LAB 06 JMP, LOOP INSTRUCTION AND BUILT-IN-PROCEDURES

STUDENT NAME ROLL NO SEC

SIGNATURE & DATE

MARKS AWARDED: _____________

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), KARACHI

Version: 1.0
Date:

Lab Session 06: J MP, LOOP & BUILT-IN-PROCEDURE

Objectives:

- JMP Instruction
- Loop Instruction
- Built-in-Procedure

JMP INSTRUCTION

Jumping is the most direct method of modifying the instruction flow. A *transfer of control*, or *branch*, is a way of altering the order in which statements are executed. There are two basic types of transfers:

- **Unconditional Transfer**
- **Conditional Transfer**

UNCONDITIONAL

The unconditional jump instruction (jmp) unconditionally transfers control to the instruction located at the target address i.e. there is no need to satisfy any condition for the jump to take place. . The general format is:

JMP *destination*

When the CPU executes an unconditional transfer, the offset of *destination* is moved into the instruction pointer, causing execution to continue at the new location.

Syntax: L2:

…………………

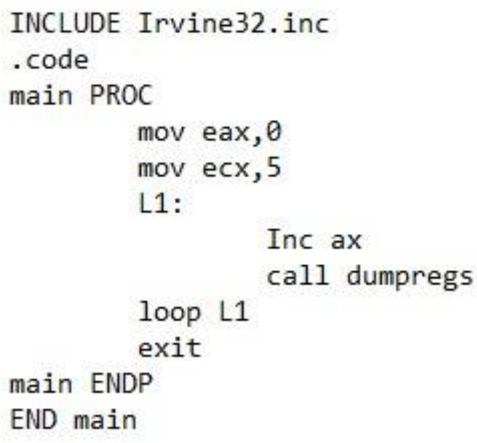
JMP L1

………………...

L1:

………...............

EXAMPLE # 01:



```
INCLUDE Irvine32.inc
.code
main PROC
        mov eax,0
        mov ecx,5
        L1:
                Inc ax
                call dumpregs
        loop L1
        exit
main ENDP
END main
```

Conditional

In these types of instructions, the processor must check for the particular condition. If it is true, then only the jump takes place else the normal flow in the execution of the statements is maintained. Syntax is:

JMP opcode *destination*

CMP Instruction

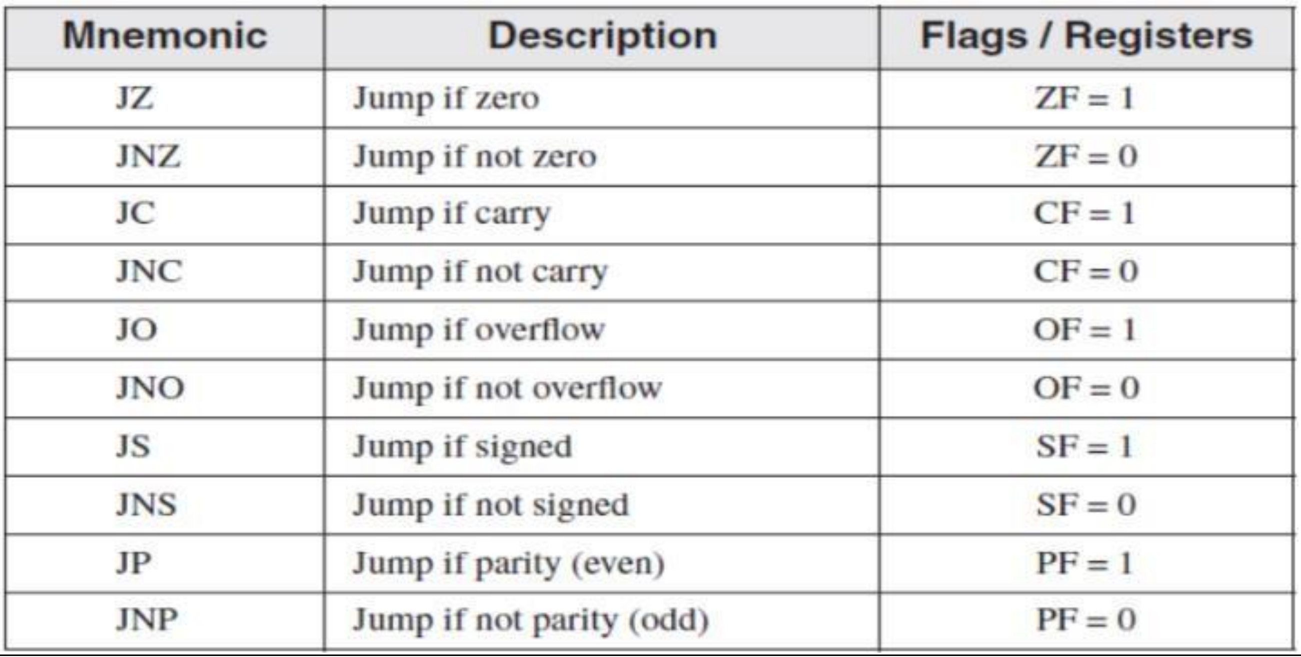
The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making. Syntax is:

CMP Destination, Source

Some conditional jump instructions treat operands of the CMP (compare) instruction as signed numbers.

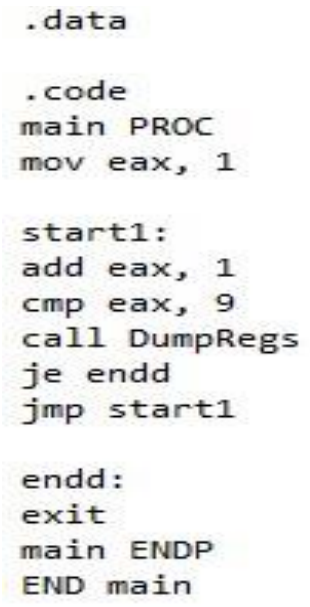
Mnemonic	Description
JE	Jump if equal
JG/JNLE	Jump if greater/Jump if not less than or equal
JL/JNGE	Jump if less/Jump if not geater
JGE/JNL	Jump if greater or equal/Jump if less
JLE/JNG	Jump if less or equal/Jump if not greater
JNE	Jump if not equal

Some conditional jump instructions can also test values of the individual CPU flags:



Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

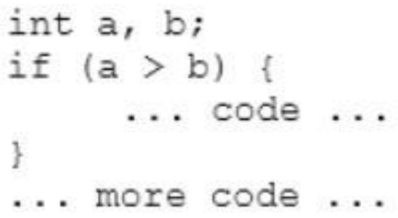
EXAMPLE # 02:



```
.data

.code
main PROC
mov eax, 1

start1:
add eax, 1
cmp eax, 9
call DumpRegs
je endd
jmp start1

endd:
exit
main ENDP
END main
```

Task 01:

Implement the following C if statement into asambly code:



```
int a, b;
if (a > b) {
      ... code ...
}
... more code ...
```

LOOP INSTRUCTION

The LOOP instruction, formally known as *Loop According to ECX Counter*, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats. Its syntax is:

LOOP *destination*

The execution of the LOOP instruction involves two steps: First, it subtracts 1 from ECX. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by *destination*. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruction following the loop.

EXAMPLE # 01:

```
INCLUDE Irvine32.inc
.code
main PROC
mov ax,0
mov ecx,5
L1:
        Inc ax  call
        dumpregs
loop L1
exit
main ENDP  END
main
```

EXAMPLE # 02:

```
INCLUDE Irvine32.inc
.data
intArray WORD 100h, 200h, 300h, 400h, 500h
.code
main PROC
mov esi, 0
mov eax, 0
mov ecx, LENGTHOF intArray
call dumpregs  L1:
        mov ax, intArray[esi]  add
        esi, TYPE intArray  call
        dumpregs
loop L1
exit
main ENDP  END
main
```

Nested Loops

When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. You can save it in a variable.

EXAMPLE # 03

```
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
```

Procedure in Irvine32 Library

1. **Clrscr**
   Clears the console window and locates the cursor at the above left corner.

2. **Crlf**
   Writes the end of line sequence to the console window.

3. **WriteBin**
   Writes an unsigned 32-bit integer to the console window in ASCII binary format.

4. **WriteChar**
   Writes a single character to the console window.

5. **WriteDec**
   Writes an unsigned 32-bit integer to the console window in decimal format.

6. **WriteHex**
   Writes a 32-bit integer to the console window in hexadecimal format.

7. **WriteInt**

Writes a signed 32-bit integer to the console window in decimal format.

8. **WriteString (EDX= OFFSET String)**
   Write a null-terminated string to the console window.

9. **ReadChar**
   Waits for single character to be typed at the keyboard and returns that character.

10. **ReadDec**
    Reads an unsigned 32-bit integer from the keyboard.

11. **ReadHex**
    Reads a 32-bit hexadecimal integers from the keyboard, terminated by the enter key.

12. **ReadInt**
    Reads a signed 32-bit integer from the keyboard, terminated by the enter key.

13. **ReadString (EDX=OFFSET, ECX=SIZEOF)**
    Reads a string from the keyboard, terminated by the enter key.

14. **Delay (EAX)**
    Pauses the program execution for a specified interval (in milliseconds).

15. **Randomize**
    Seeds the random number generator with a unique value.

16. **DumpRegs**
    Displays the EAX, EBX, ECX, EDX, ESI, EDI, ESP, EIP and EFLAG registers.

17. **DumpMem (ESI=Starting OFFSET, ECX=LengthOf, EBX=Type)** Writes the block of memory to the console window in hexadecimal.

18. **getDateTime**
    Gets the current date and time from system

19. **GetMaxXY (DX=col, AX=row)**
    Gets the number of columns and rows in the console window buffer.

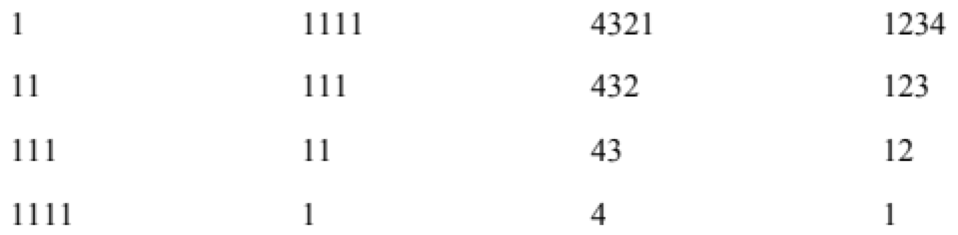
20. **GetTextColor (Background= Upper AL, Foreground= Lower AL)**
    Returns the active foreground and background text colors in the console window.

21. **Gotoxy (DH=row , DL=col)**

Exercise: Dry Run on Paper First then on IDE

Task: 1 Write a program that uses a loop to calculate the first ten numbers of Fibonacci sequence.

Task: 2 write a program that uses a nested loop to implement following patterns.



```
1          1111       4321       1234
11         111        432        123
111        11         43         12
1111       1          4          1
```

Task: 3 write a program to take input data for 5 employees and store it in appropriate variables. The program should ask for Employee ID, Name, Year of Birth & Annual Salary from the user. All variables should be stored in an array whose index represent employee number. The program should then calculate the annual salary for all employees by adding all the elements in AnnualSalary array.

Task: 4 Initialize an array named Source and use a loop with indexed addressing to copy a string represented as an array of bytes with a null terminator value in an array named as target.

Task: 5 Use a loop with direct or indirect addressing to reverse the elements of an integer array in place. Do not copy elements to any other array. Use SIZEOF, TYPE and LENGTHOF operators to make program flexible.

Task: 6 initialize a double word array consisting of elements 8, 5,1,2,6. Sort the given array in ascending order using bubble sort.