# Lecture No.15

## Lecture Outlines

6.3 Conditional Jumps (Continued)
- 6.3.4 Conditional Jump Applications

6.4 Conditional Loop Instructions
- 6.4.1 LOOPZ and LOOPE Instructions
- 6.4.2 LOOPNZ and LOOPNE Instructions

6.5 Conditional Structures
- 6.5.1 Block-Structured IF Statements
- 6.5.2 Compound Expressions
- 6.5.3 WHILE Loops
- 6.5.4 Table-Driven Selection

### 6.3.4 Conditional Jump Applications

*Testing Status Bits* One of the things assembly language does best is bit testing. Often, we do not want to change the values of the bits we're testing, but we do want to modify the values of CPU status flags. Conditional jump instructions often use these status flags to determine whether or not to transfer control to code labels. Suppose, for example, that an 8-bit memory operand named **status** contains status information about an external device attached to the computer. The following instructions jump to a label if bit 5 is set, indicating that the device is offline:

```
mov   al,status
test  al,00100000b              ; test bit 5
jnz   DeviceOffline
```

The following statements jump to a label if any of the bits 0, 1, or 4 are set:

```
mov   al,status
test  al,00010011b              ; test bits 0,1,4
jnz   InputDataByte
```

Jumping to a label if bits 2, 3, and 7 are all set requires both the AND and CMP instructions:

```
mov   al,status
and   al,10001100b              ; mask bits 2,3,7
cmp   al,10001100b              ; all bits set?
je    ResetMachine              ; yes: jump to label
```

*Larger of Two Integers* The following code compares the unsigned integers in EAX and EBX and moves the larger of the two to EDX:

```
  mov  edx,eax                  ; assume EAX is larger
  cmp  eax,ebx                  ; if EAX is >= EBX
  jae  L1                       ;    jump to L1
  mov  edx,ebx                  ; else move EBX to EDX
L1:                             ; EDX contains the larger integer
```

*Smallest of Three Integers* The following instructions compare the unsigned 16-bit values in the variables V1, V2, and V3 and move the smallest of the three to AX:

```
.data
V1 WORD ?
V2 WORD ?
V3 WORD ?
.code
    mov   ax,V1                ; assume V1 is smallest
    cmp   ax,V2                ; if AX <= V2
    jbe   L1                   ;   jump to L1
    mov   ax,V2                ; else move V2 to AX
L1: cmp   ax,V3                ; if AX <= V3
    jbe   L2                   ;   jump to L2
    mov   ax,V3                ; else move V3 to AX
L2:
```

*Loop until Key Pressed* In the following 32-bit code, a loop runs continuously until the user presses a standard alphanumeric key. The *ReadKey* method from the Irvine32 library sets the Zero flag if no key is present in the input buffer:

```
.data
char BYTE ?
.code
L1: mov  eax,10                ; create 10 ms delay
    call Delay
    call ReadKey               ; check for key
    jz   L1                    ; repeat if no key
    mov  char,AL               ; save the character
```

The foregoing code inserts a 10-millisecond delay in the loop to give MS-Windows time to process event messages. If you omit the delay, keystrokes may be ignored.

### ***Application: Sequential Search of an Array***

A common programming task is to search for values in an array that meet some criteria. For example, the following program looks for the first nonzero value in an array of 16-bit integers. If it finds one, it displays the value; otherwise, it displays a message stating that a nonzero value was not found:

```
; Scanning an Array                   (ArrayScan.asm)
; Scan an array for the first nonzero value.

INCLUDE Irvine32.inc

.data
intArray  SWORD  0,0,0,0,1,20,35,-12,66,4,0
;intArray SWORD  1,0,0,0              ; alternate test data
;intArray SWORD  0,0,0,0              ; alternate test data
;intArray SWORD  0,0,0,1              ; alternate test data
noneMsg  BYTE "A non-zero value was not found",0
```

This program contains alternate test data that are currently commented out. Uncomment each of these lines to test the program with different data configurations.

```
.code
main PROC
    mov   ebx,OFFSET intArray       ; point to the array
    mov   ecx,LENGTHOF intArray     ; loop counter

L1: cmp   WORD PTR [ebx],0          ; compare value to zero
    jnz   found                     ; found a value
    add   ebx,2                     ; point to next
    loop  L1                        ; continue the loop
    jmp   notFound                  ; none found

found:                              ; display the value
    movsx eax,WORD PTR[ebx]         ; sign-extend into EAX
    call  WriteInt
    jmp   quit

notFound:                           ; display "not found" message
    mov   edx,OFFSET noneMsg
    call  WriteString

quit:
    call Crlf
    exit
main ENDP
END main
```

### *Application: Simple String Encryption*

The XOR instruction has an interesting property. If an integer X is XORed with Y and the resulting value is XORed with Y again, the value produced is X:

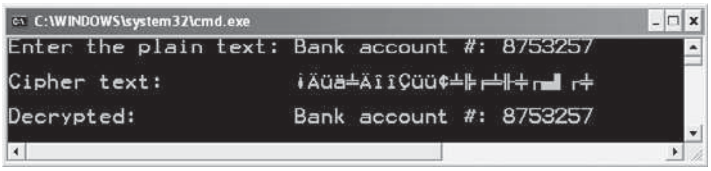
$$((X \otimes Y) \otimes Y) = X$$

This reversible property of XOR provides an easy way to perform a simple form of data encryption: A *plain text* message is transformed into an encrypted string called *cipher text by* XORing each of its characters with a character from a third string called a *key*. The intended viewer can use the key to decrypt the cipher text and produce the original plain text.

*Example Program* We will demonstrate a simple program that uses *symmetric encryption*, a process by which the same key is used for both encryption and decryption. The following steps occur in order at runtime:

1. The user enters the plain text.
2. The program uses a single-character key to encrypt the plain text, producing the cipher text, which is displayed on the screen.
3. The program decrypts the cipher text, producing and displaying the original plain text.

Here is sample output from the program:



```
Enter the plain text: Bank account #: 8753257
Cipher text:          ¡Äüä╧Äîî Çüü¢╧╟╤╫╤╓▄ ╓╤
Decrypted:            Bank account #: 8753257
```

*Program Listing* Here is a complete program listing:

```
; Encryption Program                    (Encrypt.asm)

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

.data
sPrompt  BYTE "Enter the plain text:",0
sEncrypt BYTE "Cipher text:     ",0
sDecrypt BYTE "Decrypted:       ",0
buffer   BYTE  BUFMAX+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                      ; save 32-bit registers
    mov   edx,OFFSET sPrompt    ; display a prompt
    call  WriteString
    mov   ecx,BUFMAX            ; maximum character count
    mov   edx,OFFSET buffer     ; point to the buffer
    call  ReadString            ; input the string
    mov   bufSize,eax           ; save the length
    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
```

You should never encrypt important data with a single-character encryption key, because it can be too easily decoded. Instead, the chapter exercises suggest that you use an encryption key containing multiple characters to encrypt and decrypt the plain text.

## 6.3.5 Section Review

1. Which jump instructions follow unsigned integer comparisons?
2. Which jump instructions follow signed integer comparisons?
3. Which conditional jump instruction is equivalent to JNAE?
4. Which conditional jump instruction is equivalent to the JNA instruction?
5. Which conditional jump instruction is equivalent to the JNGE instruction?
6. *(Yes/No):* Will the following code jump to the label named **Target**?

```
mov ax,8109h
cmp ax,26h
jg  Target
```

## 6.4 Conditional Loop Instructions

### 6.4.1 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. They perform the following tasks:

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

Otherwise, no jump occurs, and control passes to the next instruction. LOOPZ and LOOPE do not affect any of the status flags. In 32-bit mode, ECX is the loop counter register, and in 64-bit mode, RCX is the counter.

### 6.4.2 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. They perform the following tasks:

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

Otherwise, nothing happens, and control passes to the next instruction.

*Example* The following code excerpt (from *Loopnz.asm*) scans each number in an array until a nonnegative number is found (when the sign bit is clear). Notice that we push the flags on the stack before the ADD instruction because ADD will modify the flags. Then the flags are restored by POPFD just before the LOOPNZ instruction executes:

```
.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             ; move to next position
    popfd                             ; pop flags from stack
    loopnz  L1                        ; continue loop
```

```
        jnz    quit                        ; none found
        sub    esi,TYPE array              ; ESI points to value
    quit:
```

If a nonnegative value is found, ESI is left pointing at it. If the loop fails to find a positive number, it stops when ECX equals zero. In that case, the JNZ instruction jumps to label **quit**, and ESI points to the sentinel value (0), located in memory immediately following the array.

### 6.4.3 Section Review

1. *(True/False):* The LOOPE instruction jumps to a label when (and only when) the Zero flag is clear.
2. *(True/False):* In 32-bit mode, the LOOPNZ instruction jumps to a label when ECX is greater than zero and the Zero flag is clear.
3. *(True/False):* The destination label of a LOOPZ instruction must be no farther than −128 or +127 bytes from the instruction immediately following LOOPZ.
4. Modify the LOOPNZ example in Section 6.4.2 so that it scans for the first negative value in the array. Change the array initializers so they begin with positive values.
5. *Challenge:* The LOOPNZ example in Section 6.4.2 relies on a sentinel value to handle the possibility that a positive value might not be found. What might happen if you removed the sentinel?

## 6.5 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. No doubt you have already used conditional structures in a high-level programming language. But you may not know how language compilers translate conditional structures into low-level machine code. Let's find out how that is done.

### 6.5.1 Block-Structured IF Statements

An IF structure impli that a boolean expression is followed by two lists of statements; one performed when the expression is true, and another performed when the expression is false:

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

The **else** portion of the statement is optional. In assembly language, we code this structure in steps. First, we evaluate the boolean expression in such a way that one of the CPU status flags is affected. Second, we construct a series of jumps that transfer control to the two lists of statements, based on the value of the relevant CPU status flag.

*Example 1* In the following C++ code, two assignment statements are executed if **op1** is equal to **op2**:

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

We translate this IF statement into assembly language with a CMP instruction followed by conditional jumps. Because **op1** and **op2** are memory operands (variables), one of them must be moved to a register before executing CMP. The following code implements the IF statement as efficiently as possible by allowing the code to “fall through” to the two MOV instructions that we want to execute when the boolean condition is true:

```
      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 we implemented the == operator using JE, the resulting code would be slightly less compact (six instructions rather than five):

```
      mov   eax,op1
      cmp   eax,op2                 ; op1 == op2?
      je    L1                      ; yes: jump to L1
      jmp   L2                      ; no: skip assignments
  L1: mov   X,1                     ; assign X and Y
      mov   Y,2
  L2:
```

As you see from the foregoing example, the same conditional structure can be translated into assembly language in multiple ways. When examples of compiled code are shown in this chapter, they represent only what a hypothetical compiler might produce.

*Example 2* In the NTFS file storage system, the size of a disk cluster depends on the disk volume’s overall capacity. In the following pseudocode, we set the cluster size to 4,096 if the volume size (in the variable named **terrabytes**) is less than 16 TBytes. Otherwise, we set the cluster size to 8,192:

```
clusterSize = 8192;
if terrabytes < 16
  clusterSize = 4096;
```

Here’s a way to implement the pseudocode in assembly language:

```
      mov    clusterSize,8192    ; assume larger cluster
      cmp    terrabytes, 16      ; smaller than 16 TB?
      jae    next
      mov    clusterSize,4096    ; switch to smaller cluster
  next:
```

*Example 3* The following pseudocode statement has two branches:

```
if op1 > op2
     call Routine1
else
     call Routine2
end if
```

In the following assembly language translation of the pseudocode, we assume that **op1** and **op2** are signed doubleword variables. When comparing variables, one must be moved to a register:

```
     mov  eax,op1                 ; move op1 to a register
     cmp  eax,op2                 ; op1 > op2?
     jg   A1                      ; yes: call Routine1
     call Routine2                ; no: call Routine2
     jmp  A2                      ; exit the IF statement
A1:  call Routine1
A2:
```

### *White Box Testing*

Complex conditional statements may have multiple execution paths, making them hard to debug by inspection (looking at the code). Programmers often implement a technique known as *white box testing*, which verifies a subroutine's inputs and corresponding outputs. White box testing requires you to have a copy of the source code. You assign a variety of values to the input variables. For each combination of inputs, you manually trace through the source code and verify the execution path and outputs produced by the subroutine. Let's see how this is done in assembly language by implementing the following nested-IF statement:

```
if op1 == op2
  if X > Y
    call Routine1
  else
    call Routine2
  end if
else
  call Routine3
end if
```

Following is a possible translation to assembly language, with line numbers added for reference. It reverses the initial condition (op1 == op2) and immediately jumps to the ELSE portion. All that is left to translate is the inner IF-ELSE statement:

```
1:       mov   eax,op1
2:       cmp   eax,op2                  ; op1 == op2?
3:       jne   L2                       ; no: call Routine3

; process the inner IF-ELSE statement.
4:       mov   eax,X
5:       cmp   eax,Y                    ; X > Y?
6:       jg    L1                       ; yes: call Routine1
7:       call  Routine2                 ; no: call Routine2
8:       jmp   L3                       ; and exit
9:   L1: call  Routine1                 ; call Routine1
10:      jmp   L3                       ; and exit
11:  L2: call  Routine3
12:  L3:
```

Table 6-6 shows the results of white box testing of the sample code. In the first four columns, test values have been assigned to op1, op2, X, and Y. The resulting execution paths are verified in columns 5 and 6.

Table 6-6 Testing the Nested IF Statement.

| op1 | op2 | X | Y | Line Execution Sequence | Calls |
|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 1, 2, 3, 11, 12 | Routine3 |
| 10 | 20 | 40 | 30 | 1, 2, 3, 11, 12 | Routine3 |
| 10 | 10 | 30 | 40 | 1, 2, 3, 4, 5, 6, 7, 8, 12 | Routine2 |
| 10 | 10 | 40 | 30 | 1, 2, 3, 4, 5, 6, 9, 10, 12 | Routine1 |

### 6.5.2 Compound Expressions

#### *Logical AND Operator*

Assembly language easily implements compound boolean expressions containing AND operators. Consider the following pseudocode, in which the values being compared are assumed to be unsigned integers:

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

*Short-Circuit Evaluation* The following is a straightforward implementation using *short-circuit* evaluation, in which the second expression is not evaluated if the first expression is false. This is the norm for high-level languages:

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

We can reduce the code to five instructions by changing the initial JA instruction to JBE:

```
    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:
```

The 29% reduction in code size (seven instructions down to five) results from letting the CPU fall through to the second CMP instruction if the first JBE is not taken.

### *Logical OR Operator*

When a compound expression contains subexpressions joined by the OR operator, the overall expression is true if any of the subexpressions is true. Let's use the following pseudocode as an example:

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

In the following implementation, the code branches to L1 if the first expression is true; otherwise, it falls through to the second CMP instruction. The second expression reverses the > operator and uses JBE instead:

```
     cmp   al,bl                 ; 1: compare AL to BL
     ja    L1                    ; if true, skip second expression
     cmp   bl,cl                 ; 2: compare BL to CL
     jbe   next                  ; false: skip next statement
L1:  mov   X,1                   ; true: set X = 1
next:
```

For a given compound expression, there are multiple ways the expression can be implemented in assembly language.

## 6.5.3 WHILE Loops

A WHILE loop tests a condition first before performing a block of statements. As long as the loop condition remains true, the statements are repeated. The following loop is written in C++:

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

When implementing this structure in assembly language, it is convenient to reverse the loop condition and jump to **endwhile** if a condition becomes true. Assuming that **val1** and **val2** are variables, we must copy one of them to a register at the beginning and restore the variable's value at the end:

```
     mov   eax,val1              ; copy variable to EAX
beginwhile:
     cmp   eax,val2              ; if not (val1 < val2)
     jnl   endwhile              ;   exit the loop
     inc   eax                   ; val1++;
     dec   val2                  ; val2--;
     jmp   beginwhile            ; repeat the loop
endwhile:
     mov   val1,eax              ; save new value for val1
```

EAX is a proxy (substitute) for **val1** inside the loop. References to **val1** must be through EAX. JNL is used, implying that **val1** and **val2** are signed integers.

### *Example: IF statement Nested in a Loop*

High-level languages are particularly good at representing nested control structures. In the following C++ code, an IF statement is nested inside a WHILE loop. It calculates the sum of all array elements greater than the value in **sample**:

```
int array[] = {10,60,20,33,72,89,45,65,72,18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;
while( index < ArraySize )
{
   if( array[index] > sample )
   {
      sum += array[index];
   }
   index++;
}
```

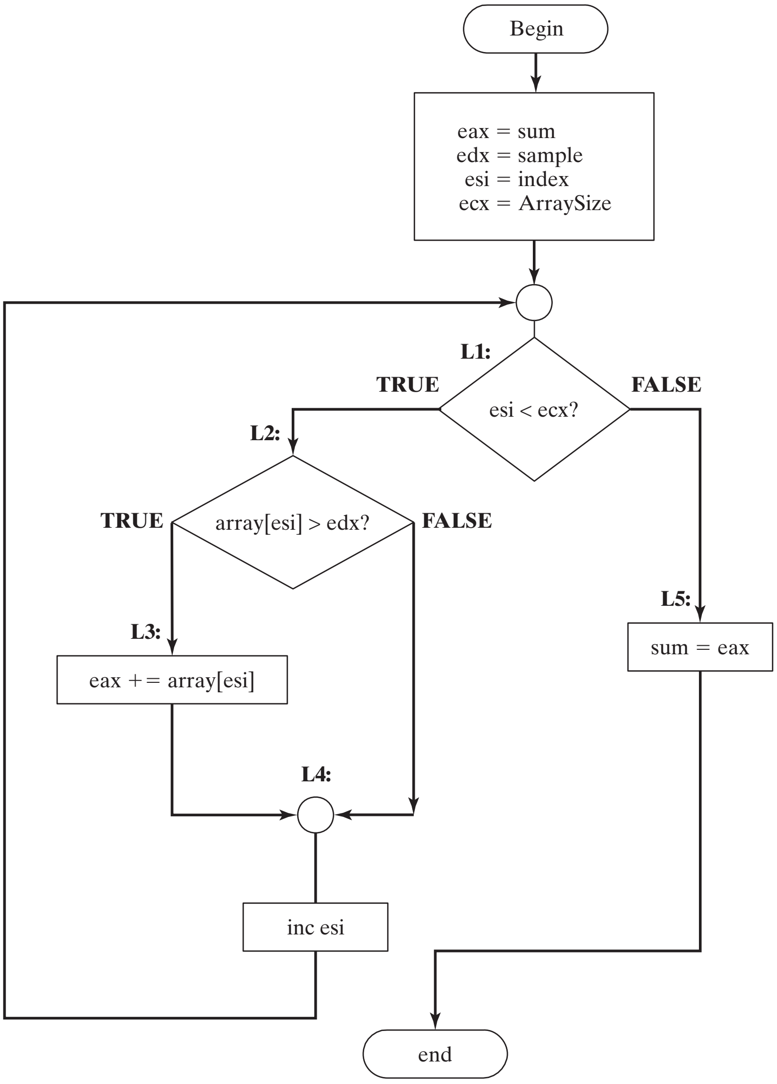
Before coding this loop in assembly language, let's use the flowchart in Fig. 6-1 to describe the logic. To simplify the translation and speed up execution by reducing the number of memory accesses, registers have been substituted for variables. EDX = sample, EAX = sum, ESI = index, and ECX = ArraySize (a constant). Label names have been added to the shapes.

*Assembly Code* The easiest way to generate assembly code from a flowchart is to implement separate code for each flowchart shape. Note the direct correlation between the flowchart labels and labels used in the following source code (see *Flowchart.asm*):

```
.data
sum DWORD 0
sample DWORD 50
array DWORD 10,60,20,33,72,89,45,65,72,18
ArraySize = ($ - Array) / TYPE array

.code
main PROC
     mov   eax,0                 ; sum
     mov   edx,sample
     mov   esi,0                 ; index
     mov   ecx,ArraySize

L1:  cmp   esi,ecx               ; if esi < ecx
     jl    L2
     jmp   L5

L2:  cmp   array[esi*4], edx     ; if array[esi] > edx
     jg    L3
     jmp   L4
L3:  add   eax,array[esi*4]

L4:  inc   esi
     jmp   L1

L5:  mov   sum,eax
```

A review question at the end of Section 6.5 will give you a chance to improve this code.

FIGURE 6–1 Loop containing IF statement.



## 6.5.4 Table-Driven Selection

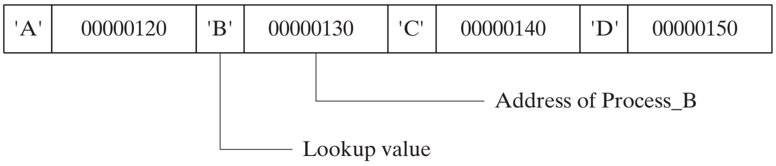
*Table-driven selection* is a way of using a table lookup to replace a multiway selection structure. To use it, you must create a table containing lookup values and the offsets of labels or procedures, and then you must use a loop to search the table. This works best when a large number of comparisons are made.

For example, the following is part of a table containing single-character lookup values and addresses of procedures:

```
.data
CaseTable BYTE   'A'          ; lookup value
    DWORD Process_A           ; address of procedure
    BYTE  'B'
    DWORD Process_B
     (etc.)
```

Let's assume Process_A, Process_B, Process_C, and Process_D are located at addresses 120h, 130h, 140h, and 150h, respectively. The table would be arranged in memory as shown in Fig. 6–2.

Figure 6–2 Table of procedure offsets.



*Example Program* In the following example program (*ProcTable.asm*), the user inputs a character from the keyboard. Using a loop, the character is compared to each entry in a lookup table. The first match found in the table causes a call to the procedure offset stored immediately after the lookup value. Each procedure loads EDX with the offset of a different string, which is displayed during the loop:

```
; Table of Procedure Offsets             (ProcTable.asm)

; This program contains a table with offsets of procedures.
; It uses the table to execute indirect procedure calls.

INCLUDE Irvine32.inc
.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
prompt BYTE "Press capital A,B,C,or D: ",0
```

Define a separate message string for each procedure:

```
msgA BYTE "Process_A",0
msgB BYTE "Process_B",0
msgC BYTE "Process_C",0
msgD BYTE "Process_D",0
```

```
.code
main PROC
     mov   edx,OFFSET prompt           ; ask user for input
     call  WriteString
     call  ReadChar                    ; read character into AL
     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
```

This CALL instruction calls the procedure whose address is stored in the memory location referenced by EBX+1. An indirect call such as this requires the NEAR PTR operator.

```
     call  WriteString                 ; display message
     call  Crlf
     jmp   L3                          ; exit the search
L2:
     add   ebx,EntrySize               ; point to the next entry
     loop  L1                          ; repeat until ECX = 0

L3:
     exit
main ENDP
```

Each of the following procedures moves a different string offset to EDX:

```
Process_A PROC
     mov   edx,OFFSET msgA
     ret
Process_A ENDP

Process_B PROC
     mov   edx,OFFSET msgB
     ret
Process_B ENDP

Process_C PROC
     mov   edx,OFFSET msgC
     ret
Process_C ENDP

Process_D PROC
     mov   edx,OFFSET msgD
     ret
Process_D ENDP
END main
```

The table-driven selection method involves some initial overhead, but it can reduce the amount of code you write. A table can handle a large number of comparisons, and it can be more easily modified than a long series of compare, jump, and CALL instructions. A table can even be reconfigured at runtime.