Lab 8 - Conditional Loop Instructions

8.1 Code Labels

A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named target, creating a loop:

```
target:
	mov ax,bx
	...
	...
	jmp target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1:	mov ax,bx
L2:
```

Label names are created using the rules for identifiers discussed in previous Labs. You can use the same code label more than once in a program if each label is unique within its enclosing procedure. (A procedure is like a function.)

```
Include Irvine32.inc

.data
	msg1 byte "Hello",0
	msg2 byte "World",0
	msg3 byte "Exit",0

.code
main proc
		mov edx, offset msg1
		call writestring
		jmp M3
	M2:
		mov edx, offset msg2
		call writestring
		jmp M4
	M3:
		mov edx, offset msg3
		call writestring
		jmp M2
	M4:
	call readint
	exit
main endp
end main
```

8.2 Loop Instruction

Loop instruction uses **ECX** register as iterator. Loop instruction keeps on executing a specific label until the value of ECX register become zero. With each iteration, value of ECX get decremented automatically.

```
Loop Label
```

Following example prints “Hello” five times:

```
Include Irvine32.inc
.data
	msg1 byte "Hello",0
.code
main proc
	mov ecx, 5
	M1:
		mov edx, offset msg1
		call writestring
		loop M1

	call readint
	exit
main endp
end main
```

8.3 Nested Loops

When dealing with nested loops, ECX Is used for both loops: inner and outer. Therefore, it is mandatory to store the status of outer loop before entering in inner and restore the status after exiting.

```
INCLUDE Irvine32.inc
.data
	count DWORD ?
	prompt BYTE 0dh,0ah,"Hello",0
	prompt1 BYTE 0dh,0ah,"World",0
.code
main PROC
	mov ecx, 5
	Loop1:
		mov edx, OFFSET prompt
		call WriteString
		mov count, ecx
		mov ecx, 3
		Loop2:
			mov edx, OFFSET prompt1
			call WriteString
		LOOP Loop2
		mov ecx, count
	LOOP Loop1

	call readInt
	exit
main ENDP
END main
```

8.4 Additional Conditional Loops

These loop statements also work using ECX value, additionally they have following condition to meet in order to execute.

```
LOOPZ                   ;loop if Zero flag = 1
LOOPE                   ;loop if (ECX > 0 and ZF = 1)
LOOPNZ                  ;loop if Zero flag = 0
LOOPNE                  ;loop if (ECX > 0 and ZF = 0)
```

8.4.1 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 to transfer the control 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.

Flags: 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.

8.4.2 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 example code 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:

```
Include Irvine32.inc
.data
	array SWORD -3,-6,-1,-10,10,30,40,4
.code
main PROC
	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 index of array
	popfd					;pop flags from stack
	loopnz L1				;continue loop
	jnz quit				;none found
	sub esi,TYPE array			;ESI points to value
	movsx EAX, WORD PTR [esi]
	call Writeint
quit:
	exit
main ENDP
end main
```

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.

8.5 Block-Structured IF Statements

An IF structure imply 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.

```
        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:
```

Example 2:

In the NTFS file storage system, the size of a disk cluster depends on the disk volume's overall capacity.

```
        clusterSize = 8192;
        if (volumeSize < 16)      //16 TeraByte
        {
              clusterSize = 4096;
        }
```

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

```
        mov clusterSize,8192                ;assume larger cluster
        cmp volumeSize, 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:
```

Example 4 - 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.

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

Let's see how this is done in assembly language by implementing the following nested-IF statement:

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

      ; process the inner IF-ELSE statement.
      mov eax,X
      cmp eax,Y                     ;X > Y?
      jg L1                         ;yes: call Routine1
      call Routine2                 ;no: call Routine2
      jmp L3                        ;and exit
L1:   call Routine1                 ;call Routine1
      jmp L3                        ;and exit
L2:   call Routine3
L3:
```

The below table shows the results of white box testing of the sample code.

op1	op1	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

8.6 Compound Expressions

8.6.1 Short-Circuit Evaluation

Short-circuiting is a programming concept in which the compiler skips the execution or evaluation of some sub expressions in a logical expression. The compiler stops evaluating the further sub-expressions as soon as the value of the expression is determined. Below is an example of the same:

```
if (a == b || c == d || e == f)
{     // do_something     }
```

In the above expression, If the expression **a == b** is **true**, then the **c == d** and **e == f** expressions are not evaluated because the expression's result has already been determined. Similarly, if the logical **AND** operator is used instead of logical **OR** and the expression **a == b** is **false**, the compiler will skip evaluating other sub expressions.

8.6.2 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
```

Using short-circuit evaluation, the assembly code will be:

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

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

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

8.6.3 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
end if
```

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 quit                     ;false: skip next statement
L1:   mov x,1                      ;true: set X = 1
quit:
```

8.7 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 exit from the loop 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:

```
Include Irvine32.inc
.data
	val1 DWORD 5
	val2 DWORD 9
.code
main PROC
	mov eax,val1				;copy variable to EAX
	beginwhile:
		cmp eax,val2			;if not (val1 < val2)
		jnl endwhile			;jump if not less (exit he loop)
		inc eax				;val1++
		dec val2			;val2--
		jmp beginwhile			;repeat the loop
	endwhile:
	mov val1,eax				;save new value for val1

	exit
main ENDP
end main
```

8.7.1 Example: IF statement Nested in a Loop

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++;
}
```

Assembly Code of the above C++ code is:

```
Include Irvine32.inc
.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
	Call Writeint

exit
main ENDP
end main
```

8.8 Conditional Control Flow Directives

In 32-bit mode, MASM includes a number of high-level conditional control flow directives that help to simplify the coding of conditional statements. Unfortunately, they cannot be used in 64-bit mode. Before assembling your code, the assembler performs a preprocessing step. In this step, it recognizes directives such as .CODE, .DATA, as well as directives that can be used for conditional control flow. The below table lists the directives:

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF *condition*	Generates code that tests *condition* and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF *condition*	Generates code that executes the block of statements if condition is true.
.REPEAT	Generates code that repeats execution of the block of statements until condition becomes true.
.UNTIL *condition*	Generates code that repeats the block of statements between .REPEAT and .UNTIL until condition becomes true.
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until CX equals zero.
.WHILE *condition*	Generates code that executes the block of statements between .WHILE and .ENDW as long as *condition* is true

8.8.1 Creating IF Statements

The .IF, .ELSE, .ELSEIF, and .ENDIF directives make it easy for you to code multiway branching logic. They cause the assembler to generate CMP and conditional jump instructions in the background, which appear in the output listing file (*progname*.lst). This is the syntax:

```
.IF condition1
      statements
[.ELSEIF condition2
      statements ]
[.ELSE
      statements ]
.ENDIF
```

The square brackets show that **.ELSEIF and .ELSE** are optional, whereas **.IF** and **.ENDIF** are required.

A condition is a Boolean expression involving the same operators used in C++. The expression is evaluated at runtime.

The following are some examples of valid conditions, using 32-bit registers and variables:

```
eax > 10000h
val1 <= 100
val2 == eax
val3 != ebx
```

The following are some examples of compound conditions:

```
(eax > 0) && (eax > 10000h)
(val1 <= 100) || (val2 <= 100)
(val2 != ebx) && !CARRY?
```

A complete list of relational and logical operators is shown in table below:

Operator	Description
expr1 == *expr2*	Returns true when *expr1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! expr	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

When you use high-level directives such as **.IF** and **.ELSE**, the assembler writes code for you. For example, let's write an **.IF** directive that compares EAX to the variable **val1**:

```
mov eax,6
.IF eax > val1
      mov result,1
.ENDIF
```

val1 and **result** are assumed to be 32-bit unsigned integers. When the assembler reads the above lines, it expands them into the following assembly language instructions, which you can view if you run the program in the Visual Studio debugger, right-click, and select *Go to Disassembly*.

```
      mov eax,6
      cmp eax,val1
      jbe @C0001                    ;jump on unsigned comparison
      mov result,1
@C0001:
```

The label name @C0001 was created by the assembler. This is done in a way that guarantees that all labels within same procedure are unique.

Example:

```
Include Irvine32.inc
.data
	var1 DWORD 4
	var2 DWORD 6

	message1 BYTE "var1 is greater than var2",0
	message2 BYTE "var1 is less than var2",0

.code
main PROC
	mov EAX,var1
	.IF EAX > var2
		mov EDX,OFFSET message1
		call WriteString
	.ELSEIF EAX < var2
		mov EDX,OFFSET message2
		call WriteString
	.ENDIF
exit
main ENDP
end main
```

8.8.2 Creating Loops with .REPEAT and .WHILE

The **.REPEAT** and **.WHILE** directives offer alternatives to writing your own loops with CMP and conditional jump instructions. The **.REPEAT** directive executes the loop body before testing the runtime condition following the **.UNTIL** directive:

```
.REPEAT
	statements
.UNTIL condition
```

The **.WHILE** directive tests the condition before executing the loop:

```
.WHILE condition
	statements
.ENDW
```

The following example display the values 1 through 10 using the .WHILE directive. The counter register (EAX) is initialized to zero before the loop. Then, in the first statement inside the loop, EAX is incremented. The .WHILE directive branches out of the loop when EAX equals 10.

```
mov eax,0
.WHILE eax < 10
	inc eax
	call WriteDec
	call Crlf
.ENDW
```

The following statements display the values 1 through 10 using the .REPEAT directive:

```
mov eax,0
.REPEAT
	inc eax
	call WriteDec
```

```
        call Crlf
.UNTIL eax == 10
```

8.8.3 Loop Containing an IF Statement

Previously we learned how to write assembly language code for an IF statement nested inside a WHILE loop. Here is the pseudocode:

```
while( op1 < op2 )
{
        op1++;
        if( op1 == op3 )
                X = 2;
        else
                X = 3;
}
```

The following is an implementation of the pseudocode using the **.WHILE** and **.IF** directives. Because **op1**, **op2**, and **op3** are variables, they are moved to registers to avoid having two memory operands in any one instruction:

```
Include Irvine32.inc
.data
        X DWORD 0
        op1 DWORD 2 ; test data
        op2 DWORD 6 ; test data
        op3 DWORD 5 ; test data
.code
main PROC
        mov eax,op1
        mov ebx,op2
        mov ecx,op3
        .WHILE eax < ebx
                inc eax
                .IF eax == ecx
                        mov X,2
                .ELSE
                        mov X,3
                .ENDIF
        .ENDW
exit
main ENDP
end main
```

8.9 Signed and Unsigned Comparisons

When you use the .IF directive to compare values, you must be aware of how MASM generates conditional jumps.

- If the comparison involves an unsigned variable, an unsigned conditional jump instruction is inserted in the generated code.
- If the comparison involves a signed variable, a signed conditional jump instruction is inserted in the generated code.

For example:

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

The assembler expands this using the **JBE** (unsigned jump) instruction:

```
	mov eax,6
	cmp eax,val1
	jbe @C0001		;jump on unsigned comparison
	mov result,1
@C0001:
```

8.9.1 Comparing a Signed Integer

If an .IF directive compares a signed variable, however, a signed conditional jump instruction is inserted into the generated code. For example:

```
.data
	val2 SDWORD -1
	result DWORD ?
.code
	mov eax,6
	.IF eax > val2
		mov result,1
	.ENDIF
```

The assembler generates code using the **JLE** instruction, a jump based on signed comparisons:

```
	mov eax,6
	cmp eax,val2
	jle @C0001 ; jump on signed comparison
	mov result,1
@C0001:
```

8.9.2 Comparing Registers

What happens if two registers are compared? Clearly, the assembler cannot determine whether the values are signed or unsigned:

```
	mov eax,6
	mov ebx,val2
	.IF eax > ebx
		mov result,1
	.ENDIF
```

The following code is generated, showing that the assembler defaults to an unsigned comparison (using **JBE** instruction):

```
	mov eax,6
	mov ebx,val2
	cmp eax, ebx
```

```
      jbe @C0001
      mov result,1
@C0001:
```

8.9.3 Compound Expressions

Many compound boolean expressions use the logical **OR** and **AND** operators. When using the .IF directive, the **||** symbol is the logical **OR** operator:

```
      .IF expression1 || expression2
            statements
      .ENDIF
```

Similarly, the **&&** symbol is the logical **AND** operator:

```
      .IF expression1 && expression2
            statements
      .ENDIF
```

8.9.4 SetCursorPosition Example:

```
      Include Irvine32.inc

      .code
      main PROC
            mov dl,79                  ; X-coordinate
            mov dh,24                  ; Y-coordinate
            call SetCursorPosition
            exit
      main ENDP


      SetCursorPosition PROC
      ; Set the cursor position.
      ; Receives: DL = X-coordinate, DH = Y-coordinate
      ; Checks the ranges of DL and DH.
      ;-----------------------------------------------
      .data
            BadXCoordMsg BYTE "X-Coordinate out of range!",0Dh,0Ah,0
            BadYCoordMsg BYTE "Y-Coordinate out of range!",0Dh,0Ah,0
      .code
            .IF (DL < 0) || (DL > 79)
               mov  edx,OFFSET BadXCoordMsg
               call WriteString
               jmp  quit
            .ENDIF
            .IF (DH < 0) || (DH > 24)
               mov  edx,OFFSET BadYCoordMsg
               call WriteString
               jmp  quit
            .ENDIF
            call Gotoxy
      quit:
            ret
      SetCursorPosition ENDP
      END main
```

8.9.5 College Registration Example:

```
Include Irvine32.inc
.data
       TRUE = 1
          FALSE = 0
          gradeAverage  WORD 275    ; test value
          credits       WORD 12     ; test value
          OkToRegister  BYTE ?
.code
main PROC
          mov OkToRegister,FALSE
          .IF gradeAverage > 350
                mov OkToRegister,TRUE
          .ELSEIF (gradeAverage > 250) && (credits <= 16)
                mov OkToRegister,TRUE
          .ELSEIF (credits <= 12)
                mov OkToRegister,TRUE
          .ENDIF
   exit
main ENDP
END main
```