Chapter 6
Conditional Processing

6.1 Introduction  150
6.2 Boolean and Comparison Instructions  151
   6.2.1 The CPU Flags  151
   6.2.2 AND Instruction  152
   6.2.3 OR Instruction  153
   6.2.4 XOR Instruction  154
   6.2.5 NOT Instruction  155
   6.2.6 TEST Instruction  155
   6.2.7 CMP Instruction  156
   6.2.8 Setting and Clearing Individual CPU Flags  157
   6.2.9 Section Review  157
6.3 Conditional Jumps  158
   6.3.1 Conditional Structures  158
   6.3.2 Jcond Instruction  158
   6.3.3 Types of Conditional Jump Instructions  159
   6.3.4 Conditional Jump Applications  163
   6.3.5 Bit Testing Instructions (Optional)  167
   6.3.6 Section Review  168
6.4 Conditional Loop Instructions  169
   6.4.1 LOOPZ and LOOPE Instructions  169
   6.4.2 LOOPNZ and LOOPNE Instructions  169
   6.4.3 Section Review  170
6.5 Conditional Structures  170
   6.5.1 Block-Structured IF Statements  170
   6.5.2 Compound Expressions  173
   6.5.3 WHILE Loops  174
   6.5.4 Table-Driven Selection  177
   6.5.5 Section Review  178
6.6 Application: Finite-State Machines  179
   6.6.1 Validating an Input String  180
   6.6.2 Validating a Signed Integer  180
   6.6.3 Section Review  183
6.7 Decision Directives  184
   6.7.1 Signed and Unsigned Comparisons  185
   6.7.2 Compound expressions  186
   6.7.3 .REPEAT and .WHILE Directives  188
6.8 Chapter Summary  189
6.9 Programming Exercises  190
Chapter 6
Conditional Processing

6.1 Introduction 150

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

6.2 Boolean and Comparison Instructions 151
6.2.1 The CPU Flags 151

• 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 when viewed as an unsigned integer.
• 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 (bit 7 carry is XORed with bit 6 Carry).
• The Parity flag is set when an instruction generates an even number of 1’s 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

6.2.2 AND Instruction 152

• Performs a Boolean AND operation between each pair of matching bits in two operands
• Syntax:
  
  \[
  \text{AND } \text{destination}, \text{source}
  \]
• AND instruction is often used to clear selected bits and preserve others.

```
  0 0 1 1 1 0 1 1
  AND 0 0 0 0 1 1 1 1
  cleared 0 0 0 0 1 0 1 1
          unchanged
```

• Application
  o Task: Convert the character in AL to upper case
  o Solution: Use the AND instruction to clear bit 5
    
    mov al, 'a' ; AL = 01100001b (61h = 'a')
    and al, 11011111b ; AL = 01000001b (41h = 'A') clear bit 5
6.2.3 OR Instruction

- Performs a Boolean OR operation between each pair of matching bits in two operands
- Syntax:
  \[ \text{OR destination, source} \]
- OR instruction is often used to set selected bits and preserve others.

\[
\begin{array}{c}
00111011 \\
\text{OR} \\
00001111 \\
\hline
\text{set}\hspace{1cm}00111111
\end{array}
\]

- Application
  - Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
  - Solution: Use the OR instruction to set bits 4 and 5.
    
    ```
    mov al, 6      ; AL = 00000110b (06h)
    or al, 00110000b ; AL = 00110110b (36h = ‘6’)
    ```

6.2.4 XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
- Syntax:
  \[ \text{XOR destination, source} \]
- XOR is a useful way to toggle (inverted) the bits in an operand.

\[
\begin{array}{c}
00111011 \\
\text{XOR} \\
00001111 \\
\hline
\text{inverted}\hspace{1cm}00110100
\end{array}
\]

- XOR reverses itself when applied twice to the same operand.
  \[ (X \oplus Y) \oplus Y = X \]
6.2.5  NOT Instruction

- Performs a Boolean NOT operation on a single destination operand.
- Syntax:
  
  \[ \text{NOT} \ \text{destination} \]
- The result is called the one's complement.

\[
\begin{array}{c}
\text{NOT} \\
0 1 1 1 0 1 1 \\
\hline
1 1 0 0 0 1 0 0 \quad \text{inverted}
\end{array}
\]

- For example, the one's complement of F0 is 0Fh:
  \[
  \begin{align*}
  \text{mov} \ \text{al}, \ 11110000b \quad &; \ AL = 11110000b \ (F0h) \\
  \text{not} \ \text{al} \quad &; \ AL = 00001111b \ (0Fh) \end{align*}
  \]

6.2.6  TEST Instruction

- Performs a nondestructive AND operation between each pair of matching bits in two operands.
- Syntax:
  
  \[ \text{TEST} \ \text{destination}, \ \text{source} \]
- No operands are modified, but the Zero flag is affected.
- Example:
  - Value 00001001 in this example is called a bit mask. Zero flag is set only when all tested bits are clear.
    \[
    \begin{align*}
    \text{mov} \ \text{al}, \ 00100101b \quad &; \ AL = 00100101b \\
    \text{test} \ \text{al}, \ 00001001b \quad &; \ AL = \text{00100101b} \ ZF = 0 \ \text{test bits 0 and 3} \\
    \text{mov} \ \text{al}, \ 00100100b \quad &; \ AL = 00100100b \\
    \text{test} \ \text{al}, \ 00001001b \quad &; \ AL = \text{00100100b} \ ZF = 1 \ \text{test bits 0 and 3}
    \end{align*}
    \]
6.2.7 CMP Instruction

- Compares the destination operand to the source operand
- Nondestructive subtraction of source from destination (destination operand is not changed)
- Syntax: CMP destination, source
- No operands are modified
- When two unsigned operands are compared, the Zero and Carry flags indicate the following relations between operands:
  - destination < source
    mov a1,4
cmp a1,5 ; CF = 1, SF = 1, ZF = 0, OF = 0
  - destination > source
    mov a1,6
cmp a1,5 ; CF = 0, SF = 0, ZF = 0, OF = 0
  - destination = = source
    mov a1,5
cmp a1,5 ; CF = 0, SF = 0, ZF = 1, OF = 0
- When two signed operands are compared, the sign, Zero, and Overflow flags indicate the following relations between operands:
  - destination < source
    mov a1,-1
cmp a1,5 ; CF = 0, SF = 1, ZF = 0, OF = 0 SF != OF
  - destination > source
    mov a1,5
cmp a1,-1 ; CF = 1, SF = 0, ZF = 0, OF = 0 SF == OF
  - destination = = source
    mov a1,5
cmp a1,5 ; CF = 0, SF = 0, ZF = 1, OF = 0 ZF = 1
6.2.8 Setting and Clearing Individual CPU Flags

- **Zero Flag**
  - ZF = 1: Test or AND an operand with Zero
  - ZF = 0: OR an operand with 1
    ```
    test al, 0    ; ZF = 1
    and al, 0     ; ZF = 1
    or al, 1      ; ZF = 0
    ```

- **Sign Flag**
  - SF = 1: OR the highest bit of an operand with 1
  - SF = 0: AND the highest bit with 0
    ```
    or al, 80h    ; SF = 1
    and al, 7Fh   ; SF = 0
    ```

- **Carry flag**
  - CF = 1: STC instruction
  - CF = 0: CLC instruction
    ```
    stc          ; CF = 1
    clc          ; CF = 0
    ```

- **Overflow Flag**
  - OF = 1: Add two positive byte values that produce a negative sum
  - OF = 0: OR an operand with 0
    ```
    mov al, 7Fh   ; AL = +127
    inc al        ; OF = 1, AL = 80 (-128)
    or al, 0      ; OF = 0
    ```
6.3 Conditional Jumps  158

- Jumps Based On:
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed comparisons
- Condition Jump Applications
- Encrypting a String
- Bit Test (BT) Instruction

6.3.1 Conditional Structures  158

- Using a combination of **comparisons and jumps**
  - **First**, an operation such as CMP, AND, or SUB modifies the CPU flags
  - **Second**, a condition jump instruction tests the flags and causes a branch to a new address.

6.3.2 Jcond Instruction  158

- A conditional jump instruction branches to a label when specific register or flag conditions are met
  - **JC**: jump if CF = 1; jump to a label if the Carry flag is set
  - **JNC**: jump if CF = 0; jump to a label if the Carry flag is clear
  - **JZ**: jump if ZF = 1; jump to a label if the Zero flag is set
  - **JNZ**: jump if ZF = 0; jump to a label if the Zero flag is clear
- **Jcond Ranges**
  - Prior to the 386:
    - Jump must be within **–128 to +127 bytes** from current location counter
  - IA-32 processors:
    - 32-bit offset permits jump **anywhere** in memory
6.3.3 Types of Conditional Jump Instructions

- Jumps Based on Specific Flag Values
  - Application:
    - Jump to label L2 if the doubleword in memory pointed to by EDI is even
      \[
      \text{test DWORD PTR [edi],1} \\
      \text{jz L2 ; Jump if zero}
      \]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF = 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if signed</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not signed</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity (even)</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if not parity (odd)</td>
<td>PF = 0</td>
</tr>
</tbody>
</table>

- Jumps Based on Equality
  - Application 1:
    - Jump to label L1 if the memory word pointed to by ESI equals Zero
      \[
      \text{cmp WORD PTR [esi],0} \\
      \text{je L1 ; Jump if equal}
      \]
  - Application 2:
    - Task: Jump to label L1 if bits 0, 1, and 3 in AL are all set.
    - Solution: Clear all bits except bits 0, 1, and 3. Then, compare the result with 00001011 binary.
      \[
      \text{and al,00001011b ; clear unwanted bits} \\
      \text{cmp al,00001011b ; check remaining bits} \\
      \text{je L1 ; all set? jump to L1}
      \]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>Jump if equal ((\text{leftOp} = \text{rightOp}))</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal ((\text{leftOp} \neq \text{rightOp}))</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX = 0</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if ECX = 0</td>
</tr>
</tbody>
</table>
• Jumps Based on Unsigned Comparisons
  o Application 1:
    ▪ Task: Jump to a label if unsigned EAX is greater than EBX
    ▪ Solution: Use CMP, followed by JA
      
      ```
      cmp eax,ebx
      ja Larger ; Jump if above
      ```
  o Application 2:
    ▪ Jump to label L1 if unsigned EAX is less than or equal to Val1
      
      ```
      cmp eax,Val1
      jbe L1 ; Jump if below or equal
      ```
  o Application 3:
    ▪ Compare unsigned AX to BX, and copy the larger of the two into a variable named Large
      
      ```
      mov Large,bx
      cmp ax,bx
      jna Next ; Jump if not below (jump if AX <= BX)
      mov Large,ax
      Next:
      ```

  TABLE 6-4 Jumps Based on **Unsigned** Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if above (if leftOp &gt; rightOp)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal (same as JA)</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal (if leftOp &gt;= rightOp)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below (same as JAE)</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below (if leftOp &lt; rightOp)</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above or equal (same as JB)</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal (if leftOp &lt;= rightOp)</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above (same as JBE)</td>
</tr>
</tbody>
</table>
Jumps Based on Signed Comparisons

- Application 1:
  - Task: Jump to a label if signed EAX is greater than EBX
  - Solution: Use CMP, followed by JG
    ```assembly
cmp eax, ebx
  jg Greater ; Jump if Greater
```

- Application 2:
  - Jump to label L1 if signed EAX is less than or equal to Val1
    ```assembly
cmp eax, Val1
  jle L1 ; Jump if less than or equal
```

- Application 3:
  - Compare signed AX to BX, and copy the smaller of the two into a variable named Small
    ```assembly
  mov Small, ax
  cmp bx, ax
  jnl Next ; Jump if not less
  mov Small, bx
  Next:
```

### TABLE 6-5 Jumps Based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if greater (if leftOp &gt; rightOp)</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if not less than or equal (same as JG)</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater than or equal (if leftOp &gt;= rightOp)</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less (same as JGE)</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less (if leftOp &lt; rightOp)</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater than or equal (same as JL)</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal (if leftOp &lt;= rightOp)</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if not greater (same as JLE)</td>
</tr>
</tbody>
</table>
6.3.4 Conditional Jump Applications

- Example: String Encryption Program
  - Tasks:
    - Input a message (string) from the user
    - Encrypt the message
    - Display the encrypted message
    - Decrypt the message
    - Display the decrypted message

```assembly
TITLE Encryption Program
(Encrypt.asm)

; This program demonstrates simple symmetric encryption using the XOR instruction.
; Chapter 6 example.
; Last update: 06/01/2006

INCLUDE Irvine32.inc
KEY = 239 ; any value between 1-255
BUFSIZE = 128 ; maximum buffer size

.data
sPrompt BYTE "Enter the plain text: ",0
sEncrypt BYTE "Cipher text: ",0
sDecrypt BYTE "Decrypted: ",0
buffer BYTE BUFSIZE+1 DUP(0)
bufSize DWORD ?

.code
main PROC
  call InputTheString ; input the plain text
  call TranslateBuffer ; encrypt the buffer
  mov edx,OFFSET sEncrypt ; display encrypted message
  call DisplayMessage
  call TranslateBuffer ; decrypt the buffer
  mov edx,OFFSET sDecrypt ; display decrypted message
  call DisplayMessage
  exit
main ENDP

;-----------------------------------------------------
InputTheString PROC
;
; Prompts user for a plaintext string. Saves the string
; and its length.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
pushad
  mov edx,OFFSET sPrompt ; display a prompt
  call WriteString
  mov ecx,BUFSIZE ; maximum character count
  push ecx
  call ReadString
  mov buffer,esi
  mov bufSize,edx
popad
```
mov edx,OFFSET buffer ; point to the buffer
call ReadString ; input the string
mov bufSize, eax ; save the length
call Crlf
call Crlf
popad
ret
InputTheString ENDP

;-----------------------------------------------------
DisplayMessage PROC
;
; Displays the encrypted or decrypted message.
; Receives: EDX points to the message
; Returns: nothing
;-----------------------------------------------------
pushad
call WriteString
mov edx,OFFSET buffer ; display the buffer
call WriteString
call Crlf
call Crlf
popad
ret
DisplayMessage ENDP

;-----------------------------------------------------
TranslateBuffer PROC
;
; Translates the string by exclusive-ORing each
; byte with the encryption key byte.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
pushad
mov ecx,bufSize ; loop counter
mov esi,0 ; index 0 in buffer
L1:
xor buffer[esi],KEY ; translate a byte
inc esi ; point to next byte
loop L1
popad
ret
TranslateBuffer ENDP
END main
6.3.5 Bit Testing Instructions (Optional)

- Copies bit $n$ from an operand into the Carry flag
- Syntax:
  
  \[
  \text{BT } \text{bitBase}, n
  \]
- Example: jump to label L1 if bit 9 is set in the AX register:
  
  \[
  \begin{align*}
  & \text{bt AX, 9} \quad ; \text{CF} = \text{bit 9} \\
  & \text{jc L1} \quad ; \text{jump if Carry}
  \end{align*}
  \]
6.4 Conditional Loop Instructions

6.4.1 LOOPZ and LOOPE Instructions

- LOOPZ (loop if zero) permits a loop to continue while Zero flag is set and the unsigned value of ECX is greater than zero.
- LOOPE (loop if equal) instruction equivalent to LOOPZ.
- Syntax:
  - LOOPE destination
  - LOOPZ destination
- Logic:
  - ECX = ECX – 1
  - if ECX > 0 and ZF=1, jump to destination

6.4.2 LOOPNZ and LOOPNE Instructions

- LOOPNZ (loop if not zero) permits a loop to continue while the unsigned value of ECX is greater than zero and Zero flag is clear.
- LOOPNE (loop if not equal) instruction equivalent to LOOPNZ.
- Syntax:
  - LOOPNE destination
  - LOOPNZ destination
- Logic:
  - ECX = ECX – 1
  - if ECX > 0 and ZF=0, jump to destination
- Useful when scanning an array for the first element that matches a given value
- Example: finds the first positive value in an array

```assembly
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
mov esi,OFFSET array
mov ecx,LENGTHOF array
L1:
test WORD PTR [esi],8000h ; test sign bit
pushfd ; push flags on stack
add esi,TYPE array
popfd ; pop flags from stack
loopnz L1 ; continue loop
jnz quit ; none found
sub esi,TYPE array ; ESI points to value
quit:
```
6.5 Conditional Structures  170

6.5.1  Block-Structured IF Statements     170

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

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

```assembly
mov eax, op1
cmp eax, op2
jne L1
mov X, 1
jmp L2
L1:  mov X, 2
L2:
```
6.5.2 Compound Expressions

- Compound Expression with AND
  - When implementing the logical AND operator, consider that high-level languages compilers for Java, C, and C++ use short-circuit evaluation for efficiency reasons.
  - In the following example, if the first expression is false, the second expression is skipped:
    \[
    \text{if (al > bl) AND (bl > cl)} \\
    \quad X = 1;
    \]
    - This is one possible implementation:
      \[
      \begin{align*}
      \text{cmp al,bl} & \quad ; \text{first expression...} \\
      \text{ja L1} & \quad \\
      \text{jmp next} & \\
      \text{L1:} & \\
      \text{cmp bl,cl} & \quad ; \text{second expression...} \\
      \text{ja L2} & \\
      \text{jmp next} \\
      \text{L2:} & \quad ; \text{both are true} \\
      \text{mov X,1} & \quad ; \text{set X to 1}
      \end{align*}
      \]
  - But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:
    \[
    \begin{align*}
    \text{cmp al,bl} & \quad ; \text{first expression...} \\
    \text{jbe next} & \quad ; \text{quit if false} \\
    \text{cmp bl,cl} & \quad ; \text{second expression...} \\
    \text{jbe next} & \quad ; \text{quit if false} \\
    \text{mov X,1} & \quad ; \text{both are true}
    \end{align*}
    \]
- Non-Short-Circuit Evaluation
  - Some language (BASIC, for example), do not perform short-circuit evaluation.
  - Implementation such a compound expression in assembly language is tricky because a flag or Boolean value is needed to hold the result form the first expression:
    \[
    \begin{align*}
    \text{mov temp,0} & \quad ; \text{clear temp flag} \\
    \text{cmp al,bl} & \quad ; \text{AL > BL?} \\
    \text{jna L1} & \quad ; \text{no} \\
    \text{mov temp,1} & \quad ; \text{yes: set true flag} \\
    \text{L1:} & \\
    \text{cmp bl,cl} & \quad ; \text{BL > CL?} \\
    \text{jna L2} & \quad ; \text{no} \\
    \text{mov temp,1} & \quad ; \text{yes: set true flag} \\
    \text{L2:} & \\
    \text{cmp temp,1} & \quad ; \text{flag equal to true?} \\
    \text{jne next} & \\
    \text{mov X,1} & \\
    \text{next:}
    \end{align*}
    \]
• Compound Expression with OR (1 of 2)
  o When implementing the logical OR operator, consider that high-level languages
    compilers for Java, C, and C++ use **short-circuit** evaluation for **efficiency** reasons.
  o In the following example, if the first expression is true, the second expression is **skipped**:
    
    ```
    if (al > bl) OR (bl > cl)
    X = 1;
    ```
  o We can use "**fall-through**" logic to keep the code as short as possible:
    
    ```
    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
    ```

6.5.3 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
• Consider the following example:
  ```
  while(eax < ebx)
  eax = eax + 1;
  ```
• This is a possible implementation:
  ```
  top:
  cmp eax,ebx   ; check loop condition
  jae next      ; false? exit loop
  inc eax       ; body of loop
  jmp top       ; repeat the loop
  next:
  ```
6.5.4 Table-Driven Selection

- Table-driven selection uses a table lookup to replace a multiway selection structure.
- Create a table containing lookup values and the offsets of labels or procedures.
- Use a loop to search the table.
- Suited to a large number of comparisons.

Steps to do it:
- Step 1: Create a table containing lookup values and procedure offsets:
  ```
  .data
  CaseTable BYTE 'A' ; lookup value
  DWORD Process_A ; address of procedure
  EntrySize = ($ - CaseTable)
  BYTE 'B'
  DWORD Process_B
  BYTE 'C'
  DWORD Process_C
  BYTE 'D'
  DWORD Process_D
  NumberOfEntries = ($ - CaseTable) / EntrySize
  ```
- Step 2: Use a loop to search the table. When a match is found, we call the procedure offset stored in the current table entry:
  ```
  mov ebx,OFFSET CaseTable ; point EBX to the table
  mov ecx,NumberOfEntries ; loop counter
  L1:
  cmp al,[ebx] ; match found?
  jne L2 ; no: continue
  call NEAR PTR [ebx + 1] ; yes: call the procedure
  jmp L3 ; and exit the loop
  L2:
  add ebx,EntrySize ; point to next entry
  loop L1 ; repeat until ECX = 0
  L3:
  ```
6.6 Application: Finite-State Machines  179

- A **finite-state machine (FSM)** is a graph structure that changes state based on some input, also called a **state-transition diagram**
- Use a graph to represent an FSM, with squares or circles called **nodes**, and lines with arrows between the circles called **edges** (or **arcs**)
- A FSM is a specific instance of a more general structure called a **directed graph (or digraph)**.
- Three basic states, represented by nodes:
  - Start state
  - Terminal state(s)
  - Nonterminal state(s)
- Accepts any sequence of symbols that puts it into an **accepting (final) state**
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
- Advantages:
  - Provides visual tracking of program's flow of control
  - Easy to modify
  - Easily implemented in assembly language

6.6.1 Validating an Input String  180

- FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':
  - The following input strings would be recognized by this FSM:
    - xaabcdefgz
    - xz
    - xyyqqrrstuvz
6.6.2 Validating a Signed Integer

- FSM that recognizes signed integers:

  ![ FSM Diagram ]

  o The following is code from State A in the Integer FSM:

    ```assembly
    StateA:
    call Getnext ; read next char into AL
    cmp al,'+' ; leading + sign?
    je StateB ; go to State B
    cmp al,'-' ; leading - sign?
    je StateB ; go to State B
    call IsDigit ; ZF = 1 if AL = digit
    jz StateC ; go to State C
    call DisplayErrorMsg ; invalid input found
    jmp Quit
    
    o IsDigit: Receives a character in AL. **Sets the Zero flag** if the character is a decimal digit.

    ```assembly
    IsDigit PROC
    cmp al,'0' ; ZF = 0
    jb ID1
    cmp al,'9' ; ZF = 0
    ja ID1
    test ax,0 ; ZF = 1
    ID1: ret
    IsDigit ENDP
    ```
6.7 Decision Directives  184

- Using the .IF Directive
  - .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.
  - Examples:

```plaintext
Example1:
  .IF eax > ebx
    mov edx,1
  .ELSE
    mov edx,2
  .ENDIF

Example2:
  .IF eax > ebx && eax > ecx
    mov edx,1
  .ELSE
    mov edx,2
  .ENDIF
```

- MASM generates "hidden" code for you, consisting of code labels, CMP and conditional jump instructions

```plaintext
.data
val1  SDWORD 5
result SDWORD ?
.code
mov eax,6
  .IF eax > val1
    mov result,1
  .ENDIF
```

- Relational and Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 == expr2</td>
<td>Returns true when expression1 is equal to expr2.</td>
</tr>
<tr>
<td>expr1 != expr2</td>
<td>Returns true when expr1 is not equal to expr2.</td>
</tr>
<tr>
<td>expr1 &gt; expr2</td>
<td>Returns true when expr1 is greater than expr2.</td>
</tr>
<tr>
<td>expr1 &gt;= expr2</td>
<td>Returns true when expr1 is greater than or equal to expr2.</td>
</tr>
<tr>
<td>expr1 &lt; expr2</td>
<td>Returns true when expr1 is less than expr2.</td>
</tr>
<tr>
<td>expr1 &lt;= expr2</td>
<td>Returns true when expr1 is less than or equal to expr2.</td>
</tr>
<tr>
<td>!expr</td>
<td>Returns true when expr is false.</td>
</tr>
<tr>
<td>expr1 &amp;&amp; expr2</td>
<td>Performs logical AND between expr1 and expr2.</td>
</tr>
<tr>
<td>expr1</td>
<td></td>
</tr>
<tr>
<td>expr1 &amp; expr2</td>
<td>Performs bitwise AND between expr1 and expr2.</td>
</tr>
<tr>
<td>CARRY?</td>
<td>Returns true if the Carry flag is set.</td>
</tr>
<tr>
<td>OVERFLOW?</td>
<td>Returns true if the Overflow flag is set.</td>
</tr>
<tr>
<td>PARITY?</td>
<td>Returns true if the Parity flag is set.</td>
</tr>
<tr>
<td>SIGN?</td>
<td>Returns true if the Sign flag is set.</td>
</tr>
<tr>
<td>ZERO?</td>
<td>Returns true if the Zero flag is set.</td>
</tr>
</tbody>
</table>
• **.REPEAT Directive**
  - Executes the loop body before testing the loop condition associated with the .UNTIL directive
  - **Example:**
    ```
    ; Display integers 1 – 10:
    mov eax,0
    .REPEAT
      inc eax
      call WriteDec
      call Crlf
    .UNTIL eax == 10
    ```

• **.WHILE Directive**
  - Tests the loop condition before executing the loop body The .ENDW directive marks the end of the loop.
  - **Example:**
    ```
    ; Display integers 1 – 10:
    mov eax,0
    .WHILE eax < 10
      inc eax
      call WriteDec
      call Crlf
    .ENDW
    ```
6.8 Chapter Summary  189

- Bitwise instructions manipulate individual bits in operands
  - AND, OR, XOR, NOT, TEST

- **CMP** instruction
  - compares operands using implied \textit{subtraction}
  - sets condition flags

- Four types of \textbf{conditional jump} instructions are shown in this chapter. Jumps based on
  - Equality: JE (jump equal), JNE (jump not equal), ...
  - Flag values: JC (jump carry), JZ (jump zero), JNC , JP, ...
  - Signed: JG (jump if greater), JL (jump if less), JNG (jump not greater), ...
  - Unsigned: JA (jump if above), JB (jump if below), JNA (jump not above), ...

- **Loops**
  - The LOOPZ (LOOPE) instruction repeats when the Zero flag is \textbf{set} and ECX is greater than Zero
  - The LOOPNZ (LOOPNE) instruction repeats when the Zero flag is \textbf{clear} and ECX is greater than zero.

- **Encryption** is a process that encodes data, and **decryption** is a process that decodes data.
  - The XOR instruction can be used to perform simple encryption and decryption, one byte at a time.

- Flowcharts are an effective tool for visually representing program logic.

- **Finite-state machine** (FSM) is an effective tool for validating string containing recognizable characters such as signed integers.

- Simplify assembly language coding
  - The \texttt{.IF, .ELSE, .ELSEIF, and .ENDIF} directives evaluate runtime expressions and greatly simplify assembly language coding. They are particularly useful when coding complex compound Boolean expression.
  - You can also create conditional loops, using the \texttt{.WHILE and .REPEAT}. 

---

CMPS293&290 Class Notes (Chap 06)    Page 23 / 23    Kuo-pao Yang