Lab 6: Conditional Processing

Contents

6.1 Unconditional Jump

The unconditional jump instruction (**`jmp`**) unconditionally transfers control to the instruction located at the target address. The general format is:

```
jmp target
```

There are two ways to specify the target address of the **jmp** instruction: *direct* and *indirect*. The most common one is the direct jump. Indirect jumps will be discussed in a later section.

In *direct jumps*, the target instruction address is specified directly as part of the instruction. As a programmer, you only specify the target address by using a *label* and let the assembler figure out the jump value of the target instruction. In the following example, **jmp L1** and **jmp L2** are direct jumps. The first jump **jmp L1** is a *forward jump*, while **jmp L2** is a *backward jump* because the target instruction precedes the jump instruction.

```
L2:
  . . .
  jmp L1
  . . .
L1:
  . . .
  jmp L2
```

6.1.1 Relative Displacement

The address that is specified in a jump instruction is not the absolute address of the target instruction. Rather, a **relative displacement** is stored in the jump instruction. The relative displacement is the number of bytes between the target instruction and the instruction following the jump instruction. Recall that **EIP** register points at the next instruction to be executed. Therefore, after fetching a jump instruction, **EIP** is advanced to point at the next instruction. When the processor executes the jump, it simply performs the following action:

EIP = EIP + relative-displacement

The relative displacement is a signed number stored inside the jump instruction itself. If the number is *positive* then it is a *forward jump*. Otherwise, it is a *backward jump*.

In an *indirect jump*, the target address is specified indirectly through a register or memory. We will defer their discussion to Section 6.6.

6.2 The Compare Instruction

The compare (**CMP**) instruction performs an implied subtraction of a *source* operand from a *destination* operand. Neither operand is modified. It has the following format:

```
CMP destination, source
```

The *Overflow*, *Carry*, *Sign*, and *Zero* flags are updated as if the subtract instruction has been performed. The main purpose of the compare instruction is to update the flags so that a subsequent conditional jump instruction can test them.

6.2.1 Lab Work: Demonstrating the Compare Instruction

The following program demonstrates the compare instruction and the affected flags.

```
TITLE Demonstrating the Compare Instruction  (cmp.asm)

.686
.MODEL flat, stdcall
.STACK
INCLUDE Irvine32.inc

.data
var1    SDWORD  -3056

.code
main PROC
    mov eax, 0f7893478h
    mov ebx,  1234F678h
    cmp al,  bl
    cmp ax,  bx
    cmp eax, ebx
    cmp eax, var1
    exit
main ENDP
END main
```

Carry the execution of the compare instructions manually by doing subtraction by hand. Guess the values of the flags and write them in the specified boxes.

`cmp al,  bl`	`OF =`	`CF =`	`SF =`	`ZF =`
`cmp ax,  bx`	`OF =`	`CF =`	`SF =`	`ZF =`
`cmp eax, ebx`	`OF =`	`CF =`	`SF =`	`ZF =`
`cmp eax, var1`	`OF =`	`CF =`	`SF =`	`ZF =`

6.2.2 Lab Work: Assemble and Link cmp.asm

6.2.3 Lab Work: Trace the Execution of Program cmp.exe

Run the 32-bit Windows Debugger, either from the **Tools** menu in the ConTEXT editor, or by typing: **windbg –QY –G cmp.exe** at the command prompt. Open the source file *cmp.asm* from the **File** menu if it is not already opened. Watch and customize the registers to have the **of cf sf zf** flags and the **eax ebx ax bx al** and **bl** registers on top of the list.

Place the cursor at the beginning of the *main* procedure and press **F7**. Now step through the program by pressing **F10** and watch the changes in the flags and registers. Observe that the compare instruction does not modify any operand. It only affects the flags. Check the flag answers that you wrote. Make the necessary corrections and understand your mistakes.

6.3 Conditional Jump Instructions

Conditional jump instructions can be divided into four groups:

- Jumps based on the value of a single arithmetic flag
- Jumps based on the value of CX or ECX
- Jumps based on comparisons of signed operands
- Jumps based on comparisons of unsigned operands

The following is a list of jumps based on the *Zero*, *Carry*, *Overflow*, *Sign*, and *Parity* flags.

Mnemonic	Description	Flags
JZ, JE	Jump if Zero, Jump if Equal	ZF = 1
JNZ, JNE	Jump if Not Zero, Jump if Not Equal	ZF = 0
JC	Jump if Carry	CF = 1
JNC	Jump if No Carry	CF = 0
JO	Jump if Overflow	OF = 1
JNO	Jump if No Overflow	OF = 0
JS	Jump if Signed (Negative)	SF = 1
JNS	Jump if Not Signed (Positive or Zero)	SF = 0
JP, JPE	Jump if Parity, Jump if Parity is Even	PF = 1
JNP, JPO	Jump if Not Parity, Jump if Parity is Odd	PF = 0

The following table shows the jumps based on the value of CX and ECX:

Mnemonic	Description
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

The following table shows a list of **signed jumps** based on comparisons of **signed** operands:

Mnemonic	Description	Condition Tested
JG, JNLE	Jump if Greater, Jump if Not Less or Equal	ZF = 0 and SF = OF
JGE, JNL	Jump if Greater or Equal, Jump if Not Less	SF = OF
JL, JNGE	Jump if Less, Jump if Not Greater or Equal	SF ≠ OF
JLE, JNG	Jump if Less or Equal, Jump if Not Greater	ZF = 1 or SF ≠ OF

The following shows a list of **unsigned jumps** based on comparisons of **unsigned** operands:

Mnemonic	Description	Condition Tested
JA, JNBE	Jump if Above, Jump if Not Below or Equal	ZF = 0 and CF = 0
JAE, JNB	Jump if Above or Equal, Jump if Not Below	CF = 0
JB, JNAE	Jump if Below, Jump if Not Above or Equal	CF = 1
JBE, JNA	Jump if Below or Equal, Jump if Not Above	ZF = 1 or CF = 1

6.4 Lab Work: Finding the Maximum of Three Integers

```
TITLE Finding the Maximum of 3 Integers  (max.asm)
.686
.MODEL flat, stdcall
.STACK
INCLUDE Irvine32.inc

.data
var1    DWORD  -30  ; Equal to FFFFFFE2 (hex)
var2    DWORD  12
var3    DWORD  7
max1    BYTE    "Maximum Signed Integer = ",0
max2    BYTE    "Maximum Unsigned Integer = ",0

.code
main PROC
    ; Finding Signed Maximum
    mov eax, var1
    cmp eax, var2
    jge L1
    mov eax, var2
L1:
    cmp eax, var3
    jge L2
    mov eax, var3
L2:
    lea  edx, max1
    call WriteString
    call WriteInt
    call Crlf

    ; Finding Unsigned Maximum
    mov eax, var1
    cmp eax, var2
    jae L3
    mov eax, var2
L3:
    cmp eax, var3
    jae L4
    mov eax, var3
L4:
    lea  edx, max2
    call WriteString
    call WriteHex
    call Crlf

    exit
main ENDP
END main
```

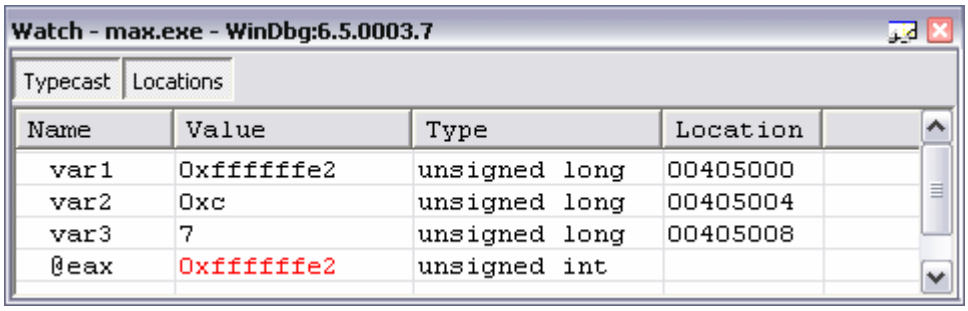
Analyze the above program and find its output:

Console Output

6.4.1 Lab Work: Assemble and Link max.asm

6.4.2 Lab Work: Trace Program max.exe

Run the 32-bit Windows Debugger. Open the source file *max.asm* from the **File** menu if it is not already opened. Add a watch to variables **var1**, **var2**, **var3**, and register **eax**. You can view a register in the Watch window by typing **@** before the register name as shown below.



Place the cursor at the beginning of the *main* procedure and press **F7**. Now step through the program by pressing **F10** and watch the changes in the **eax** register. Observe when conditional branches are taken (or not taken). Check the console output that you wrote, make the correction, and try to understand your mistakes.

6.5 Implementing High-Level Control Structures

High-level programming languages provide a number of control structures that are used for selection and iteration. Typically, an **IF** statement is used for selection and a **WHILE** loop is used for conditional iteration.

6.5.1 Implementing an IF Statement

An **IF** statement has a Boolean expression followed a list of statements that is performed when the expression is true, and an optional **ELSE** part when the expression is false.

Examples on IF	Assembly-Language Translation

`// One-Way Selection`
`if (a > b) { . . . }  // if part`
	`  mov eax, a`
	`  cmp eax, b`
	`  jle end_if`
	`  . . .  ; if part`
	`end_if:`

`// Two-Way Selection`
`if (a <= b)`
`  { . . . }  // if part`
`else`
`  { . . . }  // else part`
	`  mov eax, a`
	`  cmp eax, b`
	`  jg  else_part`
	`  . . .  ; if part`
	`  jmp end_if`
	`else_part:`
	`  . . .  ; else part`
	`end_if:`

`// Short-Circuit AND`
`if (a > b && a == c) { . . . }`
	`  mov eax, a`
	`  cmp eax, b`
	`  jle end_if`
	`  cmp eax, c`
	`  jne end_if`
	`  . . .  ; if part`
	`end_if:`

`// Short-Circuit OR`
`if (a > b || a == c ) { . . . }`
	`mov eax, a`
	`cmp eax, b`
	`jg if_part`
	`cmp eax, c`
	`jne end_if`
	`if_part:`
	`. . .  ; if part`
	`end_if:`

6.5.2 Implementing a WHILE Statement

A **WHILE** statement has a Boolean expression followed a list of statements that is performed repeatedly as long as the Boolean expression is true. The following table shows two translations for the **WHILE** statement.

WHILE Statement	Assembly-Language Translation

`// First Translation`
`while (a > b) { . . . }`
	`start_while:`
	`mov eax, a`
	`cmp eax, b`
	`jle end_while`
	`. . .  ; while body`
	`jmp start_while`
	`end_while:`

`// Second Translation`
`while (a > b) { . . . }`
	`jmp bool_expr`
	`while_body:`
	`. . .  ; while body`
	`bool_expr:`
	`mov eax, a`
	`cmp eax, b`
	`jgt while_body`
	`end_while:`

In the first translation, the Boolean expression is placed before the loop body, whereas in the second translation, it is placed after the loop body. Both translations are correct, but the second translation is slightly better than the first one. The first translation has one forward conditional jump and one backward unconditional jump, which are executed for each loop iteration, while the second translation has only one conditional backward jump. The forward unconditional jump at the beginning of the second translation is done once.

6.5.3 Lab Work: Translating Nested Control Structures

Translate the following high-level control structure into assembly-language code:

```
while (a <= b) {
  a++;
  if (b == c)
    a = a + b
  else {
    b = b - a
    c--;
  }
}
```

6.6 Conditional Loop Instructions

LOOP is a non-conditional instruction that uses the **ECX** register to maintain a repetition count. Register **ECX** is decremented and the loop repeats until **ECX** becomes zero.

LOOPZ (Loop if Zero) is a conditional loop instruction that permits a loop to continue while the Zero flag is set and the unsigned value of **ECX** is greater than zero. This is what happens when the **LOOPZ** instruction is executed:

```
ECX = ECX – 1
if (ECX > 0 and ZF == 1) jump to target instruction
```

LOOPE (Loop if Equal) instruction is equivalent to **LOOPZ**.

LOOPNZ (Loop if Not Zero) instruction is the counterpart of the **LOOPZ** instruction. The loop continues while the Zero flag is clear and the unsigned value of **ECX** is greater than zero. This is the action of the **LOOPNZ** instruction:

```
ECX = ECX – 1
if (ECX > 0 and ZF == 0) jump to target instruction
```

LOOPNE (Loop if Not Equal) instruction is equivalent to **LOOPNZ**.

These instructions are summarized in the following table:

Instruction	Action
`LOOP target`	`ECX = ECX – 1` `If (ECX > 0) jump to target`
`LOOPZ target` `LOOPE target`	`ECX = ECX – 1` `If (ECX > 0 and ZF == 1) jump to target`
`LOOPNZ target` `LOOPNE target`	`ECX = ECX – 1` `If (ECX > 0 and ZF == 0) jump to target`

6.7 Lab Work: Linear Search of an Integer Array

The following program demonstrates the use of the **LOOPNZ** instruction.

```
TITLE Linear Search  (search.asm)

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
; First element is at index 0
intArray       SDWORD   18,20,35,-12,66,4,-7,100,15
item           SDWORD   -12
FoundStr       BYTE    " is found at index ", 0
NotFoundStr    BYTE    " is not found", 0

.code
main PROC
    mov    ecx, LENGTHOF intArray   ; loop counter
    mov    eax, item                ; item to search
    mov    esi, -1                  ; index to intArray
```

```
L1:
    inc    esi                      ; increment index before search
    cmp    intArray[4*esi], eax     ; compare array element with item
    loopnz L1                       ; loop as long as item not found
    jne    notFound                 ; item not found

found:
    call  WriteInt                  ; write item
    mov   edx, OFFSET FoundStr
    call  WriteString               ; " is found at index "
    mov   eax, esi
    call  WriteDec                  ; Write index
    jmp   quit

notFound:
    call  WriteInt                  ; write item
    mov   edx, OFFSET NotFoundStr
    call  WriteString               ; " is not found"

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

Study the above program and write the Console Output in the specified box.

Console Output

6.7.1 Lab Work: Assemble, Link, and Run search.exe

Check your answer in the above program and make the necessary corrections.

Modify the *item* value in the above program from **-12** to **100**. Write below the console output. Reassemble, link, and run the modified program and check your answer.

Console Output (item = 100)

Repeat the above process but with *item* equal to **-10**. Write the Console Output in the box shown below.

Console Output (item = –10)

6.8 Indirect Jump and the Switch Statement

So far, we have used *direct jump* instructions, where the target address is specified in the jump instruction itself. In an indirect jump, the target address is specified indirectly through memory. The syntax of the indirect jump is as follows for the 32-bit FLAT memory model:

```
jmp mem32      ; mem32 is a 32-bit memory location
```

6.8.1 Lab Work: Implementing a Switch Statement

Indirect jumps can be used to implement multiway conditional statements, such as a *switch* statement in a high-level language. As an example, consider the following code:

```
switch (ch) {
  case '0':
    exit();
  case '1':
    value++;
    break;
  case '2':
    value--;
    break;
  case '3':
    value += 5;
    break;
  case '4':
    value -= 5;
    break;
}
```

To have an efficient translation of the **switch** statement, we need to build a **jump table of pointers**. The input character is converted into an index into the jump table and the indirect jump instruction is used to jump into the correct case to execute. The following assembly-language program is a translation to the above statement. The switch statement is executed repeatedly until the user enters **0**, at which point the program terminates.

The program uses a jump table, called *table*. This *table* is declared inside and is local to the *main* procedure. This *table* is an array of labels (*case0*, *case1*, …, *case4*). Each label is translated into a 32-bit address, which is the address of the instruction that comes after the label. The first jump instruction in the *main* procedure is used to bypass the *table* and to start execution at the *break* label, where *value* is displayed and the user is prompted to enter a digit between **0** and **4**. The input character is checked. If it is out of range (<'**0**' or >'**4**'), it is ignored and a new character is read. Otherwise, the input character is echoed on the screen, converted from a character into a number, and then used to index the jump *table* in the indirect jump. This indirect jump transfers control to the corresponding case label.

```
TITLE Demonstrating Indirect Jump (IndirectJump.asm)

; This program shows the implementation of a switch statement
; A jump table and indirect jump are used

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
value      SDWORD  0
valuestr   BYTE    "Value = ",0
prompt     BYTE    "Enter Selection [Quit=0,Inc=1,Dec=2,Add5=3,Sub5=4]: ",0

.code
main PROC
; Start at break to bypass the jump table
    jmp break

; Jump table is an array of labels (instruction addresses)
table       DWORD   case0, case1, case2, case3, case4

; Implementing a Switch Statement
case0:
    exit
case1:
    inc value
    jmp break
case2:
    dec value
    jmp break
```

```
case3:
    add value, 5
    jmp break
case4:
    sub value, 5
    jmp break

break:
; Display value
    mov  edx, OFFSET valuestr
    call WriteString
    mov  eax, value
    call WriteInt
    call Crlf

; Prompt for the user to enter his selection
    mov  edx, OFFSET prompt
    call WriteString

; Read input character and check its value
readch:
    mov  eax,0          ; clear eax before reading
    call ReadChar
    cmp  al, '0'
    jb   out_of_range   ; character < '0'
    cmp  al, '4'
    ja   out_of_range   ; character > '4'
    call WriteChar      ; echo character
    call Crlf
    sub  al, 30h        ; convert char into number
    jmp  table[4*eax]   ; Indirect jump using table

; Out of range: ignore input and read again
out_of_range:
    jmp  readch
main ENDP
END main
```

Determine the Console Output when the user input sequence is **1, 3, 2, 0.**

Console Output

6.8.2 Lab Work: Assemble, Link, and Run IndirectJump.exe

Check your answer for the above program and make the necessary corrections.

Review Questions

1. Which conditional jump instructions are based on unsigned comparisons?
2. Which conditional jump instruction is based on the contents of the ECX register?
3. (*Yes/No*) Are the JA and JNBE instructions equivalent?
4. (*Yes/No*) Will the following code jump to the *Target* label?

```
mov ax, -42
cmp ax, 26
ja  Target
```

5. Write instructions that jump to label L1 when the unsigned integer in DX is less than or equal to the unsigned integer in CX.
6. (*Yes/No*) The LOOPE instruction jumps to a label if and only if the *zero flag* is clear.
7. Implement the following statements in assembly language, for signed integers:

```
if (ebx > ecx) X = 1;
if (edx <= ecx && ecx <= ebx) X = 1; else X = -1;
while (ebx > ecx || ebx < edx) X++;
```

Programming Exercises

1. Write a program that uses a loop to input signed 32-bit integers from the user and computes and displays their minimum and maximum values. The program terminates when the user enters an invalid input.

2. Using the following table as a guide, write a program that asks the user to enter an integer test score between 0 and 100. The program should display the appropriate letter grade. The program should display an error message if the test score is <0 or >100.

90 to 100	85 to 89	80 to 84	75 to 79	70 to 74	65 to 70	60 to 64	55 to 59	0 to 54
A+	A	B+	B	C+	C	D+	D	F

3. Write a program that reads a single character ‘0’ to ‘9’, ‘A’ to ‘F’, or ‘a’ to ‘f’, and then converts it and displays it as a number between 0 and 15. Use the *ReadChar* procedure to read the character. Do not use the *ReadHex* procedure. Display an error message if the input character is invalid.

4. Write a program that reads a string of up to 50 characters and stores it in an array. It should then convert each lowercase letter to uppercase, leaving every other character unchanged. The program should output the modified string.

5. Write a program that reads a string of up to 50 characters and stores it in an array. It then asks the user to input a single character and matches this character against all occurrences in the string. Each matched character in the string should be converted into a blank.

6. Write a program that inputs an unsigned integer *sum* > 1, and then computes and displays the smallest integer *n*, such that 1 + 2 +…+ *n* > sum. An error message should be displayed if the input *sum* ≤ 1.

