Unit-4

8086 microprocessors

What is microprocessor ?

- A **microprocessor** is a computer processor that incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC).
- The microprocessor is a multipurpose, programmabledevice that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output

Generation of Microprocessor

- **1st Generation:** This was the period during 1971 to 1973 of microprocessor's history. In 1971, INTEL created the first microprocessor 4004 that would run at a clock speed of 108 KHz.

Cont…

- **2nd Generation:** This was the period during 1973 to 1978 in which very efficient 8-bit microprocessors -INTEL-8085.
- **3rd Generation:** From 1979 to 1980, INTEL 8086/80186/80286 were developed. It is 16-bit processor.Speeds of those processors were four times better than the 2nd generation processors.

Cont….

- **4th** Generation:After 1980, INTEL 80386 & 80486 were developed. It is 32-bit processor.

Terms related to Microprocessors

- Bit

  A digit of the binary number or code
- Nibble

  The 4-bit binary number or code
- Byte

  The 8-bit binary number or code
- Word

  The 16-bit binary number or code
- Double Word

  The 32-bit binary number or code

Terms related to Microprocessors

- **Data**

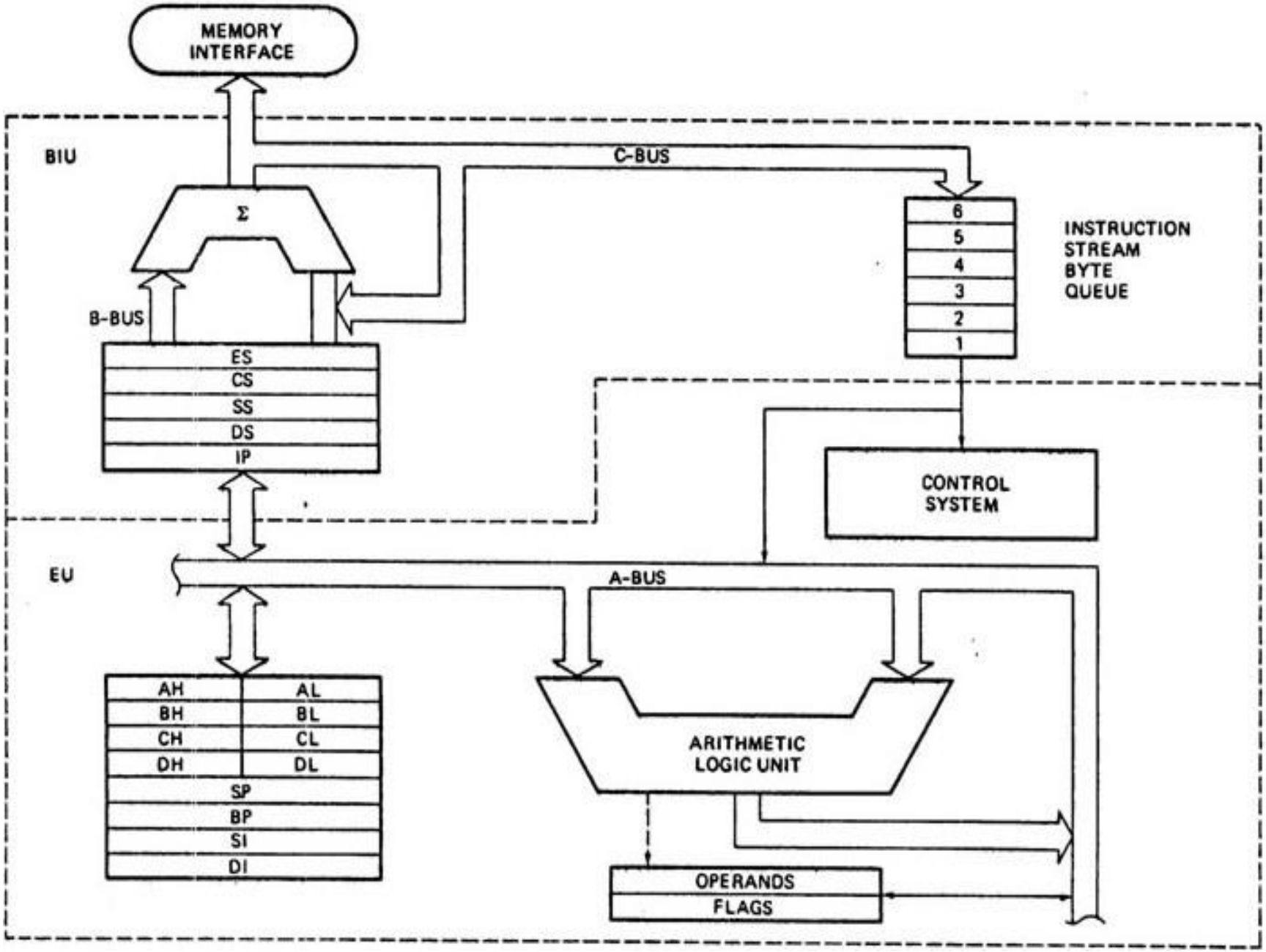
  The quantity operated by an instruction of a program is called data. The size of the data is specified as Bit, Byte,Word……..
- **Address**

  The address is an identification number in binary for Memory locations. The 8086 processor uses 20-bit address for memory.
- **Bus**

  A bus is a group of conducting lines that carries data,address and control signals.

Features of 8086

- Introduced in 1978 .
- Comes in Dual-In-Line Package(DIP) IC.
- 8086 1s a 16-bit microprocessor .
- Works on 5 volts power supply.
- It is built on single semiconductor chip and packaged in an 40-pin IC.
- It has 20-bit address bus and 16-bit data bus.
- It can directly address upto 2^{20} I.e., 1M bytes of memory.
- The maximum internal clock for 8086 is 5MHz


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

8086 Architecture

- The architecture of 8086 is divided into two functional parts i.e.,

i. Execution unit (EU)

ii. Bus interface unit (BIU)

- BIU and EU operate parallelly and independently i.e., EU executes the instructions and BIU fetches another instruction from the memory simultaneously.
- As the whole architecture is divided into two independent functional parts and both the subsystem's operations can be overlapped, hence the architecture is PIPELINING type of architecture.

8086 Architecture

EXECUTION UNIT

- The execution unit consists of the following:
  - General purpose registers(AX,BX,CX,DX)
  - Pointer & Index registers(SP,BP,SI,DI)
  - ALU
  - Flag register(FLAGS/ PSW)
  - Instruction decoder
  - Timing and control unit

8086 Architecture

Functions of EU

- Receives opcode of an instruction from the queue.
- decodes the instructions.
- Executes the instruction.

Functions of various parts of EU

- Control circuitry: Directs internal operations.
- Instruction Decoder: Translates instructions fetched from memory into series of actions.
- ALU: Performs arithmetic and logical operations.
- FLAGS: Reflects the status of program.
- General purpose registers: Used to store Temporary data.
- Index and Pointer registers: Specifies/ informs about offset of operand

8086 Architecture

BUS INTERFACE UNIT

- The BIU consists of the following:
  - Segment Registers(CS,DS,ES,SS)
  - Instruction pointer
  - 6-Byte instruction Queue Register

8086 Architecture

Functions of BIU

- Handles transfer of data and address between processor and memory / I/O devices.
- Compute physical address and send it to memory interfaces.
- Fetches instruction codes and stores it in Queue.

8086 Architecture

Functions of various parts of BIU

- Segment registers : Used to hold the starting address of the segment .
- Queue register: Used to store prefetched instructions and inputs it to EU. which is an 6-byte FIFO register set. When the EU is ready for its next instruction,it simply reads the instruction byte(s) for the instruction from the queue which is present in BIU.
- Instruction Pointer: Used to point to the next instruction to be executed by EU.

Register organization of 8086

- The various registers available internal to 8086 microprocessors are :
  1. Flag Register
  2. General purpose registers
     AX (AH,AL)
     BX (BH,BL)
     CX (CH,CL)
     DX (DH,DL)
  3. Pointer and Index registers (IP,SP,BP & SI,DI)
  4. Segment registers (ES, CS, DS,SS)

Register organization of 8086

Flag Register

- Flag register is part of EU.
- 8086 microprocessor has a 16-bit flag register.
- The flag register contents indicate the result of compuitation in the ALU.It is also known as PSW (Program Status Word).
- The flag register/psw can be divided into 2-parts:

  Conditional /status flags

  Machine Control flags
- 8086 microprocessor has 9- active flags

  6- conditional flags

  3- control flags
- Conditional flags: The lower byte of the flag register along with overflow flag, they reflect the status of program.

Register organization of 8086

- Control Flags : Higher byte of the flag register , It has 3-flags i.e., direction flag, interrupt flag and trap flag.

They control the working of machine(microprocessor)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	X	X	X	X	O	D	I	T	S	Z	X	AC	X	P	X	C

O → Overflow flag, D → Direction flag, I → Interrupt flag,

T → Trap flag, S → Sign flag, Z →Zero flag,

AC → Auxiliary Carry flag, P → Parity flag

CY → Carry flag, X → Not used / Undefined

SET → 1 & RESET → 0.

Register organization of 8086

Flag Register

Condition flags

- Bit – 0 : CF (carry flag) — addition sets flag if carry out of MSB generate; subtraction sets flag if borrow needed.
- Bit –2 : PF (parity flag) — set to 1 if low-order 8 bits of result contain even number of 1's.
- Bit – 4 : AF (auxiliary carry flag) — set if carry out of bit 3 during addition or borrow by bit 3 during subtraction.

Register organization of 8086

Flag Register

Condition flags

- Bit – 6 : ZF (zero flag) — set to 1 if result is 0; to 0 if result is nonzero
- Bit – 7 : SF (sign flag) —set to 1 if equal to MSB of result is 1.Thus this flag indicates whether the result is positive or negative.
- Bit – 11 : OF (overflow flag) — set if overflow occurs (that is, the result can not be included in the available capacity)

Register organization of 8086
Example – 1

- CF (carry flag) — carry out of MSB
- PF (parity flag) — set to 1 if low-order 8 bits (low order byte) contain even number of 1's
- AF (auxiliary carry flag) —carry out of bit 3
- ZF (zero flag) — set to 1 if result is 0; to 0 if result is nonzero
- SF (sign flag) —MSB of result
- OF (overflow flag) — set if carry in to MSB is not equal to carry out from MSB)

```
  0011 0100 1101
  1100
+0000 0111 0010
------------------
  1110
------------------
  0011 1100 0000
  1010

CF = 0

PF = 1

AF = 1

ZF = 0

SF = 0

OF = 0
```

Register organization of 8086
Example – 2

- CF (carry flag) — carry out of MSB
- PF (parity flag) — set to 1 if low-order 8 bits (low order byte) contain even number of 1's
- AF (auxiliary carry flag) —carry out of bit 3
- ZF (zero flag) — set to 1 if result is 0; to 0 if result is nonzero
- SF (sign flag) —MSB of result
- OF (overflow flag) — set if carry in to MSB is not equal to carry out from MSB)

```
    1111 1111
1110 0101
  +1111 1111 1011
0001
 1 1111 1111 1001
0110
```

CF = 1

PF = 1

AF = 0

ZF = 0

SF = 1

OF = 0

Register organization of 8086

Flag Register

Control flags

- TF (trap flag) — if set, a trap is executed after each instruction (single step execution).
- IF (interrupt enable flag) — if set, a maskable interrupt can be recognized by the CPU; otherwise, these interrupts are ignored.
- DF (direction flag) — used by string manipulation instructions; if clear to 0, then process string from low address to high; if set to 1, then process string from high address to low.

Register organization of 8086

General Purpose Register

- In 8086 there are 4- general purpose registers i.e., AX,BX,CX,DX.
- These registers are of 16-bit size and can be used either as a whole 16-bit register (the letter X used in the representation of the register indicates that the complete 16 – bit register is being used) or the upper and lower bytes can be accessed separately (the letters H and L indicates the higher order and lower order bytes respectively in the representation of the registers)
- The general purpose registers can be used to store both operands and temporary results and each of them can be accessed as whole or as sub-registers.

Register organization of 8086

General Purpose Register

- In addition to serving as general purpose registers AX,BX,CX,DX have special uses as addressing, counting, and I/O roles.
- The special uses of the general purpose registers is:

  AX → used as accumulator

  BX → used as a base register in address calculation for some of the instructions

  CX → used as an counter by certain instructions (ex : Loop)

  DX → used as a destination register in case of multiplication and division instructions

Register organization of 8086

Segment registers

- The list of various segment registers is as follows:

  CS → Code Segment Register

  DS → Data Segment Register

  ES → Extra Segment Register

  SS → Stack Segment Register

- The 8086 microprocessor contains 20-bit address bus,so that there are 2^{20} (1Mbyte) memory locations.
- This memory is divided into logical segments.each segment thus contains 64Kbytes of memory.

Register organization of 8086

- **Code segment register** :It is used to address the code segment of memory, where the executable program is stored.
- **Data segment register:** is points to data segment of memory, where the data is resided.
- **Extra segment register** :It is another data segment of memory. Thus extra segment also contains data.
- **Stack segment register:**It is used to address the stack segment of memory .which is used to store stack data.

Register organization of 8086

Pointer and index registers

- The 8086 contains 3 pointer registers(IP,SP,BP) and 2 index registers(SI,DI).All these registers are 16-bit registers.
- The pointer registers contain the offsets with in the particular segments as follows:

  IP → Offset within the Code segment.

  BP → Offset within the Data segment.

  SP → Offset within the Stack segment.
- The SI register is used to store the offset of sourse data in data segment, while DI register is used to store the offset of destination in data in data or extra segment.
- The index registers are particularly useful for string manipulations.

Generation of 20-bit(physical) address

- To access any memory location from any segment we need 20-bit physical address.
- The 8086 generates this address using contents of segment registers & offset registers.
- For this, the contents of segment register also called as segment address is shifted left bit-wise by four times & to this result,contents of offset register also called as offset adderss is added.
- The Bus Interface Unit(BIU) has a separate adder to perform this procedure for obtaining physical address.

Generation of 20-bit(physical) address

Example:

Segment address- 1005H
Offset address - 5555H
Segment address-1005H- 0001 0000 0000 0101
Shifted by 4-bit positions-0001 0000 0000 0101 0000
+
Offset address - 0101 0101 0101 0101
Physical address -0001 0101 0101 1010 0101
1 5 5 A 5

Addressing Modes Of 8086

- Addressing modes:

  addressing modes indicates a way of locating data or operands.

- Addressing modes of the instructions depending upon their types.
- According to the flow of instruction execution, the instructions are classified as

  (i) sequential control flow instructions

  (ii) control transfer insructions

Addressing modes for sequential control flow instructions:

1. Immediate Addressing Mode
2. Register Addressing Mode
3. Direct Addressing Mode
4. Register Indirect Addressing Mode
5. Indexed Addressing Mode
6. Based Indexed Addressing Mode
7. Register Relative Addressing Mode
8. Relative Based Indexed Addressing Modes

Cont……..

- Immediate Addressing Mode:

  The data is part of instruction itself and it is available in successive bytes . The immediate data may be 8 or 16 bit.

  Example: MOV AX,0005H

- Register Addressing Modes:

  In this mode data is stored in a register and it is referred using particular register.

  Example: MOV AX,BX

Cont…..

- Direct Addressing Mode:

  In this addressing mode,address of memory is specified in the instruction.

  Example: MOV AX, [5000]

  physical address=10H*DS+5000H

- Register Indirect Addressing Mode:

  In this addressing mode, address is specified indirectly by using offset registers.The offset address may be in either BX or SI or DI registers. The default segment is either DS or ES.

  Example: MOV AX,[BX]

  physical address=10H*DS+[BX]

Cont….

- Indexed Addressing Mode:

  In this mode , offset of the operand is stored in one of the segment registers.DS is the default segment for index registers SI and DI.

Example: MOV AX,[SI]

Physical address=10H*DS+[SI]

- Based Indexed Addressing Mode:

  This is same as register indirect, but the physical address of is formed by adding contents of base register to the index registers. DS and ES are default segments, SI and DI are used as index registers and BX and BP are used as base registers.

Example: MOV AX, [BX][SI]

Physical address=10H*DS+[BX]+[SI]

Cont……

- Register Relative Addressing Mode:

  The physical address is formed by adding 8-bit or 16-bit displacement with the contents of any one of the registers BX,BP,SI,DI in the default segments.

  Example: MOV AX,50H[BX]

  physical address=10H*DS+50H+[BX]

- Relative Based Indexed Addressing Mode:

  The physical address is formed by adding 8-bit or 16-bit displacement with the contents of any one of the base registers(i.e. BX/BP) and index register (i.e. SI/DI) in the default segments.

  Example: MOV AX, 50[BX][SI]

  physical address=10H*DS+50H+[BX]+[SI]

Addressing modes for control transfer instructions

- For the control transfer instructions, the addressing modes depend upon whether the address location is within the same segment or in different segment.
- The various control transfer addressing modes are:
  1. Intra-segment Direct mode
  2. Intra-segment Indirect mode
  3. Inter-segment Direct mode
  4. Inter-segment Indirect mode

- If the address location to which the control is to be transferred lies in the same segment is called Intra-segment mode and if it is different segment then it is known as Inter-segment mode.

Cont……..

- **Intra-Segment Direct Addressing Mode:**

  The address to which the control is to be transferred lies in the same segment and appears directly in the instruction as an immediate displacement value .

  Example : JMP SHORT *Label*

Cont….

- Intra-Segment Indirect Addressing Mode:

  The address to which the control is to be transferred is in the same segment but it is passed to the instruction indirectly. The address is found as the content of a register or a memory location .

  Example: JMP [BX]

Cont…

- Inter-segment Direct Addressing Mode;

  In this mode, the address to which the control is to be transferred is in a different segment.Here the CS and IP of the destination address are specified directly in the instruction

  Example: JMP 5000H:2000H ;jump to address 2000h in segment 5000h

- Inter-segment Indirect Addressing Mode:

  In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly,i.e.,contents of a memory block containing four bytes, i.e.,IP(LSB),IP(MSB),CS(LSB), CS(MSB) sequentially.

  Example: JMP [2000H] ;jump to address 2000h that points to memory block.

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 (or) logical instructions
4. String instructions
5. Control transfer (or) branch instructions
6. Processor control instructions

Data transfer instructions

- General purpose byte or word transfer instructions:
  - **1) MOV**
  - **2) PUSH**
  - **3) POP**
  - **4) XCHG**
  - **5) IN**
  - **6) OUT**
  - **7) XLAT**
  - **8) LEA**
  - **9) LDS/LES**
  - **10) LAHF**
  - **11) SAHF**
  - **12) PUSHF**
  - **13) POPF**

Data transfer instructions

- MOV: Copy byte or word from specified source to specified destination.

  Format: MOV <dest>, <source>

  Operation: (dest) ← (source)

  Examples: 1.MOV AX,5000H

  2.MOV AX,BX

  3. MOV AX,[2000H]

  4.MOV AX,[SI]

  5. MOV AX,50H[BX]

Data transfer instructions

- PUSH: pushes the specified register/memory location on to the stack.

  Format: PUSH < source >

  Operation: (SP) ← (SP) – 2

  The stack point is decremented by 2 ,after each execution of the instruction.

  Examples:1.PUSH AX

  2.PUSH [5000H]

Data transfer instructions

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

  Format: POP < destination >

  Operation:(SP) ← (SP) + 2

  The stack point is decremented by 2 ,after each execution of the instruction.

  Examples: 1.POP AX

  2.POP [5000H]

Data transfer instruction

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

  Format: XCHG < destination > , < source >

  Operation: (destination) $\Leftrightarrow$ (Source)

  Examples: XCHG AX, BX

  XCHG [5000H],AX

Data transfer instructions

- IN : Reads the data from input port and store it in accumulator.

  Format : IN <Accumulator>, <Source>

  Example :

1. IN AL,03H :This instruction reads data from an 8-bit port whose address is 03H and store it in AL.
2. IN AX, DX :This instruction reads data a from 16-bit port whose address is in DX and stores it in AX .

Data transfer instructions

- OUT : Writing data to an output port from accumulator.

  Format : OUT <Destination>, AL/AX

  (port) ← AL/AX

  Example:

  OUT 03H,AL ;sends the data available in AL to port whose address is 03h.

  OUT DX,AX ; sends the data available in AX to port whose address is specified in DX.

Data transfer instructions

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

  Format: XLAT

  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

Data transfer instructions

- 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

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

  Format: LDS Reg16,Mem

  (Reg16) ← (Mem)

  (DS) ← (Mem+2)

  Example: LDS BX,5000H

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

  Format: LES Reg16,Mem

  (Reg16) ← (Mem)

  (ES) ← (Mem+2)

  Example: LES BX,5000H

Data transfer instructions

- LAHF: Load AH with 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 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

- PUSHF: Push the flag register to the stack.

  (sp) ← (sp)-2

  the stack pointer is decremented by two for each push operation.

- POPF: loads the flag register from the stack.

  (sp) ← (sp) + 2

  the stack pointer is incremented by 2,for each pop operation.

Arithmetic instructions

- ADD
- ADC
- INC
- DEC
- SUB
- SBB
- CMP
- AAA
- AAS
- AAM

Arithmetic instructions

- AAD
- DAA
- DAS
- NEG
- MUL
- IMUL
- CBW
- CBD
- DIV
- IDIV

Arithmetic instructions

- **ADD**: Adds 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

ADD AX,0100H

ADD AX,[5000H

]

Arithmetic instructions

- ADC: Add with carry
- Add the carry bit to the result.
- Ex: ADC AX,BX

  ADC AX,[5000H]

Arithmetic instructions

- INC: Increment
- Increases the contents of register or memory location by 1
- Ex: INC AX

  INC [5000H]

Arithmetic instructions

- DEC: Decrement
- Subtracts 1 from the contents of specified register or memory location.
- Ex: DEC AX

  DEC [5000H]

Arithmetic instructions

- SUB:Subtract
- 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.
- All the conditional flags are affected by SUB instruction.

Example: SUB AX,BX

SUB AX,1000H

SUB AX,[5000H]

Arithmetic instructions

- SBB: Subtract with borrow
- Subtract 1 from subtraction obtained by SUB.
- Ex: SBB AX,BX

  SBB AX,[5000H]

Arithmetic 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 destinationoperand 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

- 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.

ARITHMETIC INSTRUCTIONS

- Example: Ascii Addition of 6 (36)& 4(34);36+34=6Ah

  1) AL = 6A; (before AAA)

  A > 9 ,hence A + 6 = 1010 + 0110

  = 10000 = 10H & AF = 1

Thus before AAA instruction AL =6AH

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

- 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

- 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.

  EX:

```
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

- AAD: ASCII adjust before division

  The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary in AL.

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

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

  Example: Divide 27 by 5

  mov AX,0207H ; dividend in unpacked BCD form
  mov BL,05H ; divisor in unpacked BCD form

  aad ; AX := 001BH

  div BL ; AX := 0205H

Arithmetic 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

- 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

- 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

- 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

- MUL : Multiplication

  MUL Reg. / Mem.

  This instruction multiplies byte or word by the contents of AL & AX respectively.

  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

  Example: MUL BL

  MUL BX

ARITHMETIC INSTRUNCTIONS

- 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.

  Example: IMUL BL
  IMUL BX

ARITHMETIC INSTRUCTIONS

- CBW : Convert signed byte to 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

  EX:

  1. If AL = 1000 0000(80h)

     Then AH ← 1111 1111(FF_h)

     AX=FF80H

Arithmetic 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.

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

  EX:

1. If AX = 1000 0000 0000 0000(8000H)

   then DX ← 1111 1111 1111 1111 ($FFFF_H$)

Arithmetic instructions

- DIV : division

  DIV <reg./Mem>

  It divides an word or double word by a 16-bit or 8-bit operand.

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

  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.

  EX: DIV BL

  DIV BX

ARITHMETIC 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 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.

  EX: IDIV BL

  IDIV BX

LOGICAL INSTRUCTIONS

- AND
- OR
- NOT
- XOR
- TEST
- SHL/SAL
- SHR
- SAR
- ROR
- ROL
- RCR
- RCL

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 0 0 0 0 = 0000H

LOGICAL INSTRUCTIONS

- EX: AND AX,0008H
- AND AX,BX
- AND AX,[5000H]

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

LOGICAL INSTRUCTIONS

- EX: OR AX,0008H
- OR AX,BX
- OR AX,[5000H]

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 .

  Example: NOT [5000H]

  NOT AX

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

LOGICAL INSTRUCTIONS

- EX: XOR AX,0008H
- XOR AX,BX
- XOR AX,[5000H]

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.

  EX:TEST AX,BX

LOGICAL INSTRUCTIONS

SHL / SAL : Shift Logical / Arithmetic Left

SHL <reg. / Mem>,count

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

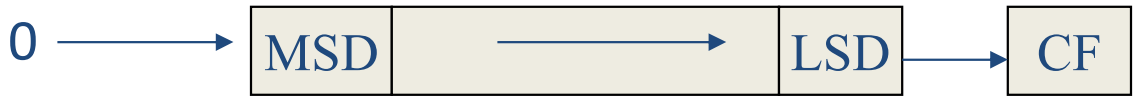
The operand to be shifted can be either register or memory location contents but cannot be immediate data.

LOGICAL INSTRUCTIONS

SHR : Shift Logical Right

SHR <reg. / Mem>,count

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.

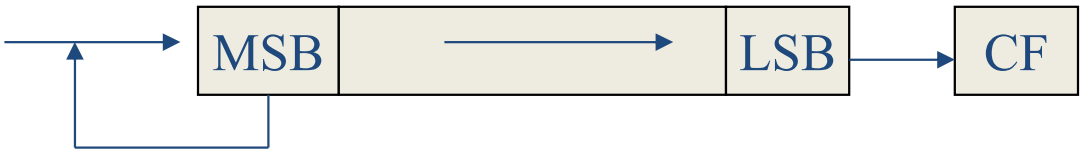
The operand to be shifted can be either register or memory location contents but cannot be immediate data.

LOGICAL INSTRUCTIONS

SAR : Shift Logical Right

SAR <reg. / Mem> , <count>

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



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.

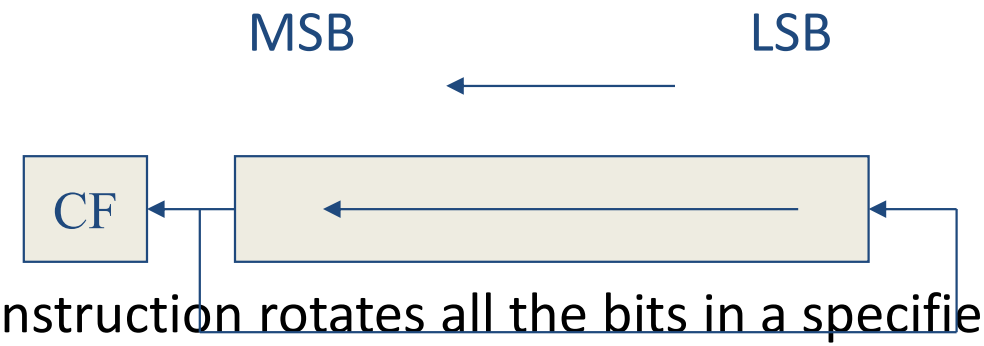
The operand to be shifted can be either register or memory location contents but cannot be immediate data.

LOGICAL INSTRUCTIONS

- 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 operand can be a register or a memory location.

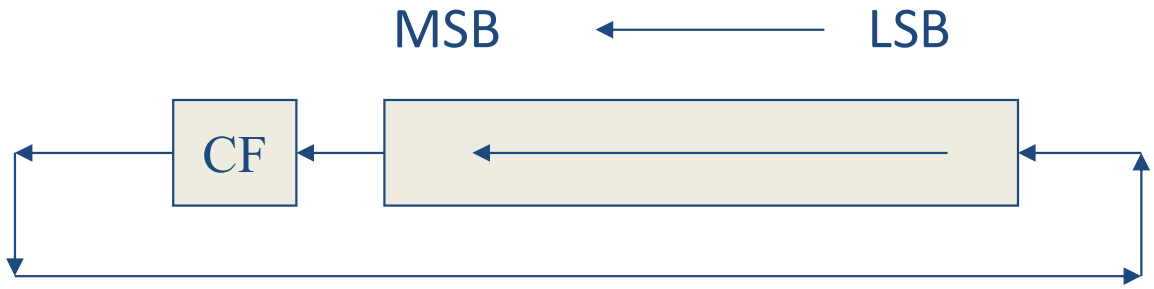
The count will be represented with CL register.

LOGICAL 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.

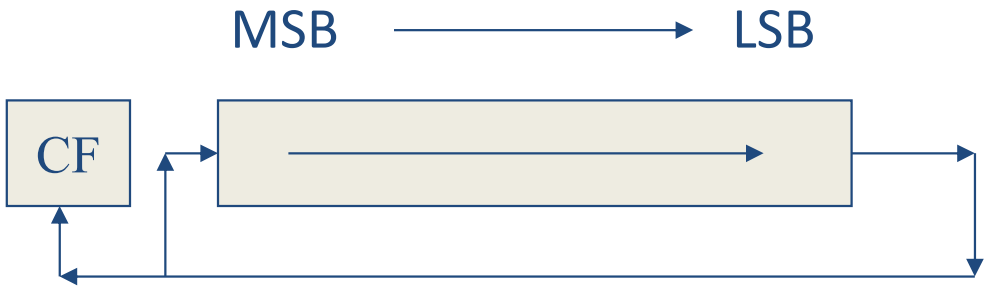
The operand can be a register or a memory location.

LOGICAL 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.

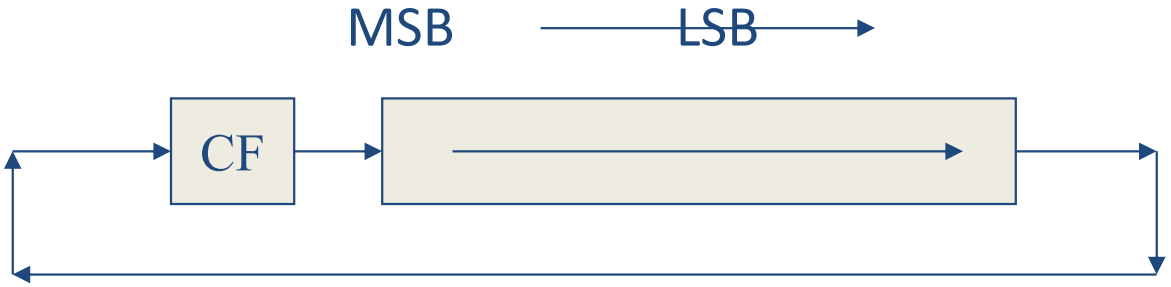
The LSB is pushed into the carry flag as well as the MSB at each operation. The remaining bits are shifted right subsequently by the specified count positions.

LOGICAL 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 operand can be a register or a memory location.

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 the case of 8086, 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.

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 : It is prefix to the other instructions

  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.

- REPNE / REPNZ : Repeat when not equal or till ZF = 0.

String Instructions

String data byte/word manipulation instructions

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

- Moves a string of bytes stored in one set of memory location to another set of memory locations

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

  REP prefix is used with MOVS instruction to repeat it by a value given in CX register.

  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.

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.

  REPE prefix is used with CMPS instruction to repeat it by a value given in CX register.

  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.

String Instructions

String data byte/word manipulation instructions

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

this instruction scans a string of bytes or words for an operand byte or word specified in AL or AX.

The string is points to the DI in the ES.

REPNE prefix is used with the SCAS instruction.

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.

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 into AL / AX register.

  The SI register points to the source string in the DS.

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

String Instructions

String data byte/word manipulation instructions

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

  stores the AL / AX register contents to the string pointed by DI in the ES
  The DI register is automatically incremented or decremented depending on the status of DF.

Flag manipulation & Processor 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 Various Flag manipulation instructions are

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

  WAIT, HLT, NOP, ESC, LOCK

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

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

- WAIT: wait for a TEST input pin to go low

  The WAIT instruction when executed,holds the processor untill TEST input pin goes low.
- HLT: Halt the processor

  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.
- NOP: No operation

  When NOP instruction is executed processor does not perform any operation

Processor Control Instructions

- ESC: Escape to external devices

  ESC instruction when executed, frees the bus for peripheral devices.

- LOCK: Bus lock instruction prefix

  LOCK is a prefix, that may appear with other instuctions. when it is executed,bus access is not allowed for another master untill the LOCK prefix instruction executed completely.

Control transfer (or) Branch instructions

- when a branch instruction is encountered the program execution control is transferred to the specified address.
- This type of instructions are classified into two types.

1) **unconditional branch instructions**:In this execution control is transferred to specified address independent of any condition.

2) **conditional branch instructions:**In this execution control is transferred to specified address depending upon result of previous operation satisfies a particular condition, otherwise execution continues in normal sequence.

Unconditional branch instructions

- CALL
- RET
- INT N
- INTO
- JMP
- IRET
- LOOP

Unconditional branch instructions

- **CALL:** Unconditional call
- This instruction is used to call a subroutine or subprogram or procedure from a main program.
- There are two types of CALL instructions:
- **Intra-segment or near call:**A near call refers to calling a procedure stored in the same code segment memory in which main program resides.
- **Inter-segment or far call:** A far call refers to calling a procedure stored in different code segment memory than that of main program.

Unconditional branch instructions

- **RET:** Return from procedure
- Every procedure or subroutine end with RET instruction.
- Thus the program control return back to main program.

Unconditional branch instructions

- **INT N:** Interrupt type N
- The INT instructions are called Software Interrupts.
- The INT instruction is accompanied by a type number, which can be in the range of 0 to 255. Thus 8086 processor has 256 software interrupts that can be implemented.

  EX: INT 3 → Break-point Interrupt

Unconditional branch instructions

- **INTO:** Interrupt on overflow
- This command is executed,when the overflow flag is set.

Unconditional branch instructions

- **IRET:** Return from ISR
- This instruction is used to terminate an interrupt service procedure and transfer the program control back to main program.

Unconditional branch instructions

- **JMP:** Jump unconditionally
- This instruction unconditionally transfers the execution control to the specified address.
- No flags are affected by this instruction.

Unconditional branch instructions

- **LOOP:** loop unconditionally
- This instruction executes the part of program from the label to loop instruction by cx no.of times.at each iteration,cx is decremented automatically.
- EX:mov cx,0005h

```
        mov ax,0001
label:mov bx,0002h
        Or ax,bx
        loop label
```

conditional branch 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)

conditional branch 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

conditional branch 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

Assembler Directives

- Types of hints are given to the assembler using some predefined alphabetical strings called Assembler Directives.
- Which helps the assembler to correctly understand the asssembly language programs.

Assembler Directives(contd.)

Assume

- Used to tell the assembler the names of the logical segments to be assumed in the program.
- Example, ASSUME CS:CODE ,DS:DATA

DB – Defined Byte

- Used to declare a byte type variable in memory.
- Example, n1 DB 49h

DW – Define Word

- Used to tell the assembler to define a word type variable in memory. ex: n1 DW 1234h

Assembler Directives(contd.)

DD – Define Double Word

- Used to declare a variable of type double word or to reserve a memory location which can be accessed as double word.

DQ – Define Quad word

- Used to tell the assembler to declare the variable as 4 words of storage in memory.

DT – Define Ten bytes

- Used to tell the assembler to declare the variable which is 10 bytes in length or reserve 10 bytes of storage in memory.

Assembler Directives(contd.)

END – End the program

- To tell the assembler to stop fetching the instruction and end the program execution.
- ENDP – it is used to end the procedure (subprogram/subroutine)
- ENDS – used to end the segment.

EQU – Equate

- Used to give name to some value or symbol.
- EX: ADDITION EQU ADD

EVEN – Align on Even memory address

- Tells the assembler to increment the location to the next even address if it is not already at an even address.

Assembler Directives(contd.)

EXTRN: EXTERNAL AND PUBLIC

- Used to tell the assembler that the names ,procedures and labels after this directive have already been defined in some other assembly language module.
- EX: MODULE1 SEGMENT

  PUBLIC FACTORIAL

  MODULE1 ENDS

  MODULE2 SEGMENT

  EXTRN FACTORIAL

  MODULE2 ENDS

GROUP – Group related segment

- Used to tell the assembler to group the logical segments named after the directive into one logical segment.
- This allows the content of all the segments to be accessed from the same group.
- EX: PROGRAM GROUP CODE,DATA,STACK

Assembler Directives(contd.)

LABEL

- Used to give the name to the current value in the location counter.
- The LABEL directive must be followed by a term which specifies the type you want associated with that name.

LENGTH

- Used to determine the number of items in some data such as string or array.
- EX:MOV CX,LENGTH ARRAY

Assembler Directives(contd.)

OFFSET

- It is an operator which tells the assembler to determine the offset or displacement of named data item or procedure from the start of the segment which contains it.
- EX: MOV SI,OFFSET LIST

ORG – Originate

- Tells the assembler to set the location value.
- Example, ORG 7000H sets the location counter value to point to 7000H location in memory.

Assembler Directives(contd.)

PROC – Procedure

- Used to identify the start of the procedure.

PTR – Pointer (BYTE OR WORD)

- Used to assign a specific type to a variable or a label.
- EX:MOV AL, BYTE PTR [SI]

  MOV AX,WORD PTR [SI]

Assembler Directives(contd.)

- **+ & - operators:**
- These operators represents arithmetic addition & subtraction.
- EX:MOV AX,[SI+2]

  MOV DX,[BX-3]
- **SEGMENT:** logical segment
- Ex:CODE SEGMENT

Assembler Directives(contd.)

SHORT

- Used to tell the assembler that only a 1-byte displacement is needed to code a jump instruction.
- If the jump destination is after the jump instruction in the program, the assembler will automatically reserve 2 bytes for the displacement.

TYPE

- Tells the assembler to determine the type of a specified variable.
- The TYPE operator can be used in instruction such as ADD BX, TYPE WORD_ARRAY, where we want to increment BX to point to the next word in an array of words.

Procedures & Macros

- The procedure is a group of repetitive instructions stored as a separate program in the memory and it is called from the main program whenever required.
- Type of procedure depends on where the procedure is stored in the memory.
- If it is in the same code segment where the main program is stored then It is called near procedure otherwise it is referred to as far procedure.
- This procedures are used by CALL and RET instructions.

Procedures & Macros

- The CALL instruction is used to transfer execution to procedure or subprogram. there are two types of CALLs, near & far.
- Near call is a call to a procedure which is in the same code segment as the call instruction.
- Far call is a call to a procedure which is in the different code segment from that which contains the call instruction.
- RET instruction will return the execution from a procedure to the next instruction after call instruction.

Procedures & Macros

- Macro is a group of repetitive instructions.
- The macro assembler generates the code in the program each time where the macro is called, such that it takes ore memory.
- Macros can be defined by MACRO and ENDM.

Differencec b/w Procedures & Macros

Procedures	Macros
Accessed by CALL & RET instuctions during program execution.	Accessed during assembly with name given to macro when defined.
Machine code for instructions is put only once in the memory.	Machine code is generated for instructions each time when macro is called.
With procedures less memory is required.	With macros more memory is required.