* **ARITHMETIC INSTRUCTIONS**
* **ADD Add**

**Operands:-** REG,MEMORY => ADD AX,0100H

Memory,REG

REG,REG => ADD AX,BX

memory , immediate => ADD AX,[1000H]

immediate, Memory

The content of the both the operands are added & the result is stored in destination.

**Eg:** ADD AX,Y ; AX=AX+Y

* **ADC Add With Carry**

Operands REG,MEMORY => ADC AX,0100H

memory,REG =>

REG,REG => ADC CL,BL

memory , immediate => ADC AX,[0300H]

immediate,Memory

The instructions adds the source operands to destination operation along with carry flags& the result is stored in destination operand.

**Eg:** MOV AX,1234H

ADC BX,FFDBH

* **SUB SUBTRACT**

Operands REG,MEMORY => SUB AX,0100H

memory,REG => SUB 0100,AX

REG,REG => SUB AX,BX

memory , immediate => SUB AX,[1000H]

immediate,Memory => SUB [5000H] , 0100

The instructions SUBTRACTS the source operands to destination operation and result is stored in destination operand.

**Eg:** MOV AL,05H

SUB AL,02H ; AL=AL-02H

* **SBB SUBTRACT With BARROW**

Operands REG,MEMORY => SBB AX,0100H

memory,REG => SBB 0100,AX

REG,REG => SBB AX,BX

memory , immediate => SBB AX,[1000H]

immediate,Memory => SBB [5000H],0100

The instructions adds the source operands to destination operation along with the values of carry flags& the result is stored in destination operand.

**Eg:** MOV AX,5678H

SBB AX,4321H

Dest operand = [(Dest operand) – (src operand) – (carry flag)]

* **INC INCREMENT**

Operands REG => INC AX

memory => INC 0100

INCREMENTS THE OPERAND BY ‘1’

This instruction increments the content of the specified register (or) memory location by ‘1’. All the condition code flags are affected except the CF (carry flag).

* **DEC DECREMENT**

Operands REG => DEC BX

memory => DEC 0100

DECRMENTS THE OPERAND BY ‘1’

This instruction decrements instructions subtracts 1 from the content of the specified register (or) memory location . All the condition code flags are affected except the CF (carry flag).

* **MUL MULTIPLICATION ( UNSIGNED MUL)**

Operands REG => MUL BH ; (AX) 🡨 AL \* BH

memory => MUL 0100H ; (AX)🡨 AX \* 0100

The instruction is used for the multiplication of 2 unsigned numbers . The content of specified register (or) memory location can be multiplied by the contents of AL (8-bit) or (16-bit) reg .

The result is placed in AX (for 8-bit manuplation) or DX:AX (if 16-bit )

In case of 16-bit multiplication ,lower significant word is placed AX and higher significant word is placed in DX register .It will affect higher CF and OF

Eg:- 1. MUL BH ; (AX)🡨 AL\* BH

2. MUL CX,(DX)(AX)🡨 (AX)\*(CX)

3. MOV AL, 0FDH

MOV CL, 05H

MUL CL

* **IMUL ( signed multiplication** )

The instruction is used for the multiplication of 2 signed numbers. The instruction multiplies a signed byte in source operand by a signed word in AX. The source can be REG (or) memory operand not immediate data.

The CF and AF are affected

Eg: 1. IMUL CL

2. IMUL CX

3. IMUL [SI]

4. MOV AL. -03H

MOV CL, -05H

IMUL CL

* **DIV ( unsigned division)**

.

* **ASCII INSTRUCTIONS**

THERE ARE 4 INSTRUCTIONS

**A) AAA (ASCII ADJUST AL-REG AFTER ADDITION )**

**B) AAS (ASCII ADJUST AL-REG AFTER SUBTRACTION )**

**C) AAM (ASCII ADJUST AL-REG AFTER MULTIPLICATION )**

**D) AAD (ASCII ADJUST AL-REG BEFORE DIVISION )**

**a) AAA (ASCII ADJUST “AL-REG” 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.
* The numerical data entered in to the computer from a keyword is usually in ASCII code . The digits 0-9 are repressed as 30H-39H.
* The AAA instruction adjusts is done in AL register

**case-1:-**

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 4-bits of LSB’s of AL is between 0 to 9 and AF = 0, AAA sets the 4 bit higher order bits of AL to 0 (cleared).The AH must be cleared before addition.

Eg:-

Before AAA instruction operation

AL = 67 (0110 0111) where 6 is higher and 7 is in lower positions since 7 is in b/w (0-9) so in AL reg the higher bits are set cleared

i.E AL= 07( 0000 0111) 🡪 after AAA instruction

**Case -2 :**

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

**Eg:-**

If AL=9 & BL=3

**Step :1** ASCII form is AL= 39 (0011 1001) & BL =33(0011 0011)

There fore ADD BL,AL

AL= BL+AL (0011 1001 + 0011 0011) = 6C

**Step 2 :**

Since in AL result ‘C’ is higher digit which is in between (A-F) so “06” is added to AL register

i.E AL= AL+06 6C => ( 0110 1100) +(0000 0110) = 72

There fore AL=72 since 2 is least digit so prefix 7 is cleared

i.E AL=02 (un packed)

**Step 3:**  since AH =00 it is added with 01

There fore AH = 01

**Step 4:** since AX = AH AL => AX= 0102(unpacked digit )

b) **AAS (ASCII ADJUST “AL-REG” AFTER SUBTRACTION )**

* This instruction is used to adjust the result in AL reg after performing subtraction.
* When operands are ASCII adjust is done as follows

**i)** if the least significant (LS hexa)of AL reg is < (or) = to 9 and AF=0 the MS hexa of AL reg is cleared (i.e is made 0) and LS hex digit is Unaltered.

**ii)** if the LS hex is of AL reg is > 9 (or) AF=1 adjustments are as follows.

* + A) 6 is subtracted from Ls hex digit of AL reg
  + B)the MSB’s of AL reg are cleared.
  + C) contents of AH are decremented by 1
  + D) carry and AF & CF flags are set to 1
* **Case 1:**
* Eg:- opr1 = 9 & opr2= 5 Ascii form is 39 & 35

Sub opr2 , opr1 39 (0011 1001) - 35( 0011 0101) = 04

i.e AL = 04

**Case 2:**

Eg: opr1 = 5 & opr2 = 9

Ascii form is 35 & 39

35 (0011 0101) - 39 (0011 1001) = FFFF FFFC

i.E AL = FC

Step 2:

Subtract 06 from AL reg

AL= FC – 06 = F6 where 4-bits of MSB’s are cleared from FC i.e AL reg

Step :3

Content s of AH is decremented By 01

AH = 00-01 = FF (this is because in AL reg FC result C is > 9)

Step : 4

Finally AX= FF06

Here AH= FF & AL= 06

**c) AAM (ASCII ADJUST AL-REG AFTER MULTIPLICATION )**

d) **AAD (ASCII ADJUST AL-REG BEFORE DIVISION )**

* **Bit Manipulation Instructions**

**Shift instructions**

* **SHL/SAL (shift logical /arithmetic instruction)**
* This instruction shifts each instruction in each destination

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.

* **SHR : Shift Logical Right**

SHR <reg. / Mem>

0 🡪L(MSB) ; R(n+1) 🡪 R(n) ; R(LSB) 🡪CF

* 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.
* **SAR : Shift Logical Right**

SAR <reg. / Mem> , <count>

L(MSB) 🡪L(MSB) 🡪 R(LSB) 🡪 R(LSB)

* 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.
* **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)

MSB LSB

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

RCL <Reg. / Mem> , <Count>

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

**MSB** **LSB**

* 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.
* **ROR : Rotate right without carry**

ROR <Reg. / Mem> , <Count>

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

**MSB** **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.
* **RCR : Rotate right through i.e., with carry**

RCR <Reg. / Mem> , <Count>

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

**MSB** **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.
* **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:

* 1. Flag manipulation instructions
  2. 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

* **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

* **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

* **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.

* **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.

* **ESC:**

Frees the bus for an external master like a coprocessor or peripheral devices.

* **LOCK:**

Prefix which may appear with another instruction.

When executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely.

Used in case of multiprocessor systems.