# Laboratory 5

Title of the Laboratory Exercise: Controlling execution flow using conditional instructions

1. Introduction and Purpose of Experiment

   Students will be able to perform control flow operations using conditional instructions

2. Aim and Objectives

   Aim

   To develop assembly language program to perform control flow operations using conditional instructions.

   Objectives

   At the end of this lab, the student will be able to

   - Identify the appropriate assembly language instruction for the given conditional operations
   - Perform all conditional operations using assembly language instructions
   - Get familiar with assembly language program by developing simple programs

3. Experimental Procedure

   1. Write algorithm to solve the given problem
   2. Translate the algorithm to assembly language code
   3. Run the assembly code in GNU assembler
   4. Create a laboratory report documenting the work

4. Questions

   Develop an assembly language program to perform the following

   1. Print all ‘n’ natural numbers in reverse order
   2. Sum of all ‘n’ natural numbers
   3. Print all even numbers in ‘n’ natural numbers
   4. Print all odd numbers in ‘n’ natural numbers
   5. Compute GCD for the given two natural numbers
   6. Compute LCM for the given two natural numbers
   7. Develop an assembly language program to compute the parity of a hexadecimal number stored in the Register1. If Register1 has odd number of ones, update Register2 with 0x01. If Register1 has even number of ones, update Register2 with 0x00. Note: Register1 and Register2 can be any General Purpose Registers.

5. Calculations/Computations/Algorithms

```
# GCD LCM of two numbers
.section .data
a:
    .int 98
b:
    .int 56
gcd:
    .int 0
lcm:
    .int 0

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    # a = 98, b = 56
    movl a, %eax
    movl b, %ebx

loop:
    movl %eax, %edx
    cmp $0, %ebx        # if b == 0
    je loop_end         # return a

    movl $0, %edx       # clear out edx
    divl %ebx           # a = a / b, d = remainder
    movl %ebx, %eax     # a = b
    movl %edx, %ebx     # b = a % b
    jmp loop

loop_end:
    movl %edx, gcd      # gcd = GCD(a, b)
    movl a, %eax        # a = a
    mull b              # a = a * b
    divl gcd            # a = a * b / gcd
    movl %eax, lcm      # lcm = a

    syscall
    call _ret           # exit

```

```
# Conditional Instructions
.section .data
n:
    .int 10

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    # n natural numbers in reverse order and their sum
    movl n, %eax
    movl $0, %ebx
loop1:
    addl %eax, %ebx
    subl $1, %eax
    cmp $0, %eax
    jne loop1

    # even numbers in n natural numbers
    movl $2, %eax
loop2:
    addl $2, %eax
    cmp n, %eax
    jle loop2

    # odd numbers in n natural numbers
    movl $1, %eax
loop3:
    addl $2, %eax
    cmp n, %eax
    jle loop3

    syscall
    call _ret           # exit

```

```
# Bit-Counting
.section .data

.section .bss

.section .text

.globl _start

# function for system exit code
_ret:
    movq    $60, %rax               # sys_exit
    movq    $0, %rdi                # exit code
    syscall

# driver function
_start:

    movl $0x7F, %ebx    # b = 127 // actual number
    movl $0, %ecx       # c = 0 // to keep track of the number of 1's
loop:
    movl $1, %eax       # a = 1
    andl %ebx, %eax     # a = a & b
    addl %eax, %ecx     # c = c + a
    sarl %ebx
    cmp $0, %ebx
    jne loop

    movl %ecx, %eax     # a = c
    andl $1, %eax       # a = a & 1 // if even no. of 1's then 0 else 1

    syscall
    call _ret           # exit

```

6. Presentation of Results

```
(gdb) info register eax
eax            0xa      10
(gdb) c
Continuing.

Breakpoint 2, loop1 () at file.s:28
28          jne loop1
(gdb) info register eax
eax            0x9      9
(gdb) c
Continuing.

Breakpoint 2, loop1 () at file.s:28
28          jne loop1
(gdb) info register eax
eax            0x8      8
(gdb) c
Continuing.

Breakpoint 2, loop1 () at file.s:28
28          jne loop1
(gdb) info register eax
eax            0x7      7
(gdb) c
Continuing.

Breakpoint 2, loop1 () at file.s:28
28          jne loop1
(gdb) info register eax
eax            0x6      6
(gdb) c
Continuing.

Breakpoint 2, loop1 () at file.s:28
28          jne loop1
(gdb) info register eax
eax            0x5      5
(gdb) 
```

*Figure 0-1 Print N natural numbers in reverse order*

```
(gdb) info register ebx
ebx            0x37     55
(gdb) 
```

*Figure 0-2 Sum of N natural numbers*

```
(gdb) info register eax
eax            0x2      2
(gdb) c
Continuing.

Breakpoint 2, loop2 () at file.s:33
33          addl $2, %eax
(gdb) info register eax
eax            0x4      4
(gdb) c
Continuing.

Breakpoint 2, loop2 () at file.s:33
33          addl $2, %eax
(gdb) info register eax
eax            0x6      6
(gdb) c
Continuing.

Breakpoint 2, loop2 () at file.s:33
33          addl $2, %eax
(gdb) info register eax
eax            0x8      8
(gdb)
```

*Figure 0-3 odd natural numbers*

```
(gdb) info register eax
eax            0x1      1
(gdb) c
Continuing.

Breakpoint 3, loop3 () at file.s:40
40          addl $2, %eax
(gdb) info register eax
eax            0x3      3
(gdb) c
Continuing.

Breakpoint 3, loop3 () at file.s:40
40          addl $2, %eax
(gdb) info register eax
eax            0x5      5
(gdb) c
Continuing.

Breakpoint 3, loop3 () at file.s:40
40          addl $2, %eax
(gdb) info register eax
eax            0x7      7
(gdb)
```

*Figure 0-4 even natural numbers*

```
(gdb) print a
$1 = 98
(gdb) print b
$2 = 56
(gdb) c
Continuing.

Breakpoint 2, loop_end () at file.s:49
49          syscall
(gdb) print gcd
$3 = 14
(gdb) print lcm
$4 = 392
(gdb)
```

*Figure 0-5 LCM and GCD of two numbers*

7. Analysis and Discussions

Algorithm for finding GCD, LCM

```
GCD(a, b):

1. if b == 0 gcd = a

2. else gcd = GCD(b, a % b)
```

| Code | `jmp address` |
|---|---|
| Example | `jmp loop` |
| Explanation | Performs:<br>Jumps to the address location<br>Description:<br>Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.<br>This instruction can be used to execute four different types of jumps: - Near jump- A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump. |

| Code | `jcc address` |
|---|---|
| Example | `jne loop` |
| Explanation | Performs:<br>Jumps to the address location if the condition is met<br>Here cc = ne, e, ge, g, etc.<br>Description:<br>Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction. |

| | |
|---|---|
| Code | `cmp op1 op2` |
| Example | `cmp $0, %eax` |
| Explanation | Performs:<br>Compares the two operands<br>Description:<br>Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand. |

8. Conclusions

Execution Flow can be controlled by using conditional instructions, which includes a cmp instruction followed by a jump instruction, a cmp instruction compares the two operands and updates the flag register, this is then used with jump instruction to go to some other part of the program, using this we can form looping structures to do stuff like print n natural numbers, sum of them and some basic programs like LCM and GCD of two numbers, even functions can be emulated in assembly by using such structures.

9. Comments

1. Limitations of Experiments

Although looping structures can be formed using the cmp, jcc instructions but recursive structures are complex to form using just these instructions.

2. Limitations of Results

None

3. Learning happened

We learnt the use of compare, unconditional jump and conditional jump instructions to form looping structures and conditional statements.

4. Recommendations

Since a program can contain numerous loop labels, each label should be carefully names, and the programmer must keep track of which parts of the program jump to where, else there might be chances of forming infinite loops.

Signature and date

Marks