LAB 08

Computer Organization & Assembly Language (COAL) | EL 2003

Student Name: _____

Roll No.: _____

Section: _____

Marks Awarded _____

Prepared by: Talha Shahid

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
(NUCES), KARACHI

Objectives:

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures

Section 01 | 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.

Input		Output
A	B	Y = A . B
0	0	0
0	1	0
1	0	0
1	1	1

Syntax:

```
AND reg, reg
AND reg, mem
AND reg, imm
AND mem, reg
AND mem, imm
```

Example:

```
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

The OR instruction performs a Boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand:

The OR instruction uses the same operand combinations as the AND instruction:

Syntax:

```
OR reg,reg
OR reg,mem
OR reg,imm
OR mem,reg
OR mem,imm
```

The operands can be 8, 16, or 32 bits, and they must be the same size. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1. The following

Input		Output
A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```
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.

Input		Output
A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
mov   al, 11100011b
xor   al, al          ; result in AL = 00000000
```

NOT Instruction

The NOT instruction toggles (inverts) all bits in an operand. The result is called the **one's complement.** The following operand types are permitted:

Syntax:

```
NOT reg
NOT mem
```

Output	
A	$Y = \overline{A}$
0	1
1	0

Example:

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

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:

Example:

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

From the following example data sets, we can infer that the Zero flag is set only when all tested bits are clear:

```
0 0 1 0 0 1 0 1 → input value
0 0 0 0 1 0 0 1 → test value
0 0 0 0 0 0 0 1 → result: ZF = 0

0 0 1 0 0 1 0 0 → input value
0 0 0 0 1 0 0 1 → test value
0 0 0 0 0 0 0 0 → result: ZF = 1
```

Note: The TEST instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in the same way as the AND instruction.

CMP Instruction

In Intel assembly language we use the CMP (compare) instruction to compare integers. Character codes are also integers, so they work with CMP as well.

```
if A > B {
      …
}

while (X > 0 & X < 200){
      …
}

if (check_for_error == true) {
      …
}
```

The CMP instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified:

Syntax:

```
CMP destination, source
```

CMP uses the same operand combinations as the AND instruction.

The CMP instruction changes the Overflow, Sign, Zero, Carry, Auxiliary Carry, and Parity flags according to the value the destination operand would have had if actual subtraction had taken place. When two unsigned operands are compared, the Zero and Carry flags indicate the following relations between operands:

CMP Results	ZF	CF
Destination < Source	0	1
Destination > Source	0	0
Destination = Source	1	0

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

CMP Results	Flags
Destination < Source	SF != OF
Destination > Source	SF = OF
Destination = Source	ZF = 1

Example

Let's look at three code fragments showing how flags are affected by the CMP instruction. When AX equals 5 and is compared to 10, the Carry flag is set because subtracting 10 from 5 requires a borrow:

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

Comparing 1000 to 1000 sets the Zero flag because subtracting the source from the destination produces zero:

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

Comparing 105 to 0 clears both the Zero and Carry flags because subtracting 0 from 105 would generate a positive, nonzero value.

```
mov si, 105
cmp si, 0                    ; ZF = 0 and CF = 0
```

Section 02 | 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
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 of 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 ?
     msg BYTE "The larger integer is: ", 0

.code
     main PROC
          mov eax, var1
          mov larger, eax
          mov ebx, var2
          cmp eax, ebx
          jae L1
          mov larger, ebx
```

```
          mov eax, larger

          L1:
          mov edx, OFFSET msg
          call WriteString
          call WriteDec
          exit
     main ENDP
END main
```

Example: This program compares and finds smaller of the three integers

```
INCLUDE Irvine32.inc

.data
     var1 DWORD 50
     var2 DWORD 25
     var3 DWORD 103
     msg BYTE "The smaller integer is: ", 0

.code
     main PROC
          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 WriteDec
          exit
     main ENDP
END main
```

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.

Syntax:

```
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.

Syntax:

```
LOOPNZ destination
```

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

Example:

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

Section 04 | 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) {
     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 eax, 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 & 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:
```

While Loops

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

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

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