# LAB # 6

# To Explain and Show the Output of Jump and Control Instructions using EMU8086 Software Tool

**Objectives**

* To explain and show the output of conditional and unconditional jump Instruction using emu8086 software tool.
* To organize the assembly programs based on subroutines using emu8086 software tool.
* To explain and show working of label using assembly language program.

**Pre-Lab Exercise**

Read the details given below in order to comprehend the basic operation of Jump and Control, Loop and Call instructions. Study in detail and become familiar with the various ways and combinations in which these instructions can be used.

**Labels**

When an instruction is stored in ROM, it is stored in individual location out of many available locations. Each location has its unique identification code called its address. Label is basically the address on which subsequent instruction is stored in ROM. For convenience, programmer can use an alphanumeric string to represent the address of any instruction in program. Consider following piece of assembly code written for 8088 CPU.

MOV AX, BX

MSI:

MOV BX, 0ABCDH OR BL, 35H AND BH, BL HLT

The string MSI is label, and its associated instruction is MOV BX, 0ABCDH. Label name must end with ‘:’ character. When this program would be assembled, label MSI will be translated to the address of MOV BX, 0ABCDH instruction, whatever it might be. Note that instructions in ROM are placed in the same order as they appear in program code. There are some rules that must be followed when selecting the label name. Label name should not be an instruction such as AND, XOR, CALL etc. Label name should not start with symbols such as +, -, % etc., though \_ (underscore can be used as starting character). Label names should not have space in between. For example, MY LABEL is an incorrect label name. If label name is to be separated for easy reading and understanding, then \_ should be used. For example, MY\_LABEL will be a valid replacement of MY LABEL.

Label names should be self-descriptive. They should depict the purpose for which they are defined.

**Subroutines**

A subroutine is a chunk of code that executes a special operation in entire program. A subroutine starts with a label name and end in RET (return) instruction. Consider following piece of code.

MY\_SUB:

MOV AX, 5623H ADD AX, 1200H MOV DX, 2255H OR DX, AX RET

The above piece of code is written to demonstrate the concept of subroutine. It may not be executing some useful operation but points to consider are the start of subroutine with label MY\_SUB and its end with RET instruction.

**JMP Instruction**

Abbreviation of jump, JMP instruction directs the program flow to the instruction associated with the label name that follows the JMP instruction. For example, consider following use of JMP instruction.

JMP CIIT

This instruction diverts the program flow to the instruction which is labeled as CIIT. Program starts executing from that instruction and flows in a normal sequential way onwards. Consider following piece of code.

MOV CL, BH AND BH, 32H JMP MSI\_LAB XOR AX, AX DEC BX

MSI\_LAB:

MOV AH, 16

MOV CL, 3 SHR AH, CL

Program execution starts from line 1 and goes to line 3 according to normal flow. In line 3, instruction JMP is encountered that directs the program flow to instruction in line 7. Remember that a label name is not an instruction. Once program flow is directed to line 7 then it continues execution subsequently by executing instructions in line 8 and then in line 9 and so on. Redirecting the program from one instruction to another which is not in subsequence of previous instruction is called jumping. As in this example program jumps to MSI\_LAB label with no dependency on results of last instruction, it is called unconditional jumping.

**CMP Instruction**

CMP (compare) instruction compares its operands through subtraction. CMP does not modify the operands rather it updates the flag register only. Consider use of CMP given below.

CMP BL, AL

This instruction subtracts the contents of AL register from contents of BL register, but result is not sored anywhere. Only flags are updated according to one of three possible outcomes which are “Is AL greater than BL”, “Is AL less than BL” and “Is AL is equal to BL”.

**JZ Instruction**

JZ (Jump if Zero) is conditional jump instruction. It is used as follows.

JZ Label name

IF result of last operation (operation completed just before using JZ instruction) is zero, zero flag (ZF) would have been set, otherwise cleared. This instruction checks the ZF and if it is found set, then program is directed to the instruction associated with label name that is used in JZ instruction. Otherwise, jumping is skipped and instruction following the JZ is executed. Consider following piece of code.

ZERO:

MOV AX, 3456 MOV BX, AX SUB AX, BX JZ ZERO

AND BX, 2345H MOV SI, BX

MOV DI, SI AND DI, AX

When instruction in line 4 is to be executed, it will be checked if ZF is set or not. As due to the instructions in line 1 to line 3 cause ZF = 1, jump to label ZF will be taken and instruction in line 8 shall be executed after instruction in line 4. If ZF would have not been set at the time of execution of instruction in line 4, then next instruction would be of line 5. Note that as label name is not an instruction, in above code, instruction in line 8 would be executed after instruction in line 6.

**JNZ Instruction**

JNZ (jump if Not Zero) behaves oppositely of JZ. Jump would be taken to specified label if ZF is cleared and will not be taken if ZF is set at the time of execution of JNZ.

**JC Instruction**

JC (Jump if Carry) directs the program flow to the specified label if CF is set at the time of execution of JC instruction. Jumping will be skipped otherwise.

**JNC Instruction**

JNC (Jump if No Carry) directs the program flow to the specified label if CF is cleared at the time of execution of JNC instruction. Jumping will be skipped otherwise.

**JG Instruction**

JG (Jump if Greater) instructions deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than operand 2, then JG will direct the flow to the label associated with it.

CMP operand 1, operand 2 JG label

**JGE Instruction**

JGE (Jump if Greater or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than or equal to operand 2, then JG will direct the flow to the label associated with it.

CMP operand 1, operand 2 JGE label

**JL Instruction**

JL (Jump if Less) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than operand 2, then JL will direct the flow to the label associated with it.

CMP operand 1, operand 2 JL label

**JLE Instruction**

JLE (Jump if Less or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than or equal to operand 2, then JLE will direct the flow to the label associated with it.

CMP operand 1, operand 2 JLE label

**CALL Instruction**

CALL is used to direct program to a subroutine. Consider following piece of code.

MOV CL, BH AND BH, 32H CALL MSI\_LAB XOR AX, AX DEC BX

MSI\_LAB:

MOV AH, 16

MOV CL, 3 RET

When program flow executes CALL instruction in line 3, it is directed to the instruction in line 7 from where it starts execution sequentially. When flow encounters the RET instruction in line 9, it directs program back to the instruction following immediately after the most recent executed CALL instruction. In this example, after executing RET instruction in subroutine MSI\_LAB, instruction in

line 4 is executed and program flows onwards normally. This is called “returning from call”. Another CALL instruction can be used without returning from a call. This is called “nested calling”

**RET Instruction**

RET (Return) instruction, normally placed at the end of a subroutine to terminate it, brings the control back to the instruction that follows immediate after the CALL instruction using which the current subroutine was called.

**LOOP Instruction**

LOOP instruction moves to prescribed label iteratively. Value in the CX register determines the number of iterations. With each LOOP execution, CX decrements automatically. LOOP keeps on executing as long as CX reaches zero.

Consider following code.

MOV CX, 100 XOR AX, AX

HERE:

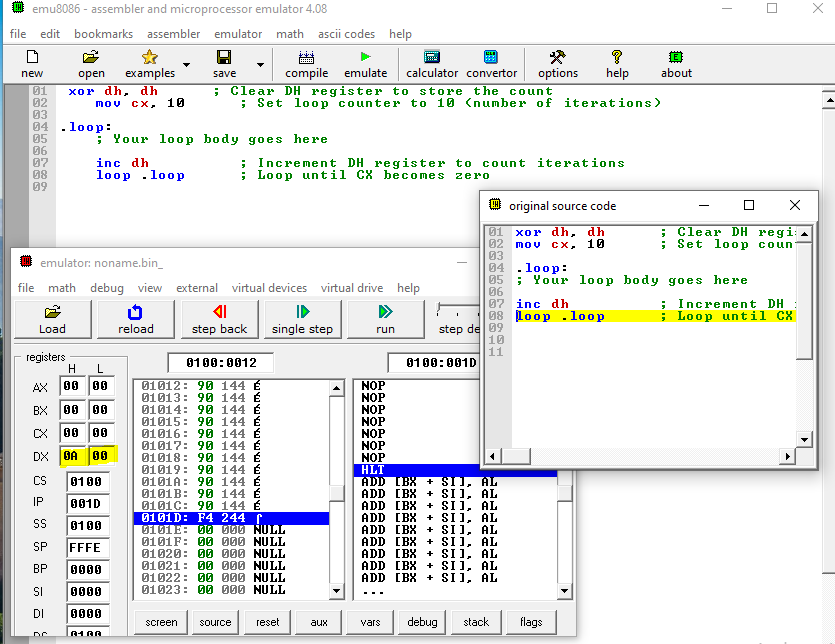
ADD AX, 1 LOOP HERE

In line 1, register CX is loaded with 100 count that determines the desired iterations of LOOP instruction. Line 2 clears the AX register. Note that this instruction has nothing to do with LOOP iterations. When line 5 is executed, flow jumps to the label HERE, CX is decremented, instruction in line 4 is executed and then LOOP HERE executes again. Now CX is decremented again (leaving 98 in it now), instruction in line 4 is executed and LOOP HERE executes again. This procedure is repeated unless CX becomes 0.

**In-Lab Exercise**

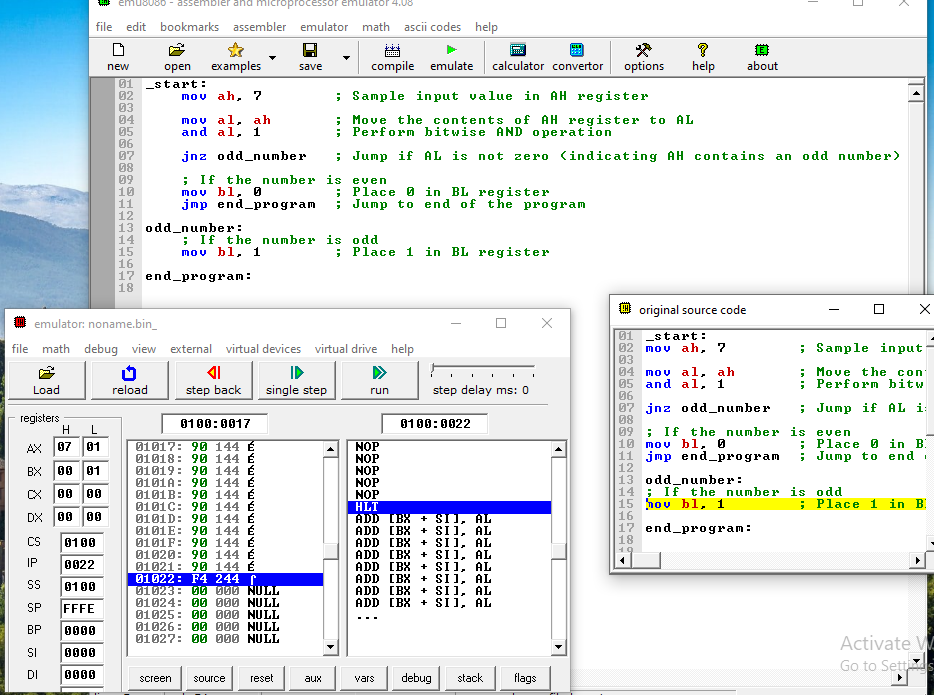
**Task 1:** Write and assembly program to count the number of 1’s using LOOP

Write an assembly language program that counts the number of ‘1’s in a byte residing in CL register. Store the counted number in DH register.



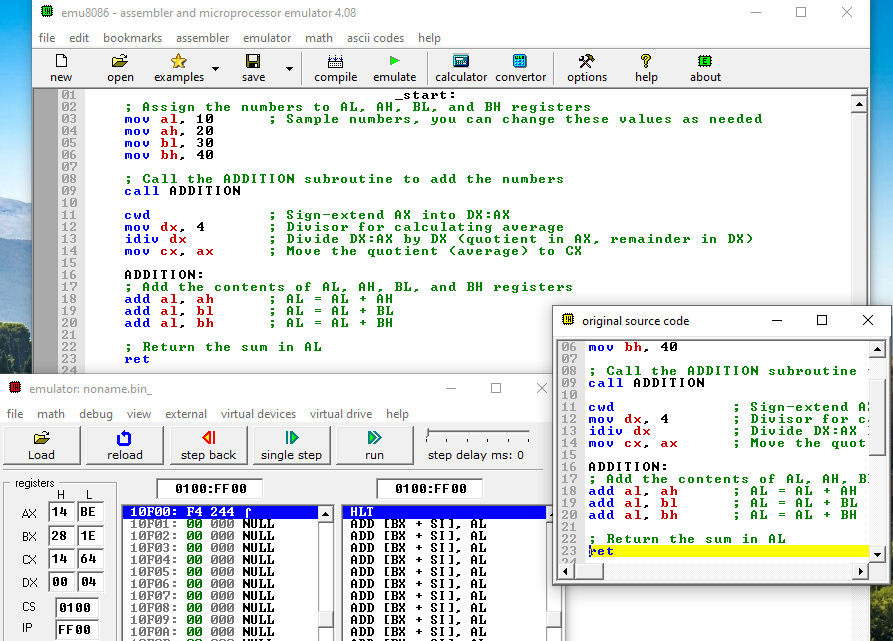
**Task 2:** To identify anodd number

Write an assembly language program, that check the contents of AH register and place a 1 in BL register if the number is Odd or a 0 otherwise.



**Task 3:** Write an assembly language program using subroutines

Write an assembly language programto calculate the average of four 8-bit numbers that uses a subroutine named ‘ADDITION’. The numbers are stored in AL, AH, BL and BH registers and their average value should be left in CX. For adding the numbers, you are required to make use a subroutine” ADDITION”



Rubric for Lab Assessment

|  |  |  |  |
| --- | --- | --- | --- |
| **The student performance for the assigned task during the lab session was:** | | | |
| Excellent | The student completed assigned tasks without any help from the instructor and showed the results appropriately. | 4 |  |
| Good | The student completed assigned tasks with minimal help from the instructor and showed the results appropriately. | 3 |  |
| Average | The student could not complete all assigned tasks and showed partial results. | 2 |  |
| Worst | The student did not complete assigned tasks. | 1 |  |

**Instructor Signature: \_\_ Date:**