Chapter 07

80 × 86 instruction set

Table of content

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch and Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions

■ Introduction

The 8086 microprocessor supports 8 types of instructions −

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Let us now discuss these instruction sets in detail.

■ Data Transfer Instructions

These instructions are used to **transfer the data from the source operand to the destination operand**. Following are the list of instructions under this group –

- **Instruction to transfer a word**

MOV − Used to copy the byte or word from the provided source to the provided destination.

PUSH − Used to put a word at the top of the stack.

POP − Used to get a word from the top of the stack to the provided location.

PUSHA − Used to put all the registers into the stack.

POPA − Used to get words from the stack to all registers.

XCHG − Used to exchange the data from two locations.

XLAT − Used to translate a byte in AL using a table in the memory.

- **Instructions for input and output port transfer**

IN − Used to read a byte or word from the provided port to the accumulator.

OUT − Used to send out a byte or word from the accumulator to the provided port.

- **Instructions to transfer the address**

LEA − Used to load the address of operand into the provided register.

LDS − Used to load DS register and other provided register from the memory

LES − Used to load ES register and other provided register from the memory.

- **Instructions to transfer flag registers**

LAHF − Used to load AH with the low byte of the flag register.

SAHF − Used to store AH register to low byte of the flag register.

PUSHF − Used to copy the flag register at the top of the stack.

POPF − Used to copy a word at the top of the stack to the flag register.

■ Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group −

- **Instructions to perform addition**

ADD − Used to add the provided byte to byte/word to word.

ADC − Used to add with carry.

INC − Used to increment the provided byte/word by 1.

AAA − Used to adjust ASCII after addition.

DAA − Used to adjust the decimal after the addition/subtraction operation.

- **Instructions to perform subtraction**

SUB − Used to subtract the byte from byte/word from word.

SBB − Used to perform subtraction with borrow.

DEC − Used to decrement the provided byte/word by 1.

NEG − Used to negate each bit of the provided byte/word and add 1/2's complement.

CMP − Used to compare 2 provided byte/word.

AAS − Used to adjust ASCII codes after subtraction.

DAS − Used to adjust decimal after subtraction.

- **Instruction to perform multiplication**

MUL − Used to multiply unsigned byte by byte/word by word.

IMUL − Used to multiply signed byte by byte/word by word.

AAM − Used to adjust ASCII codes after multiplication.

- **Instructions to perform division**

DIV − Used to divide the unsigned word by byte or unsigned double word by word.

IDIV − Used to divide the signed word by byte or signed double word by word.

AAD − Used to adjust ASCII codes after division.

CBW − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.

CWD − Used to fill the upper word of the double word with the sign bit of the lower word.

■ Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

▪ Instructions to perform logical operation

NOT − Used to invert each bit of a byte or word.

AND − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.

OR − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.

XOR − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.

TEST − Used to add operands to update flags, without affecting operands.

▪ Instructions to perform shift operations

SHL/SAL − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.

SHR − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.

SAR − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

- **Instructions to perform rotate operations**

ROL − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].

ROR − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].

RCR − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.

RCL − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

■ String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group −

REP − Used to repeat the given instruction till CX ≠ 0.

REPE/REPZ − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

REPNE/REPNZ − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

MOVS/MOVSB/MOVSW − Used to move the byte/word from one string to another.

COMS/COMPSB/COMPSW − Used to compare two string bytes/words.

INS/INSB/INSW − Used as an input string/byte/word from the I/O port to the provided memory location.

OUTS/OUTSB/OUTSW − Used as an output string/byte/word from the provided memory location to the I/O port.

SCAS/SCASB/SCASW − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.

LODS/LODSB/LODSW − Used to store the string byte into AL or string word into AX.

- **Program Execution Transfer Instructions (Branch and Loop Instructions)**

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

- **Instructions to transfer the instruction during an execution without any condition −**

CALL − Used to call a procedure and save their return address to the stack.

RET − Used to return from the procedure to the main program.

JMP − Used to jump to the provided address to proceed to the next instruction.

- **Instructions to transfer the instruction during an execution with some conditions −**

JA/JNBE − Used to jump if above/not below/equal instruction satisfies.

JAE/JNB − Used to jump if above/not below instruction satisfies.

JBE/JNA − Used to jump if below/equal/ not above instruction satisfies.

JC − Used to jump if carry flag CF = 1

JE/JZ − Used to jump if equal/zero flag ZF = 1

JG/JNLE − Used to jump if greater/not less than/equal instruction satisfies.

JGE/JNL − Used to jump if greater than/equal/not less than instruction satisfies.

JL/JNGE − Used to jump if less than/not greater than/equal instruction satisfies.

JLE/JNG − Used to jump if less than/equal/if not greater than instruction satisfies.

JNC − Used to jump if no carry flag (CF = 0)

JNE/JNZ − Used to jump if not equal/zero flag ZF = 0

JNO − Used to jump if no overflow flag OF = 0

JNP/JPO − Used to jump if not parity/parity odd PF = 0

JNS − Used to jump if not sign SF = 0

JO − Used to jump if overflow flag OF = 1

JP/JPE − Used to jump if parity/parity even PF = 1

JS − Used to jump if sign flag SF = 1

- **Processor Control Instructions**

  These instructions are used to control the processor action by setting/resetting the flag values.

  Following are the instructions under this group −

  STC − Used to set carry flag CF to 1

CLC − Used to clear/reset carry flag CF to 0

CMC − Used to put complement at the state of carry flag CF.

STD − Used to set the direction flag DF to 1

CLD − Used to clear/reset the direction flag DF to 0

STI − Used to set the interrupt enable flag to 1, i.e., enable INTR input.

CLI − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

■ **Iteration Control Instructions**

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

LOOP − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0

LOOPE/LOOPZ − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0

LOOPNE/LOOPNZ − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

JCXZ − Used to jump to the provided address if CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

INT − Used to interrupt the program during execution and calling service specified.

INTO − Used to interrupt the program during execution if OF = 1

IRET − Used to return from interrupt service to the main program

… starting from here we will shed the light on most common instructions

■ Data Transfer Instructions

▪ MOV

MOV	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
	SREG, memory
	memory, SREG
	REG, SREG
	SREG, REG
		Copy operand2 to operand1.
		
		The MOV instruction cannot:
		- set the value of the CS and IP registers.
		- copy value of one segment register to another segment register (should copy to general register first).
		- copy immediate value to segment register (should copy to general register first).
		
		Algorithm:
		operand1 = operand2
		
		Example:
		
		`ORG 100h`
		`MOV AX, 0B800h ; set AX = B800h (VGA memory).`
		`MOV DS, AX ; copy value of AX to DS.`
		`MOV CL, 'A' ; CL = 41h (ASCII code).`
		`MOV CH, 01011111b ; CL = color attribute.`
		`MOV BX, 15Eh ; BX = position on screen.`
		`MOV [BX], CX ; w.[0B800h:015Eh] = CX.`
		`RET ; returns to operating system.`

▪ PUSHA

PUSHA	No operands	**Push all general-purpose registers** AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. Note: this instruction works only on 80186 CPU and later! Algorithm: PUSH AX PUSH CX PUSH DX

		PUSH BX PUSH SP PUSH BP PUSH SI PUSH DI

- **POPA**

POPA	No operands	**Pop all general purpose registers** DI, SI, BP, SP, BX, DX, CX, AX from the stack. SP value is ignored, it is Popped but not set to SP register). Note: this instruction works only on 80186 CPU and later! Algorithm: POP DI POP SI POP BP POP xx (SP value ignored) POP BX POP DX POP CX POP AX

- **XCHG**

XCHG	REG, memory
	memory, REG
	REG, REG
		Exchange values of two operands.
		
		Example:
		
		`MOV AL, 5`
		`MOV AH, 2`
		`XCHG` `AL, AH;` AL = 2, AH = 5
		`XCHG` `AL, AH;` AL = 5, AH = 2
		`RET`

- **IN**

IN		**IN Instruction** - This IN instruction will copy data from a port to the AL or AX register.

		For the Fixed port IN instruction type the 8 – bit port address of a port is specified directly in the instruction. Example: **IN AL,0C8H** ; Input a byte from port 0C8H to AL **IN AX, 34H** ; Input a word from port 34H to AX **A_TO_D EQU 4AH** **IN AX, A_TO_D** ; Input a word from port 4AH to AX For a variable port IN instruction, the port address is loaded in DX register before IN instruction. DX is 16 bit. Port address range from 0000H – FFFFH. Example: **MOV DX, 0FF78H** ; Initialize DX point to port **IN AL, DX** ;Input a byte from a 8 bit port ;0FF78H to AL **IN AX, DX** ;Input a word from 16 bit port to ;0FF78H to AX.

- **OUT**
  - OUT Instruction - The OUT instruction copies a byte from AL or a word from AX or a double from the accumulator to I/O port specified by op.
  - Two forms of **OUT** instruction are available:
    1. Port number is specified by an immediate byte constant, (0 - 255). It is also called as fixed port form.
    2. Port number is provided in the DX register (0 – 65535)
  - Example:

```
OUT 3BH, AL        ;Copy the contents of the AL to port 3Bh
OUT 2CH,AX         ;Copy the contents of the AX to port 2Ch
```

- Example:

```
MOV DX, 0FFF8H      ;Load desired port address in DX
OUT DX, AL          ;Copy the contents of AL to FFF8h
OUT DX, AX          ;Copy content of AX to port FFF8H
```

- **LEA**

LEA	REG, memory	**Load Effective Address**.
		
		Algorithm:
		REG = address of memory (offset)
		
		Example:
		
		`MOV BX, 35h`
		`MOV DI, 12h`
		
		`LEA SI, [BX+DI]` ; SI = 35h + 12h = 47h
		
		Example:
		
		`LEA BX, PRICE` ;Load BX with offset of PRICE in DS
		`LEA BP, SS:STAK` ;Load BP with offset of STAK ;in SS

- **LAHF**
  - Copy low byte of flag register to AH
  - LAHF instruction copies the value of SF, ZF, AF, PF, CF, into bits of 7, 6, 4, 2, 0 respectively of AH register.

```
AH bit:   7    6    5    4    3    2    1    0
        [SF] [ZF] [0] [AF] [0] [PF] [1] [CF]

bits 1, 3, 5 are reserved
```

  - This **LAHF** instruction was provided to make conversion of assembly language programs written for 8080 and 8085 to 8086 easier.

- **ADD**
  - ADD Instruction - These instructions add a number from source to a number from some destination and put the result in the specified destination.
  - The add with carry instruction **ADC**, also add the status of the carry flag into the result.
  - The source and destination must be of same type, means they must be a byte location or a word location.
  - If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

Assembly Language	**Operation**
ADD AL, BL	AL = AL + BL
ADD CX, DI	CX = CX + DI
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD [BX], AL	AL adds to the contents of the data segment memory location address by BX with the sum stored in the same memory location
ADD CL, [BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADC CL, BL	Add contents of BL plus carry status to contents of CL. Results in CL

- Addition of Un Signed numbers

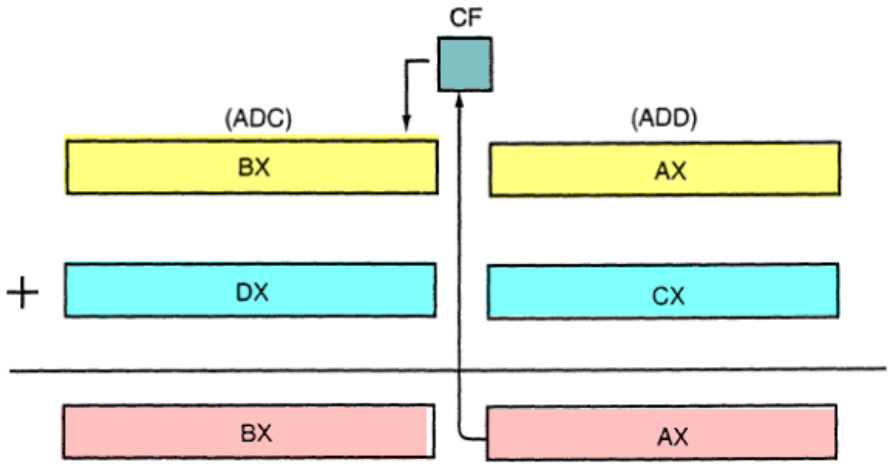
```
ADD CL, BL          ; CL = 01110011 =115 decimal
                    ; + BL = 01001111 = 79 decimal
                    ; Result in CL = 11000010 = 194
                      decimal
```

- Addition of Signed numbers

```
ADD CL, BL          ; CL = 01110011 = + 115 decimal
                    ; + BL = 01001111 = +79 decimal
```

```
                                        ; Result in CL = 11000010 = - 62
                                         decimal
                                        ; Incorrect because result is too
                                        large to fit in 7 bits.
```

- Suppose a program is written for the 8086-80286 to add the 32-bit number in BX and AX to the 32-bit number in DX and CX. Figure below illustrates this addition so the placement and function of carry flag can be understood. This addition cannot be easily performed without adding the carry flag bit because the 8086-80286 only adds 8- or 16-bit numbers.



- **INC**
  - INC Instruction - INC instruction adds one to the operand and sets the flag according to the result. INC instruction is treated as an unsigned binary number.
  - Note: **Carry flag is unchanged**

Example:

```
; AX = 7FFFh
INC AX              ;After this instruction AX = 8000h
INC BL              ; Add 1 to the contents of BL register
```

```
INC CL                ; Add 1 to the contents of CX register.
```

- **DEC**
  - Decrement subtraction (**DEC**) subtracts a 1 from a register or the contents of a memory location.
  - Table below lists some decrement instructions that illustrate register and memory decrements.

Assembly Language	**Operation**
DEC BH	BH=BH—1
DEC CX	CX=CX—1
DEC EDX	EDX = EDX — 1
DEC BYTE PTR [DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC NUMB	Subtracts 1 from the contents of the data segment memory location NUMB

- **SBB — Subtract-with-Borrow**
  - A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference. The most common use for this instruction is for subtractions that are wider than 16-bits in the 8086-80286 or wider than 32-bits in the 80386 through the Pentium Pro. Wide subtractions require that borrows propagate through the subtraction just as wide additions propagate the carry

SBB AH, AL	AH = AH - AL - carry
SBB AX, BX	AX = AX - BX – carry
SBB CL,2	CL= CL-2-carry
SBB BYTE PTR [DI],3	Both a 3 and carry subtract from the contents of the data segment memory location addressed by DI
SBB [Dl], AL	Both AL and carry subtract from the data segment memory location addressed by DI

- **NEG**

NEG	REG
	memory
		Negate. Makes operand negative (two's complement).
		
		Algorithm:
		
		- Invert all bits of the operand
		- Add 1 to inverted operand
		
		Example:
		
		`MOV AL, 5 ; AL = 05h`
		`NEG AL ; AL = 0FBh (-5)`
		`NEG AL ; AL = 05h (5)`

- **CMP — Compare**
  - The comparison instruction (**CMP**) is a subtraction that changes only the flag bits.
  - A comparison is useful for checking the entire contents of a register or a memory location against another value.
  - A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

CMP CL, BL	CL — BL
CMP AX, SP	AX — SP
CMP AX,2000H	AX — 2000H
CMP [DI], CH	CH subtracts from the contents of the data segment memory location addressed by DI
CMP CL, [BP]	The byte contents of the stack segment memory location addressed by BP subtract from CL
CMP AH, TEMP	The byte contents of the data segment memory location TEMP subtract from AH

  - Example below shows a comparison followed by a conditional jump instruction. In this example, the contents of AL are compared with a 10H. Conditional jump instructions that often follow the comparison are JA (jump above) or JB (jump below). If the JA follows the comparison, the jump occurs if the value in

AL is above 10H. If the JB follows the comparison, the jump occurs if the value in AL is below 10H. In this example, the JAE instruction follows the comparison. This instruction causes the program to continue at memory location SUBER if the value in AL is 10H or above. There is also a JBE (jump below or equal) instruction that could follow the comparison to jump if the outcome is below or equal to 10H.

- Example:

```
CMP AL,10H      ; compare with 10H
JAE SURER       ; if 10H or above
```

- **AAA** − Used to adjust ASCII after addition
  - ASCII Adjust after Addition.
  - Corrects result in AH and AL after addition when working with BCD values.
  - It works according to the following Algorithm:

```
if low nibble of AL > 9 or AF = 1 then
      AL = AL + 6
      AH = AH + 1
      AF = 1
      CF = 1
else
      AF = 0
      CF = 0
```

in both cases:

```
      clear the high nibble of AL.
```

Example:

```
MOV AX, 15      ; AH = 00, AL = 0Fh
```

```
AAA                ; AH = 01, AL = 05
```

Example:

```
MOV AH,0           ;Clear AH for MSD
MOV AL,6           ;BCD 6 in AL
ADD AL,5           ;Add BCD 5 to digit in AL
AAA                ;AH=1, AL=1 representing BCD 11.
```

Example:

```
MOV AL,'5'      ;AL=35
ADD AL,'2'      ;add to AL 32 the ASCII for 2
AAA             ;change 67H to 07H
OR AL,30        ;OR AL with 30H to get ASCII
```

Example:

```
SUB AH,AH       ;AH=00
MOV AL,'7'      ;AL=37H
MOV BL,'5'      ;BL=35H
ADD AL,BL       ;37+35=6CH therefore AL=6C
AAA             ;changes 6CH to 02 in AL and AH=CF=1
OR AX,3030H     ;AX=3132 which is the ASCII for 12H
```

- **AAS:** ASCII Adjust after Subtraction
  - Note: AAA and AAS will only work on the AL register. The data can be unpacked BCD rather than ASCII.

Example:

```
MOV AX,0105H          ;AX=0105 unpacked BCD for 15
MOV CL,06             ;CL=06H
SUB AL,CL             ;5-6=-1 (FFH)
AAS                   ;FFH in AL is adjusted to 09 and
                      ;AH is decremented leaving
                      AX=0009
```

- **MUL** – Used to multiply **unsigned** byte by byte/word by word
  - **8-bit Multiplication**

    One of the operands must be in AL. The other operand can be either in a register or in memory. After the multiplication, the result is in AX.

    when operand is a byte:

    AX = AL * operand.

Assembly Language	**Operation**
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX

Example:

```
MOV AL, 200        ; AL = 0C8h
MOV BL, 4
MUL BL             ; AX = 0320h (800)
```

- **16-bit Multiplication**

  One of the operands must be in AX. The other operand can be either in a register or in memory. After the multiplication, the lower word is in AX and the higher word is in DX.

  when operand is a word:

  (DX AX) = AX * operand.

MUL CX	AX is multiplied by CX; the unsigned product is in DX—AX
IMUL DI	AX is multiplied by DI; the signed product is in DX—AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX—AX

- **DIV** − Used to divide the unsigned word by byte or unsigned double word by word

  when operand is a byte:

  AL = AX / operand

  AH = remainder (modulus)

  when operand is a word:

  AX = (DX AX) / operand

  DX = remainder (modulus)

DIV CL	AX is divided by CL; the unsigned quotient is in AL and the remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the remainder is in AH

DIV CX	DX—AX is divided by CX; the unsigned quotient is in AX and the remainder is in DX

DIV NUMB	AX is divided by the contents of the data segment memory location NUMB; the unsigned quotient is in AX and the remainder is in DX

Example

```
MOV AX, 203          ; AX = 00CBh
MOV BL, 4
DIV BL               ; AL = 50 (32h), AH = 3
```

Example

short program that divides the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1. Here the quotient is stored in location ANSQ and the remainder in location ANSR. Notice how the contents of location NUMB are retrieved from memory and then zero-extended to form a 16-bit unsigned number for the dividend.

```
MOV AL,NUMB          ; get NUMB
MOV AH,0             ; zero-extend
DIV NUMB1            ; divide by NUMB1
MOV ANSQ,AL          ; save quotient
MOV ANSR,AH          ; save remainder
```

- **IMUL** – Used to multiply signed byte by byte/word by word

  when operand is a **byte**:

  AX = AL * operand

  when operand is a **word**:

  (DX AX) = AX * operand

  Example:

```
MOV AL, -2
MOV BL, -4
IMUL BL          ; AX = 8
```

IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
IMUL DI	AX is multiplied by DI; the signed product is in DX—AX

- **IDIV** − Used to divide the signed word by byte or signed double word by word
  - This instruction is used to divide a signed word by a signed byte or to divide a signed double word by a signed word.

```
IDIV BL          ; Signed word in AX is divided by signed
                 ; byte in BL
```

  - Example below shows the same basic program, except that the numbers are signed numbers. This means that instead of zero-extending AL into AH, it is sign-extended with the CBW instruction.
  - Example

```
MOV AL,NUMB      ; get NUMB
CBW              ; sign-extend
IDIV NUMB1       ; divide by NUMB1
MOV ANSQ,AL      ; save quotient
MOV ANSR,AH      ; save remainder
```

  - Example below shows the division of two 16-bit signed numbers. Here a -100 in AX is divided by a +9 in CX. The CWD instruction converts the -100 in AX to a -100 in DX-AX before the division. After the division, the results appear in DX-AX as a quotient of -11 in AX and a remainder of -1 in DX.

- Example

```
MOV AX, -100   ; load -100
MOV CX,9       ; load +9
CWD            ; sign-extend
IDIV CX
```

Instructions to perform logical operation

- **NOT** − Used to invert each bit of a byte or word
  - Logical inversion or the one's complement
  - The NOT instruction inverts all bits of a byte, word, or doubleword
  - Invert each bit of the operand.
  - Algorithm:

    if bit is 1 turn it to 0.

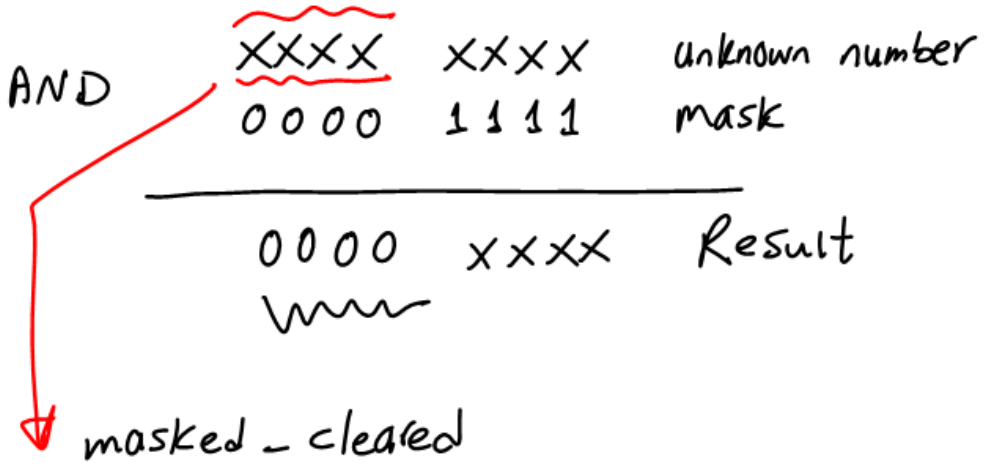
    if bit is 0 turn it to 1.
  - Example:

```
MOV AL, 00011011b
NOT AL          ; AL = 11100100b
```

NOT CH	CH is one's complemented
NOT BX	BX is one's complemented
NOT TEMP	The contents of the data segment memory location TEMP is one's complemented
NOT BYTE PTR[BX]	The byte contents of the data segment memory location addressed by BX is one's complemented

- **AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word
  - The AND operation also clears bits of a binary number. The task of **clearing** a bit in a binary number is called **masking**.
  - Figure below illustrates the process of masking. Notice that the left most four bits clear to 0, because 0 AND anything is 0. The bit

positions that AND with 1's do not change. This occurs because if a 1 ANDs with a 1, a 1 results; if a 1 ANDs with a 0, a 0 results.



- An ASCII-coded number can be converted to BCD by using the AND instruction to mask off the leftmost four binary bit positions. This converts the ASCII 30H to 39H to 0—9. Example below shows a short program that converts the ASCII contents of BX into BCD. The AND instruction in this example converts two digits from ASCII to BCD simultaneously.
- Example:

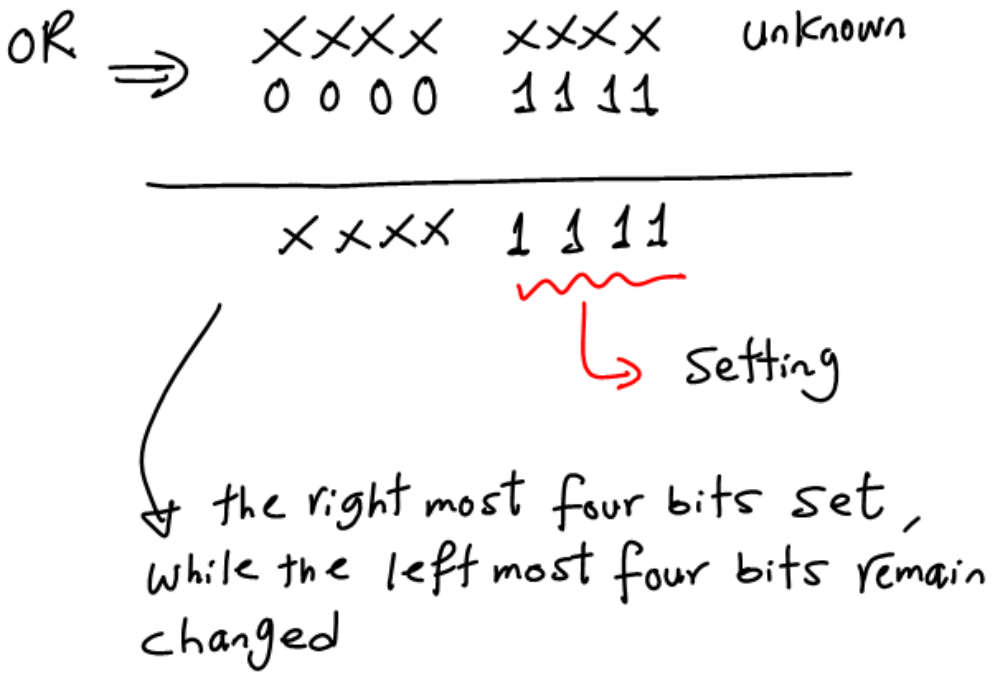
```
MOV BX,3135H   ; load ASCII
AND BX,0F0FH   ; Mask BX
```

AND AL,BL	AL=AL AND BL
AND CX,DX	CX = CX AND DX
AND CL,33H	CL = CL AND 33H
AND DI,4FFFH	DI = DI AND 4FFFH
AND AX,[DI]	AX is ANDed with the word contents of the data segment memory location addressed by DI

- **OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word

  - Figure below shows how the OR gate sets any bit of a binary number. Here, an unknown number (XXXX XXXX) ORs with a

0000 1111 to produce a result of XXXX 1111. The right-most four bits set, while the leftmost four bits remain unchanged.

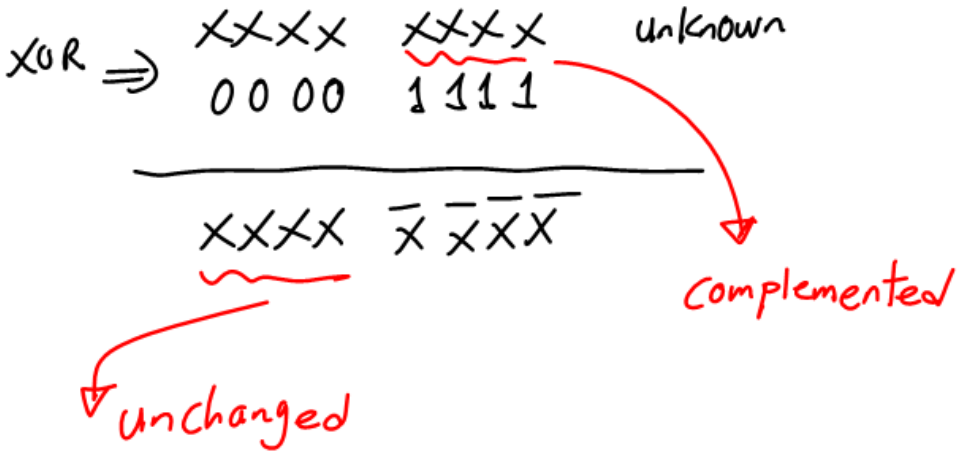


- The OR operation sets any bit, and the AND operation clears any bit. The OR instruction uses any of the addressing modes allowed to any other instruction except segment register addressing. Table below lists several OR instructions and their operations.

OR AH,BL	AH = AH OR BL
OR SI,DX	SI = SI OR DX
OR DH,0A3H	DH = DH OR A3H
OR SP,990DH	SP = SP OR 990DH
OR DX,[BX]	DX is ORed with the word contents of the data segment memory location addressed by BX

- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word

- The Exclusive-OR instruction is useful if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented.



- Figure above shows how just part of an unknown quantity can be inverted by XOR. Notice that when a 1 Exclusive-ORs with X, the result is X. If a 0 Exclusive-ORs with X, the result is X.
- Suppose that the leftmost 10-bits of the BX register must be inverted without changing the rightmost 6-bits. The XOR BX,0FFC0H instruction accomplishes this task.

  The AND instruction clears (0) bits, the OR instruction sets (1) bits, and now the Exclusive-OR instruction inverts bits. These three instructions allow a program to gain complete control over any bit, stored in any register or memory location. This is ideal for control system applications where equipment must be turned on (1), turned off (0) and toggled from on to off or off to on.
- A fairly common use for the Exclusive-OR instruction is to clear a register to zero. For example, the XOR CH,CH instruction clears register CH to 00H and requires two bytes of memory to store the instruction. Likewise, the MOV CH,00H instruction also clears CH to 00H, but requires three bytes of memory. Because of this savings, the XOR instruction is used to clear a register in place of a move immediate.

- **TEST** − Used to add operands to update flags, without affecting operands

  - The TEST instruction performs the AND operation.
  - The difference is that the AND instruction changes the destination operand, while the TEST instruction does not.
  - A TEST affects only the condition of the flag register, which indicates the result of the test.
  - The TEST instruction uses the same addressing modes as the AND instruction. Table below lists some TEST instructions and their operations.

TEST DL,DH	DL is ANDed with DH
TEST CX,BX	CX is ANDed with BX
TEST AH,4	AH is ANDed with 4

  - The TEST instruction functions in the same manner as a CMP instruction. The difference is that the TEST instruction normally tests a single bit (or occasionally multiple bits), while the CMP instruction tests the entire byte or word. The zero flag (Z) is a logic 1 (indicating a zero result) if the bit under test is a zero, and Z = 0 (indicating a non-zero result) if the bit under test is not zero.
  - Usually the TEST instruction is followed by either the JZ (jump if zero) or JNZ (jump if not zero) instruction. The destination operand is normally tested against immediate data. The value of immediate data is 1 to test the rightmost bit position, 2 to test the next bit, 4 for the next, etc.
  - Example below lists a short program that tests the rightmost and leftmost bit positions of the AL register. Here, 1 selects the rightmost bit and 128 selects the leftmost bit. (Note: A 128 is an 80H.) The JNZ instruction follows each test to jump to different

memory locations depending on the outcome of the tests. The JNZ instruction jumps to the operand address (RIGHT or LEFT in the example) if the bit under test is not zero.

- **Example**

```
TEST AL,1       ; Test right bit
JNZ RIGHT       ; if set
TEST AL,128     ; Test left bit
JNZ LEFT        ; if set
```

ROL − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF]

ROR − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF]

RCR − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB

RCL − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB