Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE
LOOPZ and LOOPE

• Syntax:
  LOOPE destination
  LOOPZ destination

• Logic:
  • ECX ← 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.
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:

```assembly
.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:
```


```assembly
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFFFh
.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 ; pop flags from stack
    popfd ; continue loop
    loope next ; none found
    jz quit ; ESI points to value
    sub esi,TYPE array
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 )
  X = 1;
else
  X = 2;
```

```assembly
mov eax,op1
cmp eax,op2
jne L1
mov X,1
jmp L2
L1: mov X,2
L2:
```
Your turn . . .

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

```assembly
if( ebx <= ecx )
{
    eax = 5;
    edx = 6;
}
```
Your turn . . .

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

```plaintext
if( var1 <= var2 )
    var3 = 10;
else
{
    var3 = 6;
    var4 = 7;
}
```
Compound Expression with AND [1/3]

- When implementing the logical AND operator, consider that HLLs use short-circuit evaluation.
- In the following example, if the first expression is false, the second expression is skipped:

```plaintext
if (al > bl) AND (bl > cl)
    X = 1;
```
Compound Expression with AND [2/3]

\[
\text{if (al > bl) AND (bl > cl)} \\
\text{X = 1;}
\]

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:
    mov X, 1       ; both are true
next:
```

Compound Expression with AND [3/3]

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

But the following implementation uses 29% less code by reversing the first relational operator.

```
cmp al,bl
jbe next

cmp bl,cl
jbe next

mov X,1

next:
```

Your turn . . .

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

```assembly
if(ebx <= ecx) AND (ecx > edx )
{
    eax = 5;
    edx = 6;
}
```
Compound Expression with OR [1/2]

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation.
- In the following example, if the first expression is true, the second expression is skipped:

```c
if (al > bl) OR (bl > cl)  
X = 1;
```
We can use "fall-through" logic to keep the code as short as possible:

```assembly
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:

```assembly
while( eax < ebx)
    eax = eax + 1;
```

This is a possible implementation:

```assembly
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:

```assembly
while( ebx <= val1 )
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```
Table-Driven Selection [1/3]

- Create a table containing lookup values and the offsets of labels or procedures
- Use a loop to search the table
Table-Driven Selection [2/3]

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 = 4
```
Table-Driven Selection [3/3]

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:  
```

Application: Finite-State Machines

- A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a state-transition diagram.
- We 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)
Finite-State Machine

- 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
FSM Examples

• FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':

• FSM that recognizes signed integers:
Your turn . . .

• Explain why the following FSM does not work as well for signed integers as the one shown on the previous slide:
Implementing an FSM

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

StateA:
    call Getnext ; read next char into AL
    cmp al, '+'
    je StateB
    cmp al, '-'
    je StateB
    call IsDigit ; ZF = 1 if AL = digit
    jz StateC
    call DisplayErrorMsg ; invalid input found
    jmp Quit

View the Finite.asm source code.
Flowchart of State A

State A accepts a plus or minus sign, or a decimal digit.
Runtime Expressions

- IF, ELSE, ELSEIF, and ENDIF can be used to create block-structured IF statements.

- Examples:

```assembly
.IF eax > ebx
    mov edx, 1
.ELSE
    mov edx, 2
.ENDIF

.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.
## Relational and Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr1 == expr2</code></td>
<td>Returns true when <code>expression1</code> is equal to <code>expr2</code>.</td>
</tr>
<tr>
<td><code>expr1 != expr2</code></td>
<td>Returns true when <code>expr1</code> is not equal to <code>expr2</code>.</td>
</tr>
<tr>
<td><code>expr1 &gt; expr2</code></td>
<td>Returns true when <code>expr1</code> is greater than <code>expr2</code>.</td>
</tr>
<tr>
<td><code>expr1 &gt;= expr2</code></td>
<td>Returns true when <code>expr1</code> is greater than or equal to <code>expr2</code>.</td>
</tr>
<tr>
<td><code>expr1 &lt; expr2</code></td>
<td>Returns true when <code>expr1</code> is less than <code>expr2</code>.</td>
</tr>
<tr>
<td><code>expr1 &lt;= expr2</code></td>
<td>Returns true when <code>expr1</code> is less than or equal to <code>expr2</code>.</td>
</tr>
<tr>
<td><code>! expr</code></td>
<td>Returns true when <code>expr</code> is false.</td>
</tr>
<tr>
<td><code>expr1 &amp;&amp; expr2</code></td>
<td>Performs logical AND between <code>expr1</code> and <code>expr2</code>.</td>
</tr>
<tr>
<td>`expr1</td>
<td></td>
</tr>
<tr>
<td><code>expr1 &amp; expr2</code></td>
<td>Performs bitwise AND between <code>expr1</code> and <code>expr2</code>.</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>
MASM-Generated Code

.data
val1 DWORD 5
result DWORD ?
.code
mov eax,6
mov result,1
.ENDIF

Generated code:

Generated code:

mov eax,6
cmp eax, val1
jbe @C0001
mov result, 1
@C0001:

MASM automatically generates an unsigned jump (JBE).
MASM automatically generates a signed jump (JLE).
.REPEAT Directive

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

Example:

```assembly
; 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