CHAPTER # 06
"Conditional Processing"

• **AND Instruction:** Performs AND operation b/w each pair of matching bits in two operand & result is stored in destination oprn.

AND destination, source

→ reg, imm & mem all of these can be used.

→ Operands must be of same size.

⇒ AND operation ke zariye hum operands ki koi bhi specifics bit ko clear kr sakte hai for eg:

AL = 11101101

If we want to clear only the 3rd & 7th bit then we will AND AL with 10111011

```
  11101101
& 10111011
----------
  10101001
```

→ baaqi saari bits unchanged rhi only 3rd & 7th are changed.

• **Flags:**

⇒ AND always clears Overflow & Carry Flags. The Sign, Zero & Parity

Converting Upper Case to Lower Case:

```
01 [1] 00001 = 61h ('a')
01 [0] 00001 = 41h ('A')
```

→ only 5th bit is different

So we can change the case by ANDING character with "11011111".

```
.data
arr BYTE "My Name is Sameeze",0
.code
mov ecx, LENGTHOF arr
mov esi, OFFSET arr
L1:
AND BYTE PTR [esi], 11011111b
inc esi
loop L1
```

OR Instruction:

OR destination, source

→ operands same size.

→ It is useful when we want to SET a particular bits without affecting others.

AL = %11011001

we want to set only 2nd one

so

OR AL, 00000010
2nd

Flags: Always clears Carry & Overflow.
Sign, Zero, & parity Flags are changed acc to destination.

→ If we want to get same info of a value we can OR it with itself or ZERO.

Zero flag	Sign Flag	Val in AL is...
0	0	Greater than 0
1	0	Equal to 0
0	1	Less than 0

— SET INSTRUCTIONS —

Set Complement: Complement of a set can be generated using NOT.

NOT eax

Set Intersection: The intersection can be obtained by AND.

Set Union: The union can be obtained by OR.

XOR Instruction:

XOR dest, source

→ A bit exclusive-ORed with 0 retains it's value

→ A bit exclusive ORed with 1 toggle (complements).

→ XOR reverses itself when applied twice to same operand.

x	y	x⊕y	(x⊕y)⊕y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Flags: Always clrs overflow & carry. Modifies SF, ZF & PF acc to dest.

Checking Parity Flag

The parity flag is set when lowest byte of the destination operand of a bitwise opera. or arithematic operation has even parity. Clear when odd pa-

→ To check parity of a number without changing it's value is to XOR the number with ZERO.

```
mov al, 10110101b ; 5 bits = odd
xor al,0          ; Parity Flag=0
mov al, 11001100b ; 4 bits = even
xor al, 0         ; PF=1
```

⇒ 16-bit parity can be checked by XORing upper & lower bytes.

```
mov ax, 64C1h ;0110 0100 1100 0001
xor ah, al    PF=1
```

NOT instruction:

NOT reg or mem

```
mov al, 11110000b
NOT al          ; 00001111b
```

No Flags are affected.

TEST Instruction:

It works same as AND, but it doesn't modify destination operand.

→ It is valuable to find whether multiple bits are set or not

```
test al, 00010010b ; test bit 0&3
```

→ we can infer that ZF=1 only when all tested bits are clear.

```
00100101 → input
00010010 → tested
00000000 → result ZF=0

00100100 → input
00010010 → tested
00000000 → result ZF=1
```

Flags: Always clear OF & CF.
ZF, PF & SF acc to destination.

CMP Instruction:

It performs a subtraction b/w a source operand from destination. Neither operand is modified.

CMP dest, source

Flags: OF, SF, ZF, CF, AF, PF

flags acc to the value the destination operand would have had if actual subtraction had taken place.

⇒ For UnSiged Numbers

CMP Results	ZF	CF
dest < src	0	1
des > src	0	0
des = src	1	0

⇒ Signed Numbers

CMP results	Flags
Dest < src	SF ≠ OF
Dest > src	SF = OF
Dest = src	ZF = 1

Setting & Clearing Individual Flags:

Zero Flag:

→ To Set ⇒ TEST or AND an operand with 0
→ To Clr ⇒ OR an operand with 1

Sign Flag:

→ To Set ⇒ OR the highest bit of oper with 1
→ To Clr ⇒ AND " " " " 0

```
or  al, 80h     ; SF = 1
and al, 7Fh     ; SF = 0
```

Overflow Flag:

→ To Set ⇒ Add two +ve val that produce -ve sum
→ To Clr ⇒ OR an operand with 0.

```
mov al, 7Fh     ; AL = +127
inc al          ; AL = 80h (-128)
                ; OF = 1
or eax, 0       ; OF = 0
```

Carry Flag:

→ To Set ⇒ use STC instr
→ To Clr ⇒ " CLC instr.

```
stc             ; CF = 1
clc             ; CF = 0
```

— Conditional Jump —

→ Types of Conditional Jump Instr

- Jumps Based on Flag Values.

Mnemonic	Description	Flags / Regs
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

- Jumps Based on Equality b/w Operands or val of (E)CX:

Mnemonics	Description
JE	Jump if equal (leftOp = rightOp)
JNE	Jump if not eq (leftOp ≠ rightOp)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Example #01:

```
mov   eax, 65 ;
cmp   eax, 65
JNE   L1:          ; jump not taken
JE    L2 ;         ; = taken
                   bcz equal
```

Example #02:

```
mov   eax, 10
sub   eax, 10
JNE   L1           ; jump not taken
JE    L2           ; jump taken
```

Example #03:

```
mov   CX, 0FFFFh   ; CX = -1
inc   CX ;         ; CX = 0
JCXZ  L2           ; jump taken
```

```
mov   ecx, ecx
JECXZ L2.          ; jump taken
```

FAST
Foundation for Advancement of Science & Technology

- Jump Based on Comparison of Unsigned Oper.

Mnemonics	Description
JA	Jump if above (if leftOP > rightop)
JNBE	Jump if not below or eq (same as JA)
JAE	Jump if above or eq (if leftOP ≥ rightOP)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if leftOP < rightOP)
JNAE	Jump if not above or eq (same as JB)
JBE	Jump if below or eq (if leftOP ≤ rightOP)
JNA	Jump if not above (same as JBE)

```
mov al, +127    ; hex val is 7Fh
cmp al, -128    ; hex val is 80h
ja  L1          ; not taken
jg  L2          ; taken
```

ja was not executed bcz unsigned 7Fh is smaller than 80h. The jg was executed as it is designed for signed. +127 > -128.

- Jumps Based on Signed Oper.

Mnemonics	Description
JG	Jump if greater (if leftOP > rightOP)
JNLE	Jump if not less than or eq (same as JG)
JGE	Jump if greater or eq (if left ≥ right)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if left < right)
JNGE	Jump if not greater than or eq (same as JL)
JLE	Jump if less or eq (if left ≤ right)
JNG	Jump if not greater (same as JLE)

— Conditional Loops —

- LOOPZ & LOOPE :
Both are same as LOOP instruction but have an additional condition; the ZERO FLAG must be SET in order for control to transfer to destination.

LOOPZ (loop if zero).
LOOPE (loop if equal).

- LOOPNZ & LOOPNE :
Same as a LOOP but additional condition is ZERO FLAG must be CLEAR to transfer control to destination.

→ Finding 1st Non-Negative no. ⇒

```
.data
    array DWORD - - -
.code
    mov   esi, OFFSET arr
    mov   ecx, LENGTHOF arr
L1:
    test  WORD PTR [esi], 8000h
    pushfd   ; saving flags bcz add will modify flag val.
    add   esi, TYPE arr
    popfd
    loopnz L1   → when ZF=0 loop stops
    jnz   quit → a label.
```

Sign bit 0 hogi to iska matlab non-negative hua sign bit ka 1 se AND kra rhe hai to agr 0 AND 1 hoga to ZF=0 1 hojayega bcz 0 AND 1 = 0. To loop stops!

⇒ clustersize = 2192
if Gigabyte < 3
clustersize = 4096

```
    mov   clustersize, 8192
    cmp   Gigabyte, 3
    jae   next
    mov   clustersize, 4096
next:
```

agr GB 3 se kam ya equal hoga to next pe jmp hojayega else cluster = 4096 hoga.

⇒ if op1 > op2 then
call r1
else
call r2
endif

```
    mov   eax, op1
    cmp   eax, op2
    jg    L1:
    call  routine2
    jmp   L2:
L1: call routine1
L2:
```

```
if op1 == op2 then
    if x > y then
        call routine 1
    else
        call routine 2
    endif
else
    call routine 3
endif
```

```
mov eax, op1
cmp eax, op2
jne L1:
    mov ebx, x
    cmp ebx, y
    jbe L2:
    call routine 1
    jmp next        → next isliye lagaya taake end main control chala jaye warna L2 execute hojayega.
    L2: call routine 2:
    jmp next
L1:
    call routine 3
next :
```

LOGICAL "AND":

```
if (al > bl) AND (bl > cl) then
    x = 1
endif
```

```
    cmp al, bl
    jbe next     → if false
    cmp bl, cl   → executes if al > bl
    jbe next
    mov x, 1     → executes if bl > cl
next:
```

LOGICAL "OR"

```
if (al > bl) OR (bl > cl) then
    x = 1
endif
```

```
    cmp al, bl
    ja L1        → if true skips 2nd cmp
    cmp bl, cl   → if false then cmp 2nd cmp
    jbe next     → if false mov to next
L1: mov x, 1
next:
```

— WHILE LOOP —

```
while (val1 < val2) {
    val1++;
    val2--;
}
```

→ When implementing while loop, it is convinient to reverse the condition & jmp to endwhile if a condition becomes true.

```
    mov eax, val1
beginWhile:
    cmp eax, val2
    jge endwhile        ; if (val1 > val2) exit loop
    inc eax             ; val1++
    dec val2            ; val2--
    jmp beginWhile      ; repeating loop
endwhile:
    mov val1, eax       ; updating value of val1
```

CONDITIONAL PROCESSING DIRECTIVES

• IF condition	• REPEAT } Do-while
• ELSE	• UNTIL
• ELSEIF condition	• UNTILCXZ → repeats until cx = 0
• ENDIF	
	• BREAK
• WHILE	• CONTINUE
• ENDW	

→ Operators which are valid.

==, !=, >, >=, <, <=, !, &&, ||, &.

→ CARRY? returns True if CF = 1
→ OVERFLOW? " " " OF = 1
→ PARITY? " " " PF = 1
→ SIGN? " " " SF = 1
→ ZERO? " " " ZF = 1

can be used with registers.

eax > 1000h
val1 <= 100
val2 == eax
val3 != ebx

compound expression

(eax > 0) && (eax > 1000h)
!(val1 <= 100) || (val2 <= 1000)

```
mov eax, 6
.IF eax >= 10
    mov result, 1
.ENDIF
```

→ prints value 1 – 10.

```
mov eax, 0
.WHILE eax < 10
    inc eax
    call WriteDec
    call Crlf
.ENDW

mov eax, 0
.REPEAT
    inc eax
    call WriteDec
    call CRLF
.UNTIL eax == 10
```