National University of computer and emerging sciences

Computer Organization and Assembly Language (EL-2003)

Semester: Fall 2025
Section: BCS-3D/3H
Course Instructor: Muhammad Owais

LAB # 06

Introduction to Jump Instructions, Loops, Nested Loops, and Built-in Procedures in Irvine Library

Lab Objectives:

By the end of this lab, students will be able to:

- Understand the working and purpose of jump instructions in assembly language.
- Learn how to implement loops and nested loops using assembly instructions.
- Explore and practice built-in procedures of the Irvine library for input, output, and other common tasks.
- Develop logical thinking by applying control flow in assembly programs.
- Strengthen problem-solving skills by writing efficient and structured assembly code.

CMP Instruction

The CMP (Compare) instruction in assembly language is used to compare two operands**.**

- It performs a subtraction internally (destination – source), but it does not store the result**.**
- Instead, it only affects the CPU flags (such as Zero Flag, Sign Flag, Carry Flag, and Overflow Flag).
- These flags are then used by conditional jump instructions to decide the program flow.

Why is CMP Used?

The CMP instruction is mainly used to:

- Compare two values (register-to-register, register-to-memory, or register-to-immediate).
- Control decision-making in programs by working with conditional jumps.
- Check conditions like equality, greater than, less than, etc.

CMP Instruction

Syntax

CMP Destination, Source

Purpose

The CMP instruction is used for comparison between two operands. It is commonly applied in decision-making statements, conditional jumps, and loops.

Working Principle

When we compare two numbers using CMP, there are three possibilities:

1. First number is greater than the second number
2. First number is less than the second number
3. First number is equal to the second number

The CMP instruction works internally as:

Destination - Source

However, unlike the subtraction instruction, the result is not stored in the destination register. Instead, the CPU flags are affected according to the result.

Flags Affected

- The **Sign Flag (SF)**, **Zero Flag (ZF)**, and **Carry Flag (CF)** are the main flags used for interpretation.
- The **Auxiliary Carry Flag (AF)** and **Parity Flag (PF)** are not relevant in comparisons and can be ignored.

Cases of CMP Instruction

1. **If Destination > Source**
    - Subtraction result is positive.
    - Flags: CF = 0, ZF = 0, SF = 0
    - Meaning: First number is greater.
2. **If Destination = Source**
    - Subtraction result is zero.
    - Flags: CF = 0, ZF = 1, SF = 0

- Meaning: Both numbers are equal.

3. **If Destination < Source**
   - Subtraction result is negative.
   - Flags: CF = 1, ZF = 0, SF = 1
   - Meaning: First number is smaller.

CODE – 01 [Demonstrate CMP with all three outcomes]

```
INCLUDE Irvine32.inc

.code
main flROC

    ; -------- Case 1: Equal --------
    mov eax, 10
    mov ebx, 10
    cmp eax, ebx          ; eax - ebx = 0, ZF = 1
    call DumpRegs         ; show registers + flags

    ; -------- Case 2: Greater --------
    mov eax, 15
    mov ebx, 10
    cmp eax, ebx          ; eax - ebx = 5, eax > ebx
    call DumpRegs         ; show registers + flags

    ; -------- Case 3: Less --------
    mov eax, 5
    mov ebx, 10
    cmp eax, ebx          ; eax - ebx = -5, eax < ebx
    call DumpRegs         ; show registers + flags

    exit
main ENDfl
END main
```

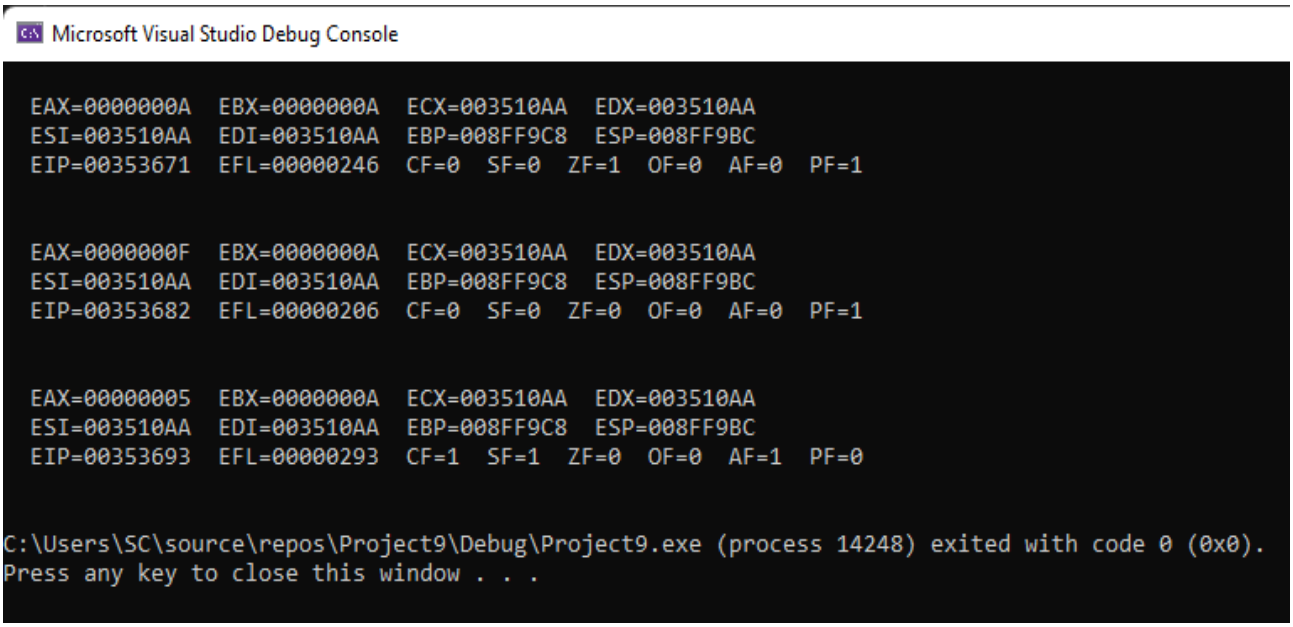

Figure 1: Output of Code – 01

JUMP Instructions

In assembly language, jump instructions are used to change the normal sequential flow of a program. Unlike high-level languages that provide constructs such as if-else, while, and for, assembly language relies on jump instructions to achieve decision-making and repetition.

When a jump instruction is executed, the Instruction Pointer (IP/EIP) is modified so that execution continues from the target location (label) instead of the next sequential instruction.

Types of Jump Instructions

Jump instructions are broadly divided into two main types:

1. Unconditional Jump Instructions
2. Conditional Jump Instructions

Unconditional Jump Instructions

An unconditional jump always transfers program control to the target location, without checking any condition.

Syntax:

JMP destination

Execution always continues from the specified destination label.

Example:

```
mov eax, 10
jmp Target
mov eax, 20      ; this instruction is skipped

Target:
mov ebx, 30
```

Here, the instruction mov eax, 20 is never executed because the control jumps directly to the label Target.

Conditional Jump Instructions

- A conditional jump transfers control only if a specific condition is true**.**
- If the condition is false, execution continues with the next sequential instruction.

Syntax:

Jcond destination

where Jcond is the jump condition mnemonic (e.g., JE, JNE, JC).

Conditional jumps are usually executed in two steps**:**

1. A condition is tested (using CMP, TEST, or arithmetic operations).
2. Based on the flags set, the program either jumps to the specified label or continues sequentially.

The following program demonstrates Unconditional jump:

CODE – 02 [Demonstrate Unconditional Jump]

```
INCLUDE Irvine32.inc

.data
   msg1 BYTE "This is the first message",0
   msg2 BYTE "This is the second message",0

.code
main PROC

   ; Print first message
   mov edx, OFFSET msg1
   call WriteString
   call Crlf

   ; Unconditional jump to skip printing msg2
   jmp SkipSecondMessage

   ; This part will be skipped
   mov edx, OFFSET msg2
   call WriteString
   call Crlf

SkipSecondMessage:
   exit

main ENDP
END main
```

Explanation of Output

- jmp SkipSecondMessage → unconditionally transfers control to the label SkipSecondMessage.
- The code between jmp and the label never executes (msg2 won't be printed).
- Demonstrates that unconditional jump does not check conditions, it always jumps.

Figure 2: Output of Code – 02

Loop Instruction

In assembly language, loops are used to execute a block of instructions repeatedly. Unlike high-level languages that provide constructs such as for, while, or do-while, assembly language implements repetition using loop instructions.

The most common **counter register** used for loops is CX (16-bit) or ECX (32-bit). The loop instructions automatically **decrement CX/ECX** and check its value to decide whether to continue looping or exit.

Syntax

LOOP destination

Working of LOOP Instruction

1. The LOOP instruction **decrements CX/ECX** by 1.
2. If CX/ECX ≠ 0, control jumps to the specified label (destination).
3. If CX/ECX = 0, execution continues with the next instruction after the loop.

Thus, the number of iterations is controlled by the initial value loaded into CX/ECX.

Simple LOOP Example

This program prints numbers from 1 to 5 using the LOOP instruction:

```
INCLUDE Irvine32.inc

.data
   count DWORD 5

.code
main PROC
   mov ecx, count       ; set loop counter
   mov eax, 1           ; starting number

L1:
   call WriteInt        ; print value of eax
   call Crlf
   inc eax              ; increment number
   loop L1              ; decrement ECX and jump if not zero

   exit
main ENDP
END main
```

Output:

```
1
2
3
4
5
```

NESTED LOOP

Simple NESTED LOOP Example

```
INCLUDE Irvine32.inc
.code
main PROC
mov eax, 0
mov ebx, 0
mov ecx, 5
L1: inc eax
mov edx, ecx
call dumpregs
mov ecx, 10
L2: inc ebx
call dumpregs
loop L2
mov ecx, edx
loop L1
call DumpRegs
exit
main ENDP
```

Procedures in Irvine Library

1. **Clrscr**
   - Clears the console screen and places the cursor at the top-left corner.
   - **Registers Used:** None.
2. **Crlf**
   - Prints a newline (carriage return + line feed) on the console.
   - **Registers Used:** None.

3. **WriteBin**
    - Prints the value in **EAX** as an unsigned 32-bit integer in **binary** format.
4. **WriteChar**
    - Prints the **AL** register's content as a single character.
5. **WriteDec**
    - Prints the value in **EAX** as an unsigned 32-bit integer in **decimal** format.
6. **WriteHex**
    - Prints the value in **EAX** as a 32-bit integer in **hexadecimal** format.
7. **WriteInt**
    - Prints the value in **EAX** as a **signed** 32-bit integer in decimal format.
8. **WriteString**
    - Prints a null-terminated string from memory.
    - **Registers Used:** EDX = OFFSET String.
9. **ReadChar**
    - Waits for a single keypress and returns the character in **AL**.
10. **ReadDec**
    - Reads an unsigned decimal number from the keyboard and returns it in **EAX**.
11. **ReadHex**
    - Reads a hexadecimal number from the keyboard and returns it in **EAX**.
12. **ReadInt**
    - Reads a signed decimal number from the keyboard and returns it in **EAX**.
13. **ReadString**
    - Reads a string from the keyboard.
    - **Registers Used:** EDX = OFFSET Buffer, ECX = SIZEOF Buffer.
14. **Delay**
    - Pauses execution for the time given in **EAX** (milliseconds).
15. **Randomize**
    - Seeds the random number generator with system time.
16. **DumpRegs**
    - Displays the contents of all major registers: EAX, EBX, ECX, EDX, ESI, EDI,

ESP, EIP, EFLAGS.

17. **DumpMem**
    - Displays a block of memory in hexadecimal format.
    - **Registers Used:** ESI = Starting OFFSET, ECX = Length, EBX = Type.
18. **GetDateTime**
    - Returns the current system date and time.
19. **GetMaxXY**
    - Gets console buffer size.
    - **Registers Used:** DX = Columns, AX = Rows.
20. **GetTextColor**
    - Returns current text color.
    - **Registers Used:** AL = Foreground, AH = Background.
21. **Gotoxy**
    - Moves cursor to specified row and column.
    - **Registers Used:** DH = Row, DL = Column.
22. **MsgBox**
    - Displays a message box with a string and title.
    - **Registers Used:** EDX = OFFSET String, EBX = OFFSET Title.
23. **MsgBoxAsk**
    - Displays a Yes/No pop-up question.
    - **Registers Used:** EDX = OFFSET String, EBX = OFFSET Title.
    - **Return:** EAX = 6 (Yes), EAX = 7 (No).
24. **SetTextColor**
    - Sets text color for console output.
    - **Registers Used:** EAX = Foreground + (Background * 16).
25. **WaitMsg**
    - Displays a message and waits for Enter key press.

LAB TASKS

Task # 01:

Write an assembly program that displays the following three lines:

- "Welcome"
- "You should not see this line"
- "Goodbye"

But use an unconditional jump (JMP) so that the second message is skipped**,** and only "Welcome" and "Goodbye" appear on the screen.

Task # 02:

Write a program that prints numbers from 5 to 15 using a LOOP instruction.

- Use ECX as the counter.
- Use WriteInt to display each number.

Task # 03:

Write a program that prints the following star pattern using nested loops:

*

**

***

****

*****

- Outer loop controls the number of rows.
- Inner loop prints stars (*) in each row using WriteChar.