6

Conditional Processing

OUTLINES

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

6.1 CONDITIONAL BRANCHING

- A programming language that permits decision making lets you alter the flow of control, using a technique known as *conditional branching.*
  - How can I use the Boolean operations AND, OR, NOT?
  - How do I write an IF statement in assembly language?
  - How are nested-IF statements translated by compilers into machine language?
  - How can I set and clear individual bits in a binary number?
  - How can I perform simple binary data encryption?
  - How are signed numbers differentiated from unsigned numbers in boolean expressions?

- Assembly language provides all the tools you need for decision-making logic

6.2 BOOLEAN AND COMPARISON INSTRUCTIONS

Status Flags (Revision)

- The **Zero** flag is set when the result of an operation equals zero.
- The **Carry** flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The **Sign** flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The **Overflow** flag is set when an instruction generates an invalid signed result.
- The **Parity** flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The **Auxiliary Carry** flag is set when an operation produces a carry out from bit 3 to bit 4.

AND Instruction: performs a boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand:

AND *destination, source*

- The following operand combinations are permitted

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

- Operands can be 8, 16, or 32 bits and must be of same size.

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

- AND instruction always clears **Overflow** and **Carry** flag.
  - Also can modify Sign, Zero, and Parity in a way that is consistent with the value assigned to the destination operand.

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

OR *destination, source*

- The OR instruction uses the same operand combinations as the AND instruction.

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

OR instruction and the Flags

- Clears Overflow, Cary
- Modifies Sign, Zero, and Parity in a way that is consistent with the value assigned to the destination operand

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:

```
XOR     destination, source
```

- The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions
- The XOR instruction always clears the Overflow and Carry flags.
- XOR modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand

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

```
NOT reg
NOT mem
```

- No flags are affected by the NOT instruction.

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

TEST Instruction: performs an AND operation between each pair of matching bits in two operands but does not modify the destination operand.

- `TEST` is particularly valuable for finding out whether individual bits in an operand are set.
- The TEST instruction permits the same operand combinations as the AND instruction.
  - E.g. Suppose we want to know whether bit 0 or bit 3 is set in the AL register.

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

- 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 :we use the CMP instruction to compare integers

- The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified, but flags are affected:

CMP *destination, source*

- When two unsigned operands are compared:

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

- When two signed operands are compared:

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

Example: destination > source

```
mov al,5
cmp al,-2           ; Sign flag == Overflow flag
```

Example: destination < source

```
mov al,-1
cmp al,5            ; Sign flag != Overflow flag
```

6.3 CONDITIONAL JUMPS

Conditional Structures: you can implement high level logic instructions using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement:

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(instruction).

Example 1:

```
cmp eax,0
jz L1                    ; jump if ZF = 1
.
.
L1:
```

Example 2:

```
and dl,10110000b
jnz L2                   ; jump if ZF = 0
.
.
L2:
```

Jcond Instruction: A *conditional jump instruction* branches to a destination label when a status flag condition is true. Otherwise, if the flag condition is false, the instruction immediately following the conditional jump is executed.

```
Jcond destination
```

JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

Types of Conditional Jump Instructions:

- Jumps based on specific flag values
- Jumps based on equality between operands or the value of (E)CX
- Jumps based on comparisons of unsigned operands
- Jumps based on comparisons of signed operands.

Table 6-2 Jumps Based on Specific 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

Table 6-3 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
JRCXZ	Jump if RCX = 0 (64-bit mode)

Example 1:

```
mov  edx,0A523h
cmp  edx,0A523h
jne  L5                          ; jump not taken
je   L1                          ; jump is taken
```

Example 2:

```
mov  bx,1234h
sub  bx,1234h
jne  L5                          ; jump not taken
je   L1                          ; jump is taken
```

Example 3:

```
mov    cx,0FFFFh
inc    cx
jcxz   L2                        ; jump is taken
```

Example 4:

```
xor    ecx,ecx
jecxz  L2                        ; jump is taken
```

Unsigned Comparisons: The jumps in following table are only meaningful when comparing unsigned values. Signed operands use a different set of jumps.

TABLE 6-4 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)

Signed Comparisons

TABLE 6-5 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 1

```
mov   edx,-1
cmp   edx,0
jnl   L5                ; jump not taken (-1 >= 0 is false)
jnle  L5                ; jump not taken (-1 > 0 is false)
jl    L1                ; jump is taken (-1 < 0 is true)
```

Example 2

```
mov   bx,+32
cmp   bx,-35
jng   L5                ; jump not taken (+32 <= -35 is false)
jnge  L5                ; jump not taken (+32 < -35 is false)
jge   L1                ; jump is taken (+32 >= -35 is true)
```

Example 3

```
mov  ecx,0
cmp  ecx,0
jg   L5                   ; jump not taken (0 > 0 is false)
jnl  L1                   ; jump is taken (0 >= 0 is true)
```

Example 4

```
mov  ecx,0
cmp  ecx,0
jl   L5                   ; jump not taken (0 < 0 is false)
jng  L1                   ; jump is taken (0 <= 0 is true)
```

6.4 CONDITIONAL LOOP INSTRUCTIONS

LOOPZ and LOOPE Instructions

- The LOOPZ (loop if zero) instruction works just like the LOOP instruction except the Zero flag must be set in order for control to transfer to the destination label.

`LOOPZ` ***destination***

  - `ECX ← ECX - 1`
  - if `ECX > 0` and `ZF=1`, jump to *destination*

- The `LOOPE` (loop if equal) instruction is equivalent to `LOOPZ`.

LOOPNZ and LOOPNE Instructions

- The loop continues while the unsigned value of ECX is greater than zero (after being decremented) and the Zero flag is clear.

LOOPNZ *destination*

  - `ECX ← ECX - 1;`
  - if `ECX > 0` and `ZF=0`, jump to *destination*

- The `LOOPNE` (loop if not equal) instruction is equivalent to `LOOPNZ`.

6.5 CONDITIONAL STRUCTURES

- We define a *conditional structure* to trigger a choice between different logical branches.
  - Each branch causes a different sequence of instructions to execute.

- Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

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

```
    mov eax,op1
    cmp eax,op2
    jne elsepart
    mov X,1
    jmp exit
elsepart:
    mov X,2
```

Implement the following pseudocode in assembly language. All values are unsigned:

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

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

(There are multiple correct solutions to this problem.)

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )
  var3 = 10;
else
{
  var3 = 6;
  var4 = 7;
}
```

```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

(There are multiple correct solutions to this problem.)

Compound Expressions

- ***Logical AND Operator***

When implementing the logical AND operator, consider that HLLs use short-circuit evaluation.

```
if (a1 > b1) AND (b1 > c1)
    X = 1
end if
```

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

One possible implementation . . .

```
    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:
```

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

```
   cmp al,bl                ; first expression...
   jbe next                 ; quit if false
   cmp bl,cl                ; second expression...
   jbe next                 ; quit if false
   mov X,1                  ; both are true
next:
```

- ***Logical OR Operator***

```
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 L1
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 = eax + 1;
```

A possible implementation:

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

SUMMARY

- Conditional Branching
- Boolean and Comparison Instructions
- Conditional Jumps
  - Jumps based on specific flag values
  - Jumps based on equality between operands or the value of (E)CX
  - Jumps based on comparisons of unsigned operands
  - Jumps based on comparisons of signed operands.
- Conditional Loop Instructions
  - LOOPZ and LOOPE
  - LOOPNZ and LOOPNE
- Conditional Structures