this instruction.

  Example: INC BX

  INC [5000H]

ARITHMETIC INSTRUCTIONS

Addition instructions

- AAA : ASCII Adjust after Addition

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

  After the addition the AAA instruction examines the lower 4-bits of AL to check whether it contains a valid BCD between 0 to 9.

  If the contents of AL is between 0 to 9 and AF = 0, AAA sets the 4 higher order bits of AL to 0.The AH must be cleared before addition.

  If the lower digit of AL is between 0 to 9 and AF = 1, 06 is added to AL, The upper 4 bits of AL are cleared and AH is incremented by one

  If the value in the lower nibble is greater than 9 then the AL is incremented by 06,AH is incremented by 1,the AF and CF flags are set to 1, and the higher nibble of AL is cleared to 0.

ARITHMETIC INSTRUCTIONS
Addition instructions

- Example: 1) AL = 67 (before AAA)
  AL = 07(after AAA)
  2) AL = 6A; AH = 00 (before AAA)
  A > 9 ,hence A + 6 = 1010 + 0110
  = 10000B = 10H & AF = 1

Thus before AAA instruction AX = 006AH

After the execution of AAA instruction AX = 0100 and after OR'ing the contents of AX with 3030H then AX = 3130H which is equivalent to ASCII equivalent of BCD number 10.

ARITHMETIC INSTRUCTIONS

Addition instructions

- DAA : Decimal adjust Accumulator

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

  If the lower nibble is greater than 9, after addition or if AF is set, it will add 06H to the lower nibble in AL. After adding 06 in the lower nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL.

  The DAA instruction affects AF,CF,PF and ZF flags. The OF is undefined.

  Example: AL = 53, CL = 29

  ADD AL,CL ; AL <- (AL) + (CL)

  AL = 53 + 29 = 7C H

  After DAA AL <- 7C + 06 H

  AL = 82

ARITHMETIC INSTRUCTIONS

Addition instructions

- Example:

  AL = 73 and CL = 29

  ADD AL,CL

  AL <- AL + CL

  AL <- 73 + 29

  AL <- 9C H

  DAA

  AL <- 02 and CF = 1

  AL → 73

  CL → + 29

  9C

  + 06

  A2

  + 60

  CF = 1 02 in AL

ARITHMETIC INSTRUCTIONS

Subtraction instructions

Subtraction instructions:

SUB, SBB, DEC, NEG, CMP, AAS, DAS

- SUB : subtraction

  SUB X,Y

  X = X – Y, X → destination, Y - source

  This instruction subtracts the source operand from the destination operand and the result is stored in destination.

  the source operand may be a register or a memory location or an immediate data and the destination operand may be a register or a memory location, but source and destination operand both must not be memory operands.The destination operand cannot be an immediate data.

  All the conditional flags are affected by SUB instruction.

  Example: SUB BX,DX

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- SBB : Subtract with Borrow
  SBB X,Y
  X = X – Y – BW(CY) X → destination, Y → Source,
  BW → Borrow i.e., Carry flag contents.

  The subtract with borrow instruction subtracts the source operand and borrow flag(CF) which may reflect the result of the previous calculations , from the destination operand.

  Subtraction with borrow is equivalent to subtracting 1 from the result of SUB (subtraction) operation when the carry flag is set.

  The result is stored in the destination operand.

  All the flags are affected (conditional flags) by this instruction.
  Example: SBB CX,DX

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- DEC : Decrement

  DEC <Destination>

  The decrement instruction subtracts 1 from the contents of the specified register or memory location.

  All the conditional flags except the carry flag are affected depending upon the result

  Immediate data cannot be operand of the instruction.

  Example : DEC AX

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- NEG : Negate

  NEG Mem./Reg.

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

  For obtaining 2’s complement,it subtracts the contents of destination from 0(zero).

  The result is stored back in the destination operand which may be a register or a memory location.

  Using NEG instruction if the OF flag is set, it will indicate that the operation was not successfully completed.

  The NEG instruction affects all conditional flags.

  Example:NEG BX

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- CMP :Compare

  CMP X,Y ; X → Destination, Y → Source

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

  For comparison, it subtracts the source operand from the destination operand but does not store the result any where.

  The flags are affected depending upon the result of the subtraction.

  ZF = 1; when the source and destination operands are equal.

  CF = 1; when source operand is greater than the destination operand.

  CF = 0; when destination operand is greater than source operand

  Example: CMP AX , 1098 H

  CMP AX, BX

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- AAS : ASCII adjust AL after subtraction

  AAS

  AAS instruction corrects the result in AL register after subtracting two Unpacked ASCII operands.

  The result is in unpacked decimal format.If the lower 4-bits of AL register are greater than 9 or if the AF flag is 1,The AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1.

  Otherwise, the CF and AF are set to 0, the result needs no correction.

  As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9.

  the procedure is similar to the AAA instruction except for subtraction of 06 from AL. AH is modified as difference of the previous contents(usually zero) of AH and borrow for adjustment.

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- DAS: Decimal adjust after subtraction

  DAS

The instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number.

The subtraction has to be in AL only.

If the lower nibble of AL is greater than 9, this instruction will subtract06 from the lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60 H from AL.

DAS instruction modifies the AF,CF,SF,PF and ZF flags. The OF flag is undefined after DAS instruction.

DAA and DAS instructions are also called packed BCD arithmetic instructions.

ARITHMETIC INSTRUCTIONS

Subtraction instructions

- Example:
- (1) AL = 75, BL = 46

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

(2) AL = 38 , DL = 61

```
SUB AL , DL     ; AL ← D7 & CF = 1(borrow)
DAS             ; AL ← 77 (as D > 9 , D-6 = 7)
                ; CF = 1 (borrow)
```

ARITHMETIC INSTRUCTIONS

Multiplication instructions

Multiplication instructions:
MUL, IMUL, AAM

- MUL : Unsigned Multiplication of byte or word.

  MUL Reg. / Mem.

  This instruction multiplies an unsigned byte or word by the contents of AL.

  The Unsigned byte or word may be in any one of the general purpose registers or memory locations.

  For Byte multiplication the most significant byte will be stored in AH register and least significant byte is stored in AL register.

  For Word multiplication the most significant word of the result is stored in DX, while the least significant word of the result is stored in AX register

  All the flags are modified depending upon the result of the operation.

  Immediate operand is not allowed in this instruction.

  If the most significant byte or word of result is ‘0’ CF and OF both will be set.

  Example: MUL BL

  MUL BX

ARITHMETIC INSTRUCTIONS

Multiplication instructions

- IMUL : Signed Multiplication.

  This instruction multiplies a signed byte in source operand by a signed byte in AL register or A signed word in AX register.

  The source can be a general purpose register, memory operand,index register or base register, but it cannot be an immediate data.

  While using this instruction the content of accumulator and register should be sign extended binary in 2's complement form and the result is also in sign extended binary.

  In case of 32-bit results, the higher order word(MSW) is stored in DX and lower order word is stored in AX

  In case of 16-bit result it will be stored in AX register.

  The AF, PF, SF, and ZF flags are undefined after IMUL instruction execution.

  If AH and DX contains parts of 16-bit and 32-bit result respectively, CF and OF both will be set.

  The AL and AX are the implicit operands in case of 8 –bits and 16-bits multiplication respectively.

  The unsigned higher bits of the result are filled by sign bit and CF,AF are cleared.

  Example: IMUL BL / IMUL BX

ARITHMETIC INSTRUCTIONS

Multiplication instructions

- AAM : ASCII Adjust after Multiplication

  This instruction after execution, converts the product available in AL into unpacked BCD format.

  The AAM instruction follows a multiplication instruction that multiplies two unpacked BCD operands, i.e., higher nibbles of the multiplication operands should be '0'.The multiplication of such operands is carried out using MUL instruction.

  The result of the multiplication will be available in AX.

  $(AH) = (AL) \div 0A_H$

  $(AL) = (AL) \text{ MOD } 0A_H$

  The AAM instruction replaces the contents of AH by tens of decimal multiplication and AL by singles of the decimal multiplication.

```
MOV AL, 04  ; AL ← 04
MOV BL, 09  ; BL ← 09
MUL BL      ; AH:AL ← 24 H (9 X 4)
AAM         ; AH ← 03
            ; AL ←06
```

ARITHMETIC INSTRUCTIONS

Division instructions

Division instructions: DIV, IDIV, AAD, CBW, CWD

- DIV : Unsigned division

  DIV <reg./Mem>

  This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand.

  The dividend must be in AX for 16-bit operation and the divisor may be specified using any one of the addressing modes except immediate.

  The dividend for 32-bit operation will be in DX:AX register pair (Most significant word in DX and least significant word in AX).

  All the flags are undefined for DIV instruction.

  The result of division is for 16-bit number divided by 8-bit number the Quotient will be in AL register and the remainder will be in AH register similarly for 32-bit number divided by 16-bit number the Quotient will be in AX register and the remainder will be in DX register.

  If the result is too big to fit into AL or AX register then Type-0 (divide by zero) interrupt is generated and the ISR for the Type zero will be executed such that correction steps are taken to accommodate the result.

ARITHMETIC INSTRUCTIONS

Division instructions

- For 16-bit ÷ 8-bit
  (AL) ← (AX) ÷ (reg.-8) ; Quotient
  reg.-8 : 8 – bit register
  (AH) ← (AX) Mod (reg.-8) ; Remainder

- For 32-bit ÷ 16-bit
  (AX) ← (DX)(AX) ÷ (reg.-16) ; Quotient
  (DX) ← (DX)(AX) Mod (reg.-16) ; Remainder
  Reg.-16 : 16 – bit register
  Example: DIV AX/ DIV [BX]

ARITHMETIC INSTRUCTIONS

Division instructions

- IDIV : signed division

  IDIV <reg./Mem>

  This instruction performs signed division. It divides an signed word or double word by a signed 16-bit or 8-bit operand.

  While using IDIV instruction the contents of accumulator and register should be sign extended binary.

  The signed dividend must be in AX for 16-bit operation and the signed divisor may be specified using any one of the addressing modes except immediate.

  The signed dividend for 32-bit operation will be in DX:AX register pair (Most significant word in DX and least significant word in AX).

  All the flags are undefined for IDIV instruction.

  The sign of the quotient depends on the sign of dividend and divisor. The sign of remainder will be same as that of dividend.

  The result of division of signed division is also stored in the same way as the result of unsigned division but the sign of the quotient and remainder depends on the sign of dividend and divisor.

  If the result is too big to fit into AL or AX register then Type-0 (divide by zero) interrupt is generated and the ISR for the Type zero will be executed such that correction steps are taken to accommodate the result.

ARITHMETIC INSTRUCTIONS

Division instructions

- AAD: ASCII adjust before division

  The AAD instruction converts two unpacked BCD digits in AH and AL .

  The ASCII adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte.

  After execution of AAD instruction PF,SF,ZF are modified but AF,CF and OF are not defined.

  The AAD instruction has to be used before DIV instruction is used in the program.

  (AL) ← (AH) X 16_{10} + (AL)

  (AH) ← 00H

  Example: AX = 0508

  After executing AAD instruction AX = 3A ; As 58D = 3AH

  All the ASCII adjust instructions are also called as unpacked BCD arithmetic instructions.

ARITHMETIC INSTRUCTIONS

Division instructions

- CBW : Convert signed byte or word

  This instruction converts a signed byte to a signed word i.e., it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word.

  The byte to be converted will be in AL register and the result will be in AX register

  CBW is used before signed division.

  CBW does not affect any flags.

  Using bit-7 of AL is moved to all the bits of AH register.

  1. If AL = 1xxx xxxx (i.e., $\geq 80_H$)
     Then AH ← 1111 1111(FF_H)
  2. If AL = 0xxx xxxx (i.e., $< 80_H$)
     Then AH ← 0000 0000 (00_H)

ARITHMETIC INSTRUCTIONS

Division instructions

- CWD : Convert signed word to double word

  CWD instruction copies the sign bit of AX to all the bits of DX register

  This operation is to be done before signed division.

  CWD is used for sign extension of 16-bit number into 32-bit number.

  CWD does not affect any flags.

  Bit-15 of AX is moved to all the bits of DX register.

  1. If AX = 1xxx xxxx xxxx xxxx (i.e., $\geq 8000_H$)
     then DX ← 1111 1111 1111 1111 ($FFFF_H$)
  2. If AX = 0xxx xxxx xxxx xxxx (i.e. $< 8000_H$)
     then DX ← 0000 0000 0000 0000 (0000_H)

Bit Manipulation Instructions

- The Logical group includes instructions for performing AND, OR, EX-OR, complement, shift and rotate operations on binary data. The mnemonics used for logical instructions are AND, OR, XOR, TEST etc…….
- The Logical instructions except shift and rotate involve two operands i.e., source and destination operand.
- The source operand can be a register or memory location or immediate data.
- The destination operand can be a register or memory location.
- The result of the logical operation is stored in destination register or memory except in the case of TEST instruction , for which the result is used only to modify the flags and then the result is discarded.
- In double operand logical instructions, both the source and destination cannot refer to memory locations in the same instruction . Thus performing logical operation on two memory bytes or words simultaneously is not possible.
- In double operand logical instructions, the source and destination operand should be of same size, either both the operand size should be byte or word.

Bit Manipulation Instructions

- The Logical instructions alter the flags of 8086.
- The processor uses the result of logical operations to alter the flags which reflect the status of the result.

- The various categories of Bit manipulation instructions are
  1. Logical instructions
  2. Shift instructions
  3. Rotate instructions

- The various logical instructions are
  NOT
  AND
  OR
  XOR
  TEST

Bit Manipulation Instructions
Logical Instructions

- NOT : Complement / Negation (Invert each bit of operand)

  NOT < Destination>

  The NOT instruction inverts each bit (forms the 1's complement) of the byte or word at the specified destination.

  The destination can be a register or a memory location specified by any one of the addressing mode except for immediate addressing mode.

  No flags are affected by the NOT instruction.

  Example: NOT AL

  NOT [AX]

Bit Manipulation Instructions

Logical Instructions

- AND : Logical AND of corresponding bits of two operands

  AND <Destination> , <Source>

  This instruction ANDs each bit in a source byte or word (which might be a register or a memory location or an immediate data) with the same number bit in a destination (which might be a register or a memory location) byte or word.

  The result is stored in destination operand. At least one of the operands should be a register or an memory location , but both the operands cannot be memory locations or immediate operands and also immediate operand cannot be a destination operand.

  The AND operation gives output 1 only when both the inputs are high.

  AND AX , 0008H (let [AX] = 4567H

  4567 = 0100 0101 0110 0111 ; 0008 = 0000 0000 0000 1000

  ∴ 4567 AND 0008 ➔ 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1

  AND ↓↓↓↓ ↓↓↓↓ ↓↓↓↓ ↓↓↓↓

  0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

  ---

  0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 = 0008H

Bit Manipulation Instructions

Logical Instructions

- OR : Logical OR of corresponding bits of two operands

  OR <Destination> , <Source>

  This instruction ORs each bit in a source byte or word (which might be a register or a memory location or an immediate data) with the same number bit in a destination (which might be a register or a memory location) byte or word.

  The result is stored in destination operand. At least one of the operands should be a register or an memory location , but both the operands cannot be memory locations or immediate operands and also immediate operand cannot be a destination operand.

  The OR operation gives output 1 when any one of the inputs are high.

  OR AX , 0008H (let [AX] = 4567H

  4567 = 0100 0101 0110 0111 ; 0008 = 0000 0000 0000 1000

  ∴ 4567 OR 0008 ➔ 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1

  OR ↓↓↓↓ ↓↓↓↓ ↓↓↓↓ ↓↓↓↓

  0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

  ---

  0 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1 = 456F H

Bit Manipulation Instructions

Logical Instructions

- XOR : Logical XOR of corresponding bits of two operands

  XOR <Destination> , <Source>

  This instruction XORs each bit in a source byte or word (which might be a register or a memory location or an immediate data) with the same number bit in a destination (which might be a register or a memory location) byte or word.

  The result is stored in destination operand. At least one of the operands should be a register or an memory location , but both the operands cannot be memory locations or immediate operands and also immediate operand cannot be a destination operand.

  The XOR operation gives output 1 only when both the inputs are dissimilar.

  XOR AX , 0018H (let [AX] = 4567H

  4567 = 0100 0101 0110 0111 ; 0008 = 0000 0000 0000 1000

  ∴ 4567 XOR 0018 ➔ 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1

  XOR ↓↓↓↓ ↓↓↓↓ ↓↓↓↓ ↓↓↓↓

  0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0

  ---

  0 1 0 0 0 1 0 1 0 1 1 1 1 1 1 1 = 457F H

Bit Manipulation Instructions

Logical Instructions

- TEST : Logical Compare instruction(AND operands to update flags)

  TEST <Destination> , <Source>

  This instruction performs a bit by bit logical AND operation on two operands.

  The source and destination operands are not altered they simply update the flags.

  The result of the ANDing operation is not available for further use,but flags are affected.The affected flags are OF, CF,SF,ZF and PF.

  The TEST instruction is often used to set flags before a conditional jump instruction.

  The source operand can be a register or a memory location or immediate data.

  The destination operand can be either a register or a memory location .

  But both source and destination cannot be memory location.

  CF and OF are both 0’s after TEST instruction execution.

  AF will be undefined for TEST instruction.

Bit Manipulation Instructions

Shift Instructions

Shift instructions

SHL / SAL, SHR, SAR

SHL / SAL : Shift Logical / Arithmetic Left

SHL <reg. / Mem>

CF ← R(MSB) ; R(n+1) ← R(n) ; R(LSD) ← 0

These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits.

The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.

The operand to be shifted can be either register or memory location contents but cannot be immediate data.

All the flags are affected depending upon the result.The shift operation will considering using carry flag.

Bit Manipulation Instructions

Shift Instructions

SHR : Shift Logical Right

SHR <reg. / Mem>

CF ← R(LSB) ; R(n) ← R(n+1) ; R(MSD) ← 0

These instructions shift the operand word or byte bit by bit to the right and insert zeros in the newly introduced Most significant bits.

The result of the shift operation will be stored in the register itself.

The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.

The operand to be shifted can be either register or memory location contents but cannot be immediate data.

All the flags are affected depending upon the result.The shift operation will considering using carry flag.

Bit Manipulation Instructions

Shift Instructions

SAR : Shift Logical Right

SAR <reg. / Mem> , <count>

CF ← R(LSB) ; R(n) ← R(n+1) ; R(MSD) ← R(MSD)

These instructions shift the operand word or byte bit by bit to the right.

SAR instruction inserts the most significant bit of the operand in the newly inserted bit positions.

The result will be stored in the register or memory itself.

The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.

The operand to be shifted can be either register or memory location contents but cannot be immediate data.

All the flags are affected depending upon the result.The shift operation will considering using carry flag.

Bit Manipulation Instructions

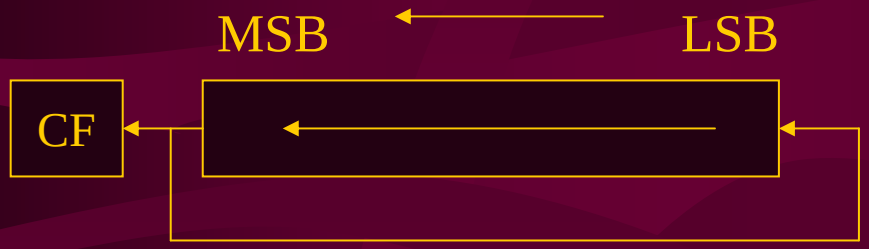
Rotate Instructions

Rotate instructions
ROL, RCL, ROR, RCR

- ROL : Rotate left without carry

  ROL <Reg. / Mem> , <Count>

  R(n+1) ← R(n) ; CF ← R(MSB) ; R(LSB) ← R(MSB)



This instruction rotates all the bits in a specified word or byte to the left by the specified count (bit-wise) excluding carry.

The MSB is pushed into the carry flag as well as into LSB at each operation. The remaining bits are shifted left subsequently by the specified count positions.

The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.

The count will be in instruction if it is 1, and in CL register if greater than 1.

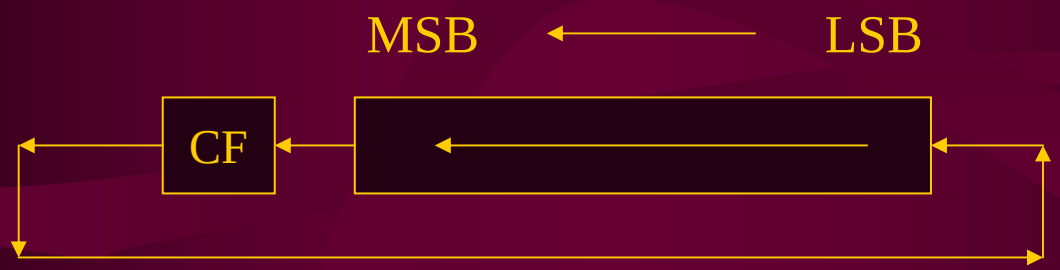
Bit Manipulation Instructions

Rotate Instructions

- RCL : Rotate left through i.e., with carry

  RCL <Reg. / Mem> , <Count>

  R(n+1) ← R(n) ; CF ← R(MSB) ; R(LSB) ← CF



This instruction rotates all the bits in a specified word or byte to the left by the specified count (bit-wise) including carry.

The MSB is pushed into the CF and CF into LSB at each operation. The remaining bits are shifted left subsequently by the specified count positions.

The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.

The count will be in instruction if it is 1, and in CL register if greater than 1.

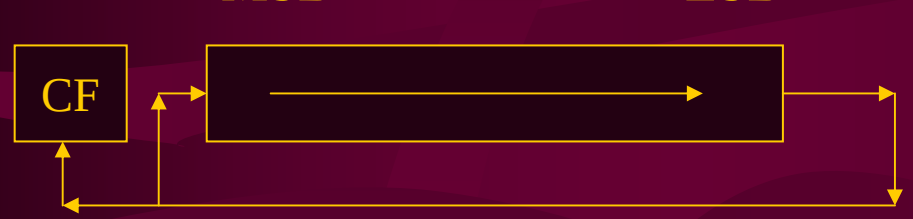
Bit Manipulation Instructions

Rotate Instructions

- ROR : Rotate right without carry

  ROR <Reg. / Mem> , <Count>

  R(n) ← R(n + 1) ; R(MSB) ← R(LSB) ; CF ← R(LSB)



This instruction rotates all the bits in a specified word or byte to the right by the specified count (bit-wise) excluding carry.

The LSB is pushed into the carry flag as well as the MSB at each operation. The remaining bits are shifted right subsequently by the specified count positions.

The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.

The count will be in instruction if it is 1, and in CL register if greater than 1.

Bit Manipulation Instructions

Rotate Instructions

- RCR : Rotate right through i.e., with carry

  RCR <Reg. / Mem> , <Count>

  R(n) ← R(n + 1) ; R(MSB) ← CF ; CF ← R(LSB)

This instruction rotates all the bits in a specified word or byte to the right by the specified count (bit-wise) excluding carry.

The LSB is pushed into the carry flag as well as the MSB at each operation. The remaining bits are shifted right subsequently by the specified count positions.

The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.

The count will be in instruction if it is 1, and in CL register if greater than 1.

String Instructions

- A string is a sequence of bytes or words i.e., a series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually and is known as *byte strings* or *word strings*.
- For referring to a string, two parameters are required,
  - ✓ Starting or end address of the string.
  - ✓ Length of the string.
- The length of the string is usually stored as count in the CX register.
- In case of 8085, the strings are referred by using pointers and counter arrangement which are modified at each iteration, till the required condition for proceeding further is satisfied.
- But in the case of 8086, a set of instructions are used for manipulating the string bytes or words and the index registers are used as pointers for the source and destination strings (SI and DI respectively). The pointers are updated i.e., incrementing and decrementing of the pointers depending on the status of the DF flag.
- If the pointers are byte pointers then they are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter is decremented by one for both byte and word strings.

String Instructions

- The 8086 instruction set includes instructions for string movement, comparison, scanning, loading and storing.
- Generally the string instructions have prefix for repeating the execution of string instructions till the condition is satisfied.
- The string instructions end with “S” or “SB” or “SW” , where “S” represents string, “SB” represents string byte, “SW” represents string word.
- All the string instructions has implied source and destination operand (i.e., the operands are not specified as a part of the instruction).
- The string instructions MOVS and CMPS assume that the source operand is in data segment memory, and the destination is in extra segment memory.
- The string instructions STOS and SCANS assumes that the source operand is in accumulator, and the destination is in extra segment memory.
- The string instruction LODS assumes that the source operand is in data segment memory and the destination is accumulator.
- For string operations, the offset or the effective address of the source operand is stored in SI register and that of destination operand is stored in DI register.
- On execution of string instruction depending on DF, SI and DI registers are automatically updated to point to the next byte / word of the source and destination.If DF = 0 SI and DI are incremented by 1 and if DF = 1 then SI and DI are decremented by 1.

String Instructions

- The string instructions are categorized as
  1. Prefix instructions
  2. String data manipulation instructions

- The Prefix instructions are:
  REP
  REPE / REPZ
  REPNE / REPNZ
- The string data manipulation instructions are :
  MOVS / MOVSB / MOVSW
  CMPS / CMPSB / CMPSW
  SCAS / SCASB / SCASW
  LODS / LODSB / LODSW
  STOS / STOSB / STOSW

String Instructions

Prefix instructions

- REP : Repeat instruction Prefix

  The instruction with REP prefix will be executed repeatedly until the CX register becomes zero(for each iteration CX is automatically decremented by one).

  When CX becomes zero , the execution proceeds to the next instruction in the sequence.
- REPE / REPZ : Repeat when equal or till ZF = 1.

  The instruction with REPE / REPZ prefix will be repeated if CX ≠ 0 and ZF =1 (for each iteration CX is automatically decremented by 1).

  The repeat operation is terminated if CX = 0 or ZF = 0.
- REPNE / REPNZ : Repeat when not equal or till ZF = 0.

  The instruction with REPNE / REPNZ prefix will be repeated if CX ≠ 0 and ZF =0 (for each iteration CX is automatically decremented by 1).

  The repeat operation is terminated if CX = 0 or ZF = 1.

String Instructions

String data byte/word manipulation instructions

- MOVS / MOVSB / MOVSW: Move string byte or word

  One byte or word of a string data stored in data segment is copied into extra segment.

  The SI register points to the source string and DI register points to the destination string.

  The CX register is decremented by one for each byte / word movement.

  The SI and DI registers are automatically incremented or decremented depending on the status of DF.

  $MA = (DS) \times 16_{10} + (SI)$

  $MA_E = (ES) \times 16_{10} + (DI)$

  $(MA_E) \leftarrow (MA)$

  For byte operation

  If DF = 0, then (DI) ← (DI) + 1 ; (SI) ← (SI) + 1

  If DF = 1, then (DI) ← (DI) - 1 ; (SI) ← (SI) – 1

  For word operation

  If DF = 0, then (DI) ← (DI) + 2 ; (SI) ← (SI) + 2

  If DF = 1, then (DI) ← (DI) - 2 ; (SI) ← (SI) - 2

String Instructions

String data byte/word manipulation instructions

- CMPS / CMPSB / CMPSW: Compare string byte or word

  Compare one byte or word of a string data stored in data segment with that stored in extra segment.

  The SI register points to the source string and DI register points to the destination string.

  The CX register is decremented by one for each byte / word movement.

  The SI and DI registers are automatically incremented or decremented depending on the status of DF.

  MA = (DS) X 16_{10} + (SI)

  MA_E = (ES) X 16_{10} + (DI)

  Modify flags ← (MA) - (MA_E)

  If (MA) > (MA_E) then CF = 0 ; ZF = 0 ; SF = 0

  If (MA) < (MA_E) then CF = 1 ; ZF = 0 ; SF = 1

  If (MA) = (MA_E) then CF = 0 ; ZF = 1 ; SF = 0

String Instructions

String data byte/word manipulation instructions

- For byte operation

  If DF = 0, then (DI) ← (DI) + 1 ; (SI) ← (SI) + 1

  If DF = 1, then (DI) ← (DI) - 1 ; (SI) ← (SI) – 1

  For word operation

  If DF = 0, then (DI) ← (DI) + 2 ; (SI) ← (SI) + 2

  If DF = 1, then (DI) ← (DI) - 2 ; (SI) ← (SI) - 2

String Instructions

String data byte/word manipulation instructions

- SCAS / SCASB / SCASW: Scan string byte or String word

  One byte or word of a string data stored in extra segment is subtracted from the contents of AL / AX and the result modifies the flags.

  The DI register points to the string byte or word.

  The CX register is decremented by one for each byte / word movement.

  The DI register is automatically incremented or decremented depending on the status of DF.

  MA = (DS) X 16_{10} + (SI)

  MA_E = (ES) X 16_{10} + (DI)

  Modify flags ← (AL) - (MA_E) / (AX) - (MA_E : MA_E + 1)

  If (AL) > (MA_E) then CF = 0 ; ZF = 0 ; SF = 0

  If (AL) < (MA_E) then CF = 1 ; ZF = 0 ; SF = 1

  If (AL) = (MA_E) then CF = 0 ; ZF = 1 ; SF = 0

  For byte operation

  If DF = 0, then (DI) ← (DI) + 1

  If DF = 1, then (DI) ← (DI) - 1

  For word operation

  If DF = 0, then (DI) ← (DI) + 2

  If DF = 1, then (DI) ← (DI) - 2

String Instructions

String data byte/word manipulation instructions

- LODS / LODSB / LODSW: Load string byte or word into AL register.

  One byte or word of a string data stored in data segment is loaded or stored into AL / AX register.

  The SI register points to the source string .

  The SI register is automatically incremented or decremented depending on the status of DF.

  load instruction does not affect any flags.

  MA = (DS) X 16_{10} + (SI)

  (AL) ← (MA) / (AX) ← (MA : MA + 1)

  For byte operation

  If DF = 0, then (SI) ← (SI) + 1

  If DF = 1, then (SI) ← (SI) – 1

  For word operation

  If DF = 0, then (SI) ← (SI) + 2

  If DF = 1, then (SI) ← (SI) - 2

String Instructions

String data byte/word manipulation instructions

- STOS / STOSB / STOSW: Store string byte or word from AL register.

  One byte or word of a string data stored in AL / AX register is copied or stored as string data into extra segment.

  The DI register points to the destination string.

  The DI register is automatically incremented or decremented depending on the status of DF.

  $MA_E = (ES) \times 16_{10} + (DI)$

  $(MA_E) \leftarrow (AL) / (MA_E : MA_E + 1) \leftarrow (AX)$

  For byte operation

  If DF = 0, then (DI) ← (DI) + 1

  If DF = 1, then (DI) ← (DI) - 1

  For word operation

  If DF = 0, then (DI) ← (DI) + 2

  If DF = 1, then (DI) ← (DI) - 2

Processor Control Instructions

- The Processor control instructions include flag manipulation and processor control instructions.These instructions control the functioning of the available hardware (programmer accessible hardware) inside the processor chip.
- These are categorized into two types:

a) Flag manipulation instructions

b) Machine control instructions

- The flag manipulation instructions directly modify some of the flags of the 8086 flag register.
- The machine control instructions controls the bus usage and execution.
- The processor control group includes instructions to set or clear carry flag, direction flag, and interrupt flag.It also includes the HLT, NOP, LOCK and ESC instructions which controls the processor operation
- The Various Flag manipulation instructions are

  CLC, CMC, STC, CLD, STD, CLI, STI
- The Various machine control instructions are

  WAIT, HLT, NOP, ESC, LOCK

Processor Control Instructions

Flag manipulation instructions

- CLC : Clear Carry

  The carry flag is reset to zero i.e., CF = 0

  CF ← 0
- CMC : Complement the carry

  The carry Flag is Complemented i.e., if CF = 0 before CMC then after CMC CF =1 and vice versa.

  CF ← ~ CF
- STC : Set Carry

  The carry flag is set to one i.e., CF = 1

  CF ← 1
- CLD : Clear direction

  The direction flag is cleared to zero i.e., DF = 0

  DF ← 0
- STD : Set direction

  The direction flag is set to 1 i.e., DF = 1

  DF ← 1

Processor Control Instructions

Flag manipulation instructions

- CLI : Clear Interrupt
  The Interrupt flag is cleared to zero i.e., IF = 0
  IF ← 0

- STI : Set Interrupt
  The Interrupt flag is set to 1 i.e., IF = 1.
  IF ← 1

Processor Control Instructions

Machine control instructions

- WAIT : Wait for Test input pin to go low or an interrupt signal

  This instruction causes the processor to enter into an idle state or wait state and continue to remain in that state until a signal is asserted on the TEST input pin or until a valid interrupt signal is received on the INTR or NMI interrupt input pin.

  If a valid interrupt signal occurs while the 8086 is in idle state, the 8086 will return to the idle state after the interrupt service procedure executes. It returns to the idle state because the address of the WAIT instruction is the address pushed on to the stack when the 8086 responds to the interrupt request.

  WAIT affects no flags

  The WAIT instruction is used to synchronize the 8086 processor with the external hardware such as the 8087 math processor.

- HLT : Halt Processing

  The HLT instruction will cause the 8086 to stop the fetching and execution of the instructions. The 8086 will enter a halt state i.e., used to terminate a program.

  The only ways to get processor out of Halt state are with an interrupt signal on INTR pin, an interrupt signal on NMI pin, or a valid reset signal on RESET input.

Processor Control Instructions

Machine control instructions

- NOP : No Operation

  No operation is performed for three clock periods

  This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction.

  The NOP instruction does not affect any flag.

  The NOP instruction can be used to increase the delay of a delay loop.

  When hand coding, a NOP can also be used to hold a place in a program for instruction that will be added later.

Processor Control Instructions

Machine control instructions

- ESC : Escape

  ESC opcode, Mem. / Reg.

  This instruction is used to pass instructions to a coprocessor , such as the 8087 math coprocessor which shares the address and data bus with 8086

  Instructions for coprocessor are represented by a 6-bit code embedded in the escape instruction.

  As 8086 fetches the instructions bytes, the coprocessor also catches these bytes from the data bus and puts them in its queue , but treats all the normal 8086 instructions as NOPs and when ESC instruction is fetched by 8086, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code in the instruction.

  In most cases 8086 treats the ESC instruction as NOP but in some cases 8086 will access a data item in memory for the coprocessor.

  For ESC opcode, Mem format the data is accessed by 8087 from memory

  For ESC opcode, Mem format the data is accessed by 8087 from 8086 register specified in the instruction.

Processor Control Instructions

Machine control instructions

- LOCK : Assert Bus Lock signal

  The LOCK is used as a prefix to a critical instruction which has to be executed without any disturbances to system bus from other bus masters.

  When LOCK prefix is used in an instruction then during execution of this instruction the lock prefix ensures that the shared system resources are not taken over by other bus masters in the middle of the critical instruction execution.

  When an instruction with LOCK prefix is executed the 8086 will assert its bus lock signal output.This signal is connected to an external bus controller device, which then prevents any other processor from taking over the system bus

  LOCK affects no flags.

Program execution transfer instructions

- The control transfer group consists of call, jump, loop and software interrupt instructions.
- Normally a program is executed sequentially(i.e., the program instructions are executed one after the other), when a branch instruction is encountered the program execution control is transferred to the specified destination or target instructions. The transfer of program execution control is done either by changing the content of IP or by changing the contents of IP and CS.
- When the content of IP alone is modified, the program control branches to new memory location in the same segment.
- When the contents of IP and CS are modified, the program control branches to new memory location in another memory segment.
- The control transfer instructions do not affect the flags of 8086.
- The jump and loop instructions can be classified into conditional and unconditional instructions.
- In conditional instructions, the status of one or more flags are checked and control transfer takes place only if the specified condition is satisfied.

Program execution transfer instructions

- The program execution transfer instructions can be categorized as:
  - Unconditional transfer instructions
  - Conditional transfer instructions
  - Iteration control instructions
  - Software interrupt instructions
- Unconditional transfer instructions:

  CALL

  RET

  JMP
- CALL : Unconditional Call

  The CALL instructions transfer control to a subprogram or subroutine or a procedure after saving return address in the stack memory.

  There are two types of CALL instructions:

  Intra-segment or near call

  Inter-segment or far call

Program execution transfer instructions

Unconditional transfer instructions

- A near call refers to calling a procedure stored in the same code segment memory in which main program(or calling program) resides.
- A far call refers to calling a procedure stored in different code segment memory than that of main program.
- While executing near call, the content of IP alone is pushed to stack. While executing far call the contents of CS and IP are pushed to stack.

Program execution transfer instructions

Unconditional transfer instructions

- Direct near call:

  CALL Disp-16

  This instruction is near-direct call in which the program control is transferred within the same segment.

  The stack pointer is decremented by 2, the IP is pushed into stack and effective address (Disp-16) of the subroutine or procedure to be executed is loaded in IP.

- CALL Disp-16

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (IP)

  (IP) ← Disp-16

Program execution transfer instructions

Unconditional transfer instructions

- In-direct Near CALL

  CALL reg. / Mem.

  This instruction is near- indirect call in which the control transfer is within same segment and the effective address of subroutine / procedure to be called is stored in register or memory.

  The stack pointer is decremented by 2, the IP is pushed into the stack and the effective address of the subroutine / procedure to be executed is loaded in IP from the register / memory.

  CALL reg. / CALL Mem.

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (IP)

  (IP) ← (reg.) / (IP) ← (Mem.)

Program execution transfer instructions

Unconditional transfer instructions

- Direct Far CALL

  CALL $Addr_{offset}$, $Addr_{base}$

  This instruction is far-direct call in which the program control is transferred to another segment.

  The offset and segment base address of the procedure to be executed are stored in memory.

  The stack pointer is decremented by 2, the IP is pushed into the stack. The stack pointer is again decremented by 2 and CS is pushed onto the stack and the base address of the procedure to be executed is loaded into CS.

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (IP)

  (IP) ← $Addr_{offset}$

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (CS)

  (IP) ← $Addr_{base}$

Program execution transfer instructions

Unconditional transfer instructions

- In-direct Far CALL

  CALL Mem.

  This instruction is far-indirect call in which the program control is transferred to another segment.

  The offset and segment base address of the procedure to be executed are directly given in the instruction.

  The stack pointer is decremented by 2, the IP is pushed into the stack. The stack pointer is again decremented by 2 and CS is pushed onto the stack and the base address available in memory is loaded into CS.

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (IP)

  (IP) ← $(Mem.)_{(offset\ address)}$

  (SP) ← (SP) – 2

  MA_S = (SS) X 16_{10} + (SP)

  (MA_S) ← (CS)

  (IP) ← $(Mem.)_{(base\ address)}$

Program execution transfer instructions

Unconditional transfer instructions

- Every procedure or subroutine end with RET instruction.
- The execution of RET instruction at the end of subroutine or procedure, will pop the contents of top of the stack to IP in case of near call or to IP and CS in case of far call. Thus the program control return back to main program.
- Return instruction does not affect any flags.
- Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return inter-segment
4. Return inter-segment adding 16-bit immediate to the SP contents

Program execution transfer instructions

Unconditional transfer instructions

- Return from call within segment

  RET

  Return the control back to calling procedure from the called procedure with in the segment.

  The content of top of stack is transferred to IP and the stack pointer is incremented by two.

  $MA_S = (SS) \times 16_{10} + (SP)$

  $(IP) \leftarrow (MA_S)$

  $(SP) \leftarrow (SP) + 2$

Program execution transfer instructions

Unconditional transfer instructions

- Return from call within segment adding immediate value to SP

  RET data-16

  Return the control back to calling procedure from the called procedure within the segment. The content of top of the stack is transferred to IP and the SP is incremented by a value (data-16) specified in the instruction.

  $MA_S = (SS) \times 16_{10} + (SP)$

  $(IP) \leftarrow (MA_S)$

  $(SP) \leftarrow (SP) + data\text{-}16$

Program execution transfer instructions

Unconditional transfer instructions

- Return from inter-segment call

  RET

  Return the control back to calling procedure from the called procedure which is in different segment. The content of top of the stack is transferred to IP and the SP is incremented by 2. Next the content of the current top of the stack is moved to CS and SP is incremented by 2.

  $MA_S = (SS) \times 16_{10} + (SP)$

  $(IP) \leftarrow (MA_S)$

  $(SP) \leftarrow (SP) + 2$

  $MA_S = (SS) \times 16_{10} + (SP)$

  $(CS) \leftarrow (MA_S)$

  $(SP) \leftarrow (SP) + 2$

Program execution transfer instructions

Unconditional transfer instructions

- Return from inter-segment call adding immediate data to SP.

  RET data-16

  Return the control back to calling procedure from the called procedure which is in different segment. The content of top of the stack is transferred to IP and the SP is incremented by 2. Next the content of the current top of the stack is moved to CS and SP is incremented by a value (data-16) specified in the instruction.

  MA_S = (SS) X 16_{10} + (SP)

  (IP) ← (MA_S)

  (SP) ← (SP) + 2

  MA_S = (SS) X 16_{10} + (SP)

  (CS) ← (MA_S)

  (SP) ← (SP) + data-16

Program execution transfer instructions

Unconditional transfer instructions

- JMP : Unconditional Jump

  The unconditional jump instructions does not check for any flag condition. When the unconditional jump instruction is executed the program control is transferred to new memory location either in same segment or in another segment.

  In near jump instruction the program control is transferred to new memory location in the same segment by modifying the content of instruction pointer (IP).

  In far jump instruction the program control is transferred to new memory location in another segment by modifying the content of instruction pointer (IP) and code segment (CS) register.

- JMP Disp-16 (near jump instruction)

  The 16-bit value (Disp-16)given in the instruction is added to instruction pointer (IP).

  (IP) ← (IP) + Disp-16

Program execution transfer instructions

Unconditional transfer instructions

- JMP Disp-8 (near jump instruction)

  The 8-bit value (Disp-8) given in the instruction is sign extended to 16-bit and added to instruction pointer(IP).

  Disp-16 ← (sign extended) Disp-8

  (IP) ← (IP) + Disp-16

- JMP reg. / Mem. (near jump instruction)

  JMP Reg.

  JMP Mem.

  The 16-bit value stored in the register or memory is added to instruction pointer (IP).

  (IP) ← (IP) + (Reg.)

  (IP) ← (IP) + (Mem)

Program execution transfer instructions

Unconditional transfer instructions

- JMP $Addr_{offset}$, $Addr_{base}$ (Far jump instruction)

  The offset address given in the instruction is loaded in IP and the base address given in the instruction is loaded in CS register.

  (IP) ← $Addr_{offset}$

  (CS) ← $Addr_{base}$

- JMP Mem.

  The content of (16-bit) memory is moved into IP and the next word in memory is moved into CS register.

  (IP) ← (Mem.)

  (CS) ← (Mem. + 2)

Program execution transfer instructions

Iteration control instructions

- The iteration instructions are also known as Loop instructions.
- Loop instructions are used to execute a group of instructions, a number of times as specified by a count value stored in CX register.
- The number of instructions to be looped will be specified directly in the instruction as a signed eight bit number (Displacement 0r Disp-8).
- For positive displacement the instructions below the LOOP instruction are executed and for negative displacement the instructions above the LOOP instruction are executed.
- The contents of CX register is decremented by one after each execution of looped instructions. The effective address of first instruction of the loop is obtained by sign extending the Disp-8 to 16-bit and adding to IP.

Program execution transfer instructions

Iteration control instructions

- LOOP Disp-8

  LOOP <Disp-8>

  Repeat execution of the group of instructions until the content of CX is zero. After each execution CX is decremented by one.

  Loop if (CX) ≠ 0

  (CX) ← (CX) – 1

- Example:

```
        MOV BX, OFFSET List ; BX loaded with the address of 1st
                            ;element in array List array .
        MOV CX, 20 ; load the no. of elements in array List into CX Reg.
X1 :    MOV AL, [BX] ; get the element pointed by BX into AL Reg.
        ADD AL, 05H ; Add 5H to element
        MOV [BX], AL ; Store the result back into array List
        INC BX ; Increment pointer
        LOOP X1 ; Repeat the process till count is zero
        LOOP X1 similar to DEC CX ; JNZ X1
```

Program execution transfer instructions

Iteration control instructions

- LOOPZ / LOOPE <Disp-8>

  Repeat execution of the group of instructions, if the content of CX is not zero and ZF = 1. After each execution CX is decremented by one.

  Loop if (CX) ≠ 0 and ZF =1

  (CX) ← (CX) – 1

  Example:

```
        MOV BX, OFFSET List ; Get offset of 1st element of List array
                            ; into BX.
        MOV CX, 10 ; Count in CX
        DEC BX
X1 :    INC BX
        CMP [BX] , 45H   ; compare all array elements with 45H
        LOOPE X1
```

  If (CX) = 0 and ZF = 1 on exit then all elements of array are 45H;

  If (CX) ≠ 0 and ZF = 0 BX pointing to first element ≠ 45H in array ;

  If (CX) = 0 and ZF =0 then last element of the array is 45 H.

Program execution transfer instructions

Iteration control instructions

- LOOPNZ / LOOPNE <Disp-8>

  Repeat execution of the group of instructions, if the content of CX is not zero and ZF = 0. After each execution CX is decremented by one.

  Loop if (CX) ≠ 0 and ZF =0

  (CX) ← (CX) – 1

  Example:

```
        MOV BX, OFFSET List
        MOV CX, 20
        DEC BX
X1 :    INC BX
        CMP [BX], 40H      ; Compare the array elements with 40H
        LOOPNE X1
```

  If (CX) = 0 and ZF =0 then 40H was not found in array ; If (CX) ≠ 0 and ZF = 1 then BX is pointing to the element which is 40H ; If (CX) = 0 and ZF = 1 then the last element in the array was 40H.

Program execution transfer instructions

Iteration control instructions

- JCXZ : Jump if the CX register is zero

  This instruction will cause a jump to a label given in the instruction if the CX register contains all 0's.

  If CX does not contain all zeros , execution will simply proceed to the next instruction. This instruction does not refer to the zero flag when it decides whether to jump or not.

  The destination label for this instruction must be in the range of –128 to +127 bytes from the address of the instruction after the JCXZ instruction.

  JCXZ affects no flags.

  Example:

```
        JCXZ X1 ; If CX is all zeros then go to label X1
X2 : ADD [BX], 05H
        INC BX
        LOOP X2
X1 : MOV [BX], 00H
```

Program execution transfer instructions

Software Interrupt Instructions

- The Software Interrupt instructions are the instructions used for executing interrupts through instructions.
- The INT instructions are called Software Interrupts.
- The INT instructions are used to call a procedure or subroutine on Interrupt basis.
- The procedure executed on Interrupt basis is called Interrupt Service Routine (ISR).
- The INT instruction is accompanied by a type number, which can be in the range of 0 to 255. Thus 8086 processor has 256 types of software interrupts that can be implemented.
- The software interrupts are used to implement the system call service of the operating system.
- In order to execute an ISR, a 16-bit effective address for IP and a 16-bit base address for CS are needed. Thus for each INT instruction four memory locations are reserved in the first 1K address space of memory.

Program execution transfer instructions

Software Interrupt Instructions

- In the reserved locations, the 1st two locations are used to store the effective address(to be loaded into IP) and the next two locations are used to store the base address(to be loaded into CS register).
- The address of the reserved memory location is called vector address.
- The vector address of an interrupt is obtained by multiplying the type number by 4.
- Before executing ISR, the contents of IP, CS and Flag register are pushed to stack. Each ISR is terminated by IRET (Interrupt return) instruction.
- On executing IRET instruction the top of stack are Popped to IP, CS and Flag register.
- Thus the program control returns back to main program after executing ISR.
- INT → Interrupt program execution

  INT 3 → Break-point Interrupt

  INTO → Interrupt on overflow

  IRET → Interrupt return

Program execution transfer instructions

Software Interrupt Instructions

- INT <type>
  - This instruction is a software interrupt and used to call a service procedure(or subroutine) on interrupt basis.
  - The type number is from 0 to 255.On execution of this instruction the contents of flag register, CS register and IP are pushed to stack, one by one after decrementing SP by 2 before each push operation.
  - The flags IF and TF are also cleared.
  - The effective vector address is calculated by multiplying the type number by 4.
  - The memory location pointed by vector address contain the address of interrupt service routine.
  - The 1st word pointed by the calculated vector address is moved to IP and the next word is moved to IP and the next word is moved to CS register.

Program execution transfer instructions

Software Interrupt Instructions

- (SP) ← (SP) – 2
  (MA_S) ← (flags)
  (SP) ← (SP) –2
  (MA_S) ← (CS)
  (SP) ← (SP) –2
  (MA_S) ← (IP)
  (IP) ← (0000 : (Type X 4))
  (CS) ← (0000 : (Type X 4) + 2)
  IF ← 0; TF ← 0

  For each push operation stack memory is given by

  $MA_S = (SS) \times 16_{10} + (SP)$

Program execution transfer instructions

Software Interrupt Instructions

- The sequence operations for executing INT instruction by 8086:
  1. Decrement the stack pointer by 2 and push the flags on to the stack.
  2. Decrement the stack pointer by 2 and push the contents of CS on to the stack.
  3. Decrement the stack pointer by 2 and push the offset of the next instruction after the INT number instruction on to the stack.
  4. Get a new value for IP from an absolute memory address of 4-times the type specified in the instruction.EX : For INT 8 the new IP value will be read from address 00020H
  5. Get a new value for CS from an absolute memory address of 4-times the type specified in the instruction + 2; FOR INT 8 instruction the new CS value will be read from 00022H location
  6. Reset both IF and TF flags but other flags are not affected by INT instruction

Program execution transfer instructions

Software Interrupt Instructions

- INT 3H
  - This instruction is a special type of software interrupt which has the single byte code of CC_H.
  - Many systems use this as a break point instruction.
  - The operations performed by this instruction is same as that of type-3 interrupt.
- (SP) ← (SP) – 2
  (MA_S) ← (flags)
  (SP) ← (SP) –2
  (MA_S) ← (CS)
  (SP) ← (SP) –2
  (MA_S) ← (IP)
  (IP) ← $(0000C_H)$; 3 X 4 = 12 i.e. 0CH
  (CS) ← $(0000E_H)$; 12 + 2 = 14 i.e. 0EH
  IF ← 0; TF ← 0
  MA_S = (SS) X 16_{10} + (SP) for each PUSH operation.

Program execution transfer instructions

Software Interrupt Instructions

- INTO

  If overflow flag (OF) is 1, then type-4 interrupt is performed.

  If OF = 1, then following operation are performed.

  (SP) ← (SP) – 2

  (MA_S) ← (flags)

  (SP) ← (SP) –2

  (MA_S) ← (CS)

  (SP) ← (SP) –2

  (MA_S) ← (IP)

  (IP) ← (00010_H) ; 4 X 4 = 16 i.e. 10H

  (CS) ← (00012_H) ; 16 + 2 = 18 i.e. 12H

  IF ← 0; TF ← 0

  MA_S = (SS) X 16_{10} + (SP) for each PUSH operation.

Program execution transfer instructions

Software Interrupt Instructions

- IRET

This instruction is used to terminate an interrupt service procedure and transfer the program control back to main program.

On execution of this instruction the contents of top of the stack (pointed by SP) are moved (popped) to IP , CS and Flag registers one by one

After each pop operation the SP is incremented by 2.

The RET instruction should not be used to return from procedures because the RET instruction does not copy the flags from the stack back to the flag register.

$(IP) \leftarrow (MA_S)$; $(SP) \leftarrow (SP) + 2$

$(CS) \leftarrow (MA_S)$; $(SP) \leftarrow (SP) + 2$

$(Flags) \leftarrow (MA_S)$; $(SP) \leftarrow (SP) + 2$

For each pop operation the stack memory address is calculated as

$MA_S = (SS) \text{ X } 16_{10} + (SP)$

Program execution transfer instructions

Conditional Jump Instructions

- In a conditional jump instruction one or more flag conditions are checked. If conditions are true, then the program control is transferred to new memory location in the same segment by modifying the contents of IP.
- All conditional instructions are only near jump (or short jump), hence the contents of CS is not altered.
- In all conditional jump instructions an 8-bit value (Disp-8), will be directly specified in the instruction which is sign extended to 16-bit and added to IP. The new value of IP is the effective address of the instruction where the program control is transferred, if the condition is true.
- The conditional jump instructions are often used after a compare instruction.
- When the conditional jump instructions are executed execution control is transferred to the address specified relatively in the instruction, provide the condition implicit in the opcode is satisfied, when the condition is not satisfied the execution continues sequentially and the conditions here indicate the status of the conditional flags in the flag register.

Program execution transfer instructions

Conditional Jump Instructions

- Conditional jump instructions does not affect any flags.
- As the address has to be specified in the instruction relatively in terms of displacement which must lie within –80H to 7FH (i.e. –128 to 127) bytes from the address of the branch instruction.
- A label may represent the displacement, if it lies within the above specified range.
- As the conditional jump instructions are generally used with congestion with comparison of numbers the comparison can be between unsigned and as well as between signed numbers. Thus the term *below* and *above* refer to *unsigned binary* numbers. The terms *greater than* and *lesser than* refer to *signed binary* numbers.
- The term *above* means *larger in magnitude* and *below* means *smaller in magnitude.*
- The term *greater than* means *more positive* and the *lesser than* means *more negative*.

Program execution transfer instructions

Conditional Jump Instructions

S.No.	Mnemonic	Condition	Operation by instruction
1.	JA / JNBE	CF = 0 AND ZF =0	Jump if above / Jump if not below or equal.
2.	JAE / JNB	CF =0	Jump if above or equal / Jump if not below
3.	JB /JNAE	CF = 1	Jump if below / Jump if not above or equal
4.	JC	CF = 1	Jump if carry flag (CF) = 1.
5.	JNC	CF = 0	Jump if no carry i.e., CF = 0
6.	JE /JZ	ZF =1	Jump if equal / Jump if zero (ZF = 1)
7.	JNE / JNZ	ZF = 0	Jump if not equal/ Jump if Non zero (ZF = 0)

Program execution transfer instructions

Conditional Jump Instructions

S.No.	Mnemonic	Condition	Operation by instruction
8.	JO	OF = 1	Jump if Over flow flag OF = 1
9.	JNO	OF = 0	Jump if no over-flow (OF = 0)
10.	JP / JPE	PF = 1	Jump if parity / Jump if even parity
11.	JNP / JPO	PF = 0	Jump if no parity / Jump if Odd parity
12.	JS	SF = 1	Jump if sign
13.	JNS	SF = 0	Jump if no sign

Program execution transfer instructions

Conditional Jump Instructions

S.No	Mnemonic	Condition	Operation by instruction
14.	JG / JNLE	ZF = 0 (AND) CF = OF	Jump if greater / Jump if not less than or equal
15.	JGE / JNL	SF = OF	Jump if greater than or equal / Jump if not less than
16.	JL / JNGE	SF ≠ OF	Jump if less than / Jump if not greater than or equal
17.	JLE / JNG	SF ≠ OF (OR) ZF =1	Jump if less than or equal / Jump if not greater than

Program execution transfer instructions

Conditional Jump Instructions

- JG / JNLE :

  Greater means more positive and not less than or equal means not more negative and also not equal.

  Example:

  00000111 is greater than 11101010. As in signed notation 00000111 is more positive than 11101010 as the second number has MSB = 1 thus negative number.

- JL / JNGE :

  Lesser than means more negative and Not greater than or equal means more negative and also not equal.