EL2003 Computer Organization & Assembly Language

Lab 08
Conditional Processing

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

LAB 08

Learning Objectives

a. Boolean and Comparison Instructions
b. Conditional Jumps
c. Conditional Loop Instructions
d. Conditional Structures

Boolean and Comparison Instructions

A programming language that permits decision making lets you alter the flow of control, using a technique known as **conditional branching.**

- **AND Instruction**

It is boolean AND operation between a source operand and destination operand. If both bits equal 1, the result bit is 1; otherwise, it is 0. The operands can be 8, 16, or 32 bits, and they must be the same size.

Syntax: *AND reg, reg*
AND reg, mem
AND reg, imm
AND mem, reg
AND mem, imm

```
mov al,10101110b
and al,11110110b                ; result in AL = 10100110
```

The AND instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand.

NOTE: The All other instructions use the same operand combinations and sizes as the AND instruction.

- **OR Instruction**

It is boolean AND operation between a source operand and destination operand. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1.

```
mov al,11100011b
or al,00000100b                 ; result in AL = 11100111
```

The OR instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand.

- **XOR Instruction**

The XOR instruction performs a boolean exclusive-OR operation between each pair of matching bits in two operands and stores the result in the destination operand. If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1.

The XOR instruction always clears the Overflow and Carry flags. XOR modifies the Sign, Zero, and Parity flags.

- **NOT Instruction**

The NOT instruction toggles (inverts) all bits in an operand. The result is called the one's complement.

Syntax: *NOT reg*
NOT mem

```
mov al,11110000b
not al                          ; AL = 00001111b
```

- **TEST Instruction**

The TEST instruction performs an implied AND operation between each pair of matching bits in two operands and sets the Sign, Zero, and Parity flags based on the value assigned to the destination operand.

The only difference between TEST and AND is that TEST does not modify the destination operand.

Testing Multiple Bits: The TEST instruction can check several bits at once. Suppose we want to know whether bit 0 or bit 3 is set in the AL register. We can use the following instruction to find this out:

```
test al,00001001b               ; test bits 0 and 3
```

EXAMPLE

```
.code
        mov   al, 10101110b     ; Clear only bit 3
        and   al, 11110110b     ; AL = 10100110

        mov   al,  11100011b    ; set bit 2
        or    al, 00000100b     ; AL = 11100111

        mov   al,  10110101b    ; 5 bits means odd parity
        xor   al, 0             ;PF=0(PO)

        mov   al,  10100101b    ; 4 bits means even parity
        xor   al, 0             ;PF=1(PE)
```

```
mov   al,  11110000b
not   al                    ; AL = 00001111b

mov   al, 00100101b
test  al, 00001001b         ;ZF=0

mov   al, 00100101b
test  al, 00001000b         ;ZF=1

call  DumpRegs
exit
```

- **CMP Instruction**

  CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand for comparison. Neither operand is modified.

  Syntax:

  CMP destination, source

Flags: The CMP instruction changes the Overflow, Sign, Zero, Carry, Auxiliary Carry, and Parity flags according to the value the destination operand.

When ***two unsigned operands*** are compared, the Zero and Carry flags indicate the following relations between operands:

- Destination < source ZF=0 CF=1
- Destination > source ZF=0 CF=0
- Destination = source ZF=1 CF=0

When ***two signed operands*** are compared, the Sign, Zero, and Overflow flags indicate the following relations between operands:

- Destination < source SF!=OF
- Destination > source SF=OF
- Destination = source ZF=1

Examples: Let's look at three code fragments showing how flags are affected by the CMP instruction.

```
.code
    mov   ax, 5
    cmp   ax, 10        ; ZF = 0    and    CF = 1
```

```
mov   ax, 1000
cmp   ax, 1000      ;ZF=1        and    CF =0

mov   si, 106
cmp   si, 0         ;ZF=0        and    CF=0
```

- **Conditional Jumps**

1. an operation such as CMP, AND, or SUB modifies the CPU status flags.
2. a conditional jump instruction tests the flags and causes a branch to a new address.

- **Jcond Instruction**

A conditional jump instruction branches to a destination label when a status flag condition is true.
Syntax:

```
Jcond destination
```

The conditional jump instructions can be divided into four groups:

- **Jumps based on Flag values**

Mnemonic	Description	Flags / Registers
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**

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

- **Jumps based on unsigned comparisons**

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* ≥ *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* ≤ *rightOp*)
JNA	Jump if not above (same as JBE)

- **Jumps based on signed comparisons**

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)

EXAMPLE

; This program compares and finds larger of the two integers

```
INCLUDE Irvine32.inc
.data
        var1 DWORD 500
        var2 DWORD 125
        larger DWORD ?
.code
main PROC
        mov     eax, var1
        mov     larger, eax
        mov     ebx, var2
        cmp     eax, ebx
        jae     L1
        mov     larger, ebx
        L1:
        exit
main ENDP
END main
```

EXAMPLE

```
; This program compares and finds smallest of the three integers

.data
        var1 DWORD 50
        var2 DWORD 25
        var3 DWORD 103
        msg BYTE "The smallest integer is: ", 0
.code
        mov   eax, var1
        cmp   eax, var2
        jbe     L1
        mov   eax, var2
        L1:
        cmp   eax, var3
        jbe     L2
        mov   eax, var3
        L2:
        mov   edx, OFFSET msg
        call    WriteString
        call crlf
        call    WriteDec
        call crlf
        exit
```

- **Conditional Loop Instructions**

- **LOOPZ and LOOPE Instructions**

The LOOPZ (loop if zero) instruction works just like the LOOP instruction except that it has one additional condition: The Zero flag must be set in order for control to transfer to the destination label.

The syntax is

LOOPZ destination

The LOOPE (loop if equal) instruction is equivalent to LOOPZ and they share the same opcode.

- **LOOPNZ and LOOPNE Instructions**

The LOOPNZ (loop if not zero) instruction is the counterpart of LOOPZ. The loop continues while the unsigned value of ECX is greater than zero (after being decremented) and the Zero flag is clear.

The syntax is LOOPNZ destination

The LOOPNE (loop if not equal) instruction is equivalent to LOOPNZ and they share the same opcode.

Example:

; The following take input from user until user press 0

```
.code
        mov ecx,5
        L1:
        CALL readInt
        cmp eax, 0
        LOOPNZ  L1
call DumpRegs
```

- **Conditional Structures**

We define a conditional structure to be one or more conditional expressions that trigger a choice between different logical branches. Each branch causes a different sequence of instructions to execute.

- **Block-Structured IF Statements**

An IF structure implies that a boolean expression is followed by two lists of statements; one per-formed when the expression is true, and another performed when the expression is false.

```
if( boolean-expression )
  statement-list-1
else
  statement-list-2
```

If structure: In High level Vs Assembly Language

```
if( op1 == op2 ) then
{
    X = 1;
    Y = 2;
}
```

```
        mov   eax,op1
        cmp   eax,op2              ; op1 == op2?
        jne   L1                   ; no: skip next
        mov   X,1                  ; yes: assign X and Y
        mov   Y,2
  L1:
```

```
if( ebx <= ecx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    ja  next
    mov eax,5
    mov edx,6
next:
```

If-else structure: In High level Vs Assembly Language

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

```
    mov eax,op1
    cmp eax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

- **Compound Expression with AND**

When implementing the logical AND operator in compound expression, if the first expression is false, the second expression is skipped.

```
if (al > bl) AND (bl > cl)
  X = 1;
```

```
    cmp al,bl                  ; first expression...
    ja  L1
    jmp next
L1:
    cmp bl,cl                  ; second expression...
    ja  L2
    jmp next
L2:                            ; both are true
    mov X,1                    ; set X to 1
next:
```

- **Compound Expression with OR**

  When implementing the logical OR operator in compound expression, if the first expression is true, the second expression is skipped.

```
if (al > bl) OR (bl > cl)
  X = 1;
```

```
   cmp al,bl                ; is AL > BL?
   ja  L1                   ; yes
   cmp bl,cl                ; no: is BL > CL?
   jbe next                 ; no: skip next statement
L1: mov X,1                 ; set X to 1
next:
```

- **<u>While Loops</u>**

  A WHILE loop is really an IF statement followed by the body of the loop, followed byan unconditional jump to the top of the loop.

```
while( eax < ebx)
   eax = eax + 1;
```

```
top: cmp eax,ebx            ; check loop condition
     jae next               ; false? exit loop
     inc eax                ; body of loop
     jmp top                ; repeat the loop
next:
```

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
top: cmp ebx,val1           ; check loop condition
     ja  next               ; false? exit loop
     add ebx,5              ; body of loop
     dec val1
     jmp top                ; repeat the loop
next:
```

ACTIVITY:

Task#1
Write a program that takes four input integers from the user. Then compare and display a message whether these integers are equal or not.

Task#2
Use cmp and jumps to find the first non-zero value in the given array:

intArr SWORD 0, 0, 0, 150, 120, 35, -12, 66, 4, 0

Task#3 Implement the following given code in Assembly and Consider var = 5 , edx = var+1 and counter value from array initialized in task#2.

```
if ( var < ecx )   AND          (ecx >= edx)    then
        x = 0
else
        x = 1
```

Task#4 Implement the following given code in Assembly and consider var = 0.

```
while ( var <= 10)
        if (var < 5)
                Print "Hello"
        else
                Print "World"
var = var + 1
end while
```

Task#5
Write a program for sequential search. Take an input from the user and find if it occurs in the following array:

arr WORD 10, 4, 7, 14, 299, 156, 3, 19, 29, 300, 20

Task#6
Write a program for bubble sort on this array.

arr WORD 10, 4, 7, 14, 299, 156, 3, 19, 29, 300, 20

Task#7
Write a program to print weekday based on given number.

Task#8
Write a program to check whether a character is alphabet or not.