Arithmetic Instructions

1) ADD/ADC destination, source
Adds the source to the destination and stores the **result** back **in** the **destination**.
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.

Eg: **ADD AL, 25H**	; *AL ← AL + 25H*
ADD BL, CL	; *BL ← BL + CL*
ADD BX, CX	; *BX ← BX + CX*
ADC BX, CX	; *BX ← BX + CX + Carry Flag*

2) SUB/SBB destination, source
It is similar to ADD/ADC except that it does subtraction.

3) INC destination
Adds "1" to the specified **destination**.
Destination: Register, Memory Location
Note: Carry Flag is NOT affected.

Eg: **INC AX**	; *AX ← AX + 1*
INC BL	; *BL ← BL + 1*
INC BYTE PTR [BX]	; *Increment the **byte** pointed by BX in the Data Segment*
	; *i.e. DS:[BX] ← DS;[BX] + 1*
INC WORD PTR [BX]	; *Increment **word** pointed by BX in the Data Segment*
	; *{DS:[BX], DS:[BX+1]}← {DS:[BX], DS:[BX+1]}+1*

4) DEC destination
It is similar to INC. Here also Carry Flag is NOT affected.

5) MUL source(**unsigned** 8/16-bit register)
If the **source** is **8-bit**, it is **multiplied with AL** and the **result** is stored in **AX** (AH–higher byte, AL–lower byte)
If the **source** is **16-bit**, it is **multiplied** with **AX** and the **result** is stored in **DX-AX** (DX–higher byte, AX–lower byte)
Source: Register, Memory Location
MUL affects AF, PF, SF and ZF.

Eg: **MUL BL**	; *AX ← AL × BL*
MUL BX	; *DX-AX ← AX × BX*
MUL BYTE PTR [BX]	; *AX ← AL × DS:[BX]*

6) IMUL source(**signed** 8/16-bit register)
Same as MUL except that the source is a SIGNED number.

7) DIV source(**unsigned** 8/16-bit register – divisor)
This instruction is used for **UNSIGNED** division.
Divides a **WORD by a BYTE**, OR a **DOUBLE WORD by a WORD.**
If the **divisor is 8-bit** then the **dividend is in AX** register.
After division, the **quotient is in AL** and the **Remainder in AH**.
If the **divisor is 16-bit** then the **dividend is in DX-AX** registers.
After division, the **quotient is in AX** and the **Remainder in DX**.

Source: Register, Memory Location ☺ **For doubts contact Bharat Sir on 98204 08217**
ALL flags are **undefined** after DIV instruction.

Eg: **DIV BL** ; *AX ÷ BL :- AL ← Quotient; AH ← Remainder*
DIV BX ; *{DX,AX} ÷ BX :- AX ← Quotient; DX ← Remainder*

Please Note: If the divisor is 0 or the result is too large to fit in AL (or AX for 16-bit divisor), them 8086 does a Type 0 interrupt (Divide Error).

8) IDIV source(**signed** 8/16-bit register – divisor)
Same as DIV except that it is used for **SIGNED** division.

9) NEG destination
This instruction forms the **2's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **Assume** AL= 0011 0101 = 35 H then
NEG AL ;*AL ← 1100 1011 = CBH. i.e. AL ← 2's Complement (AL)*

10)CMP destination, source
This instruction **compares the source with the destination**.
The source and the destination must be of the same size.
Comparison is **done by internally SUBTRACTING** the **SOURCE form DESTINATION**.
The result of this subtraction is NOT stored anywhere, instead the Flag bits are affected.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **CMP BL, 55H** ; *BL compared with 55H i.e. BL – 55H.*
CMP CX, BX ; *CX compared with BX i.e. CX – BX.*

11)CBW [Convert signed BYTE to signed WORD]
This instruction **copies sign of** the byte in **AL into** all the bits of **AH**.
AH is then called *sign extension of AL.*
No Flags affected.

Eg: Assume
AX = XXXX XXXX **1**001 0001
Then **CBW** gives
AX = **1111 1111 1**001 0001

12)CWD [Convert signed WORD to signed DOUBLE WORD]
This instruction **copies sign of** the **WORD** in **AX into** all the bits of **DX**.
DX is then called *sign extension of AX.*
No Flags affected.

Eg: Assume
AX = **1**000 0000 1001 0001
DX = XXXX XXXX XXXX XXXX
Then **CWD** gives
AX = **1**000 0000 1001 0001
DX = **1111 1111 1111 1111**

Note: Both CBW and CWD are used for Signed Numbers.

Decimal Adjust Instructions

13)DAA [Decimal Adjust for Addition]

It makes the **result** in **packed BCD** form **after** BCD **addition** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 - D_0 > 9$ OR Auxiliary Carry Flag is set => ADD 06H to AL.
If $D_7 - D_4 > 9$ OR Carry Flag is set => ADD 60H to AL.

Assume AL = 14H
CL = 28H
Then **ADD AL, CL** gives
AL = 3CH
Now **DAA** gives
AL = 42 (06 is added to AL as C > 9)
If you notice, $(14)_{10} + (28)_{10} = (42)_{10}$

14)DAS [Decimal Adjust for Subtraction]

It makes the **result** in **packed BCD** form **after** BCD **subtraction** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 - D_0 > 9$ OR Auxiliary Carry Flag is set => Subtract 06H from AL.
If $D_7 - D_4 > 9$ OR Carry Flag is set => Subtract 60H from AL.

Assume AL = 86H
CL = 57H
Then **SUB AL, CL** gives
AL = 2FH
Now **DAS** gives
AL = 29 (06 is subtracted from AL as F > 9)
If you notice, $(86)_{10} - (57)_{10} = (29)_{10}$

ASCII Adjust Instructions (for the AX register ONLY)

15)AAA [ASCII Adjust for Addition]

It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 … 9** are represented as **30 … 39**.
When we **add ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAA** instruction **after the addition** is performed.
AAA updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.
Eg: Assume
AL = 0011 0100 … ASCII 4.
CL = 0011 1000 … ASCII 8.
Then **ADD AL, CL** gives
AL = 01101100
i.e. AL = 6CH … it is the Incorrect temporary Result

Now **AAA** gives
AL = 0000 0010 … Unpacked BCD for 2.
Carry = 1 … this indicates that the answer is 12.

16)AAS [ASCII Adjust for Subtraction]
It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 … 9** are represented as **30 … 39**.
When we **subtract ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAS** instruction **after the subtraction** is performed.
AAS updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.
Eg: Assume
AL = 0011 1001 … ASCII 9.
CL = 0011 0101 … ASCII 5.
Then **SUB AL, CL** gives
AL = 0000 0100
i.e. AL = 04H
Now **AAS** gives
AL = 0000 0100 … Unpacked BCD for 4.
Carry = 0 … this indicates that the answer is 04.

17)AAM [BCD Adjust After Multiplication]
Before we multiply two ASCII digits, we mask their upper 4 bits.
Thus we have two unpacked BCD operands.
After the two unpacked BCD operands are multiplied, the AAM instruction converts this result into unpacked BCD form in the AX register.
AAS updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.
Eg: Assume
AL = 0000 1001 … unpacked BCD 9.
CL = 0000 0101 … unpacked BCD 5.
Then **MUL CL** gives
AX = 0000 0000 0010 1101 = 002DH.
Now **AAM** gives
AX = 0000 0100 0000 0101 = 0405H.
This is 45 in the unpacked BCD form.

18)AAD [Binary Adjust before Division]
This instruction converts the unpacked BCD digits in AH and Al into a Packed BCD in AL.
AAD updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.
Eg: Assume
CL = 07H.
AH = 04.
AL = 03.
∴ AX = 0403H … unpacked BCD for $(43)_{10}$
Then **AAD** gives
AX = 002BH … i.e. $(43)_{10}$
Now **DIV CL** gives (divide AX by unpacked BCD in CL)
AL = Quotient = 06 … unpacked BCD
AH = Remainder = 01 … unpacked BCD