Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Lecture 22: Conditional Loops

Slides prepared by Kip R. Irvine

Revision date: 07/11/2002

Modified by Dr. Nikolay Metodiev Sirakov Fall 2012

- Chapter corrections (Web) Assembly language sources (Web)

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:
  - LOOPE *destination*
  - LOOPZ *destination*
- Logic:
  - ECX $\leftarrow$ ECX – 1
  - if ECX > 0 and ZF=1, jump to *destination*
- Useful when scanning an array for the first element that does not match a given value.

LOOPNZ and LOOPNE

- LOOPNZ (LOOPNE) is a conditional loop instruction
- Syntax:

  LOOPNZ *destination*

  LOOPNE *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.

LOOPNZ Example

The following code finds the first positive value in an array:

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

Your turn . . .

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0          ; check for zero


     (fill in your code here)



quit:
```

. . . (solution)

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0          ; check for zero
    pushfd                         ; push flags on stack
    add esi,TYPE array
    popfd                          ; pop flags from stack
    loope L1                       ; continue loop
    jz quit                        ; none found
    sub esi,TYPE array             ; ESI points to value
quit:
```

Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection

Block-Structured IF Statements

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

```
if( op1 == op2 )
  body1;
else
  body2;
```

```
    mov eax,op1
    cmp eax,op2
    jne L1
    body1
    jmp L2
L1: body2
L2:
```

Your turn . . .

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

```
if( ebx <= ecx )
{
  body
}
```

```
    cmp ebx,ecx
    ja  next
    body
next:
```

(There are multiple correct solutions to this problem.)

Your turn . . .

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 Expression with AND (1 of 3)

- We can implement a Boolean expression that uses the local AND operator:
- In the following example, if the first expression is false, the second expression is skipped;
- The Assembler will execute the body if both conditions are satisfied.

```
if (al > bl) AND (bl > cl)
{
  body
}
```

Compound Expression with AND (2 of 3)

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

This is 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:
```

Compound Expression with AND (3 of 3)

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

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:

```
    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:
```

Your turn . . .

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

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

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

(There are multiple correct solutions to this problem.)

Compound Expression with OR (1 of 2)

- We can implement a Boolean expression that uses the local OR operator
- In the following example, if the first expression is true, the second expression is skipped;
- The Assembler will execute the body if at least one condition is satisfied.

```
if (al > bl) OR (bl > cl)
  body;
```

Compound Expression with OR (1 of 2)

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

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
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. 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:
```

Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
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:
```

Table-Driven Selection (1 of 3)

- Table-driven selection uses a table lookup to replace a multi-way 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

Table-Driven Selection (2 of 3)

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
```

Table-Driven Selection (3 of 3)

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:
```

required for
procedure pointers