Laboratory 3

Title of the Laboratory Exercise: Logical operations

1. Introduction and Purpose of Experiment

   Students will be able to perform all logical operations using assembly instructions.

2. Aim and Objectives

   Aim

   To develop assembly language program to perform all logical operations

   Objectives

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

   - Identify the appropriate assembly language instruction for the given logical operations
   - Perform all logical 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:

   1. Consider the following source code fragment

      int a,b,c,d;

      a= (b AND c) XOR d;

      a=(b XOR c) OR d;

      Assume that *b, c, d* are in registers. Develop an assembly language program to perform this assignment statements. Assume that *b, c* are in registers and *d* in memory. Develop an assembly language program to perform this assignment statements.

2. Consider the following source code fragment

   int a ,b, c, d;

   *A = (b*c) / d;*

   Perform multiplication and division by shift operations

5. Calculations/Computations/Algorithms

5.1 To perform *a= (b AND c) XOR d;*

```
Step 1: start

Step 2: declare variable a, b, c, d
Step 3: move value of b, c, d in ebx, ecx, edx registers
Step 4: add ebx and ecx
Step 5: subtract ecx = ecx - edx
Step 6: move value of ecx to esi register
Step 7: move value of b to eax
Step 8: multiply eax = eax * c
Step 9: divide eax = eax / d
Step 10: add esi and eax
Step 11: move value of eax to a
Step 12: stop
```

5.2 To perform *a=(b XOR c) OR d;*

```
Step 1: start

Step 2: declare variable a, b, c, d
Step 3: move value of b, c, d in ebx, ecx, edx registers
Step 4: xor between ebx ,ecx
Step 5: move value of ecx to eax register
Step 6: or between edx ,eax
Step 7: move value of eax to a
Step 8: stop
```

5.3 for performing *A = (b*c) / d;* by shift operations

```
Step 1: start
Step 2: assign value to register ebx
Step 3: sall $4,%ebx # ebx = ebx * (2^4)
Step 4: sarl $2, %ebx # ebx = ebx / (2^2)
```

```
Step 5: stop
```

6. Presentation of Results

6.1 For performing *a= (b AND c) XOR d;*

a=(b XOR c) OR d;

```
# Arithmetic Operations
.section .data
a:
    .int 1 # 0001
b:
    .int 2 # 0010
c:
    .int 5 # 0101
d:
    .int 3 # 0011
.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 b,%ebx # ebx = b
    movl c,%ecx # ecx = c
    movl d,%edx # edx = d

    andl %ebx,%ecx # b and c = ecx
    movl %ecx, %eax # ecx = eax
    xorl %edx, %eax
 # eax =  (b AND c) XOR d
    movl %eax,a

    xorl %ebx,%ecx # ecx = b xor c
    movl %ecx, %eax # eax = ecx
    orl %edx,%eax # eax = (b xor c) or d

    movl %eax,a

    syscall
    call _ret           # exit
```

Figure 1 ASM code

```
Reading symbols from lab3_1a...done.
(gdb)
(gdb) break 27
Breakpoint 1 at 0x4000d5: file lab3_1a.s, line 27.
(gdb) break 33
Breakpoint 2 at 0x4000e2: file lab3_1a.s, line 33.
(gdb) break 39
Breakpoint 3 at 0x4000ef: file lab3_1a.s, line 39.
(gdb)
```

Figure 2 adding breakpoints

```
(gdb) info register ebx ecx edx
ebx            0x2      2
ecx            0x5      5
edx            0x3      3
(gdb)
```

Figure 3 moved values to registers

```
(gdb) c
Continuing.

Breakpoint 2, _start () at lab3_1a.s:34
34          xorl %ebx,%ecx # ecx = b xor c
(gdb) print a
$1 = 3
(gdb)
```

Figure 4 (b and c) xor d

```
(gdb) c
Continuing.

Breakpoint 3, _start () at lab3_1a.s:40
40          syscall
(gdb) print a
$2 = 3
(gdb)
```

Figure 5 (b xor c)or d

6.2 For performing A = *(b*c) / d;* by shift operations

```
# Arithmetic Operations
.section .data

.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 $6,%ebx # ebx = 6
    sall $4,%ebx # ebx = ebx * (2^4)
    sarl $2, %ebx # ebx = ebx / (2^2)


    syscall
    call _ret           # exit
```

Figure 6 ASM code

```
(gdb) break 18
Breakpoint 1 at 0x40008d: file sun.s, line 18.
(gdb) break 19
Breakpoint 2 at 0x400090: file sun.s, line 19.
(gdb) break 20
Breakpoint 3 at 0x400093: file sun.s, line 20.
(gdb)
```

Figure 7 Adding breakpoints

```
(gdb) r
Starting program: /mnt/d/LINUX/sun

Breakpoint 1, _start () at sun.s:18
18          sall $4,%ebx # ebx = ebx * (2^4)
(gdb) c
Breakpoint 2, _start () at sun.s:19
19          sarl $2, %ebx # ebx = ebx / (2^2)
(gdb) info register ebx
ebx            0x60     96
(gdb) c
Continuing.

Breakpoint 3, _start () at sun.s:22
22          syscall
(gdb) info register ebx
ebx            0x18     24
(gdb)
```

Figure 8 at breakpoint 2 , sall operates. at breakpoint 3 sarl operates

7. Analysis and Discussions

Code	`and <source> <destination>`
Example	`andl $20, %ebx`
Explanation	Performs: `Destination = Destination AND Source` Description: Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Code	or <source> <destination>
Example	orl $20, %ebx
Explanation	Performs: Destination = Destination OR Source Description: Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

Code	xor <source> <destination>
Example	xorl $20, %ebx

Explanation	Performs: Destination = Destination XOR Source Description: Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

Code	sal <shift_amt> <destination> sar <shift_amt> <destination>
Example	sal $2, %ebx sar $7, %ebx
Explanation	Performs: Destination = bitwise shift destination shit_amt times, either to the left or to the right, depending upon usage of sal or sar respectively. Description: The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the

	destination operand to the right (toward less significant bit locations).

8. Conclusions

To perform logical operations, we have instructions such as `and`, `or` and `xor`, that perform logical AND, logical OR and logical XOR bitwise respectively. These instructions take in two arguments, which is the source and the destination, and the operation is done for source and destination and the result is then stored in the destination.

To perform multiplication and division we use instructions such as `sal`, and `sar`, which are shift arithmetic left and shift arithmetic right respectively, these basically shift the bits in the register. Shifting the bits to the left multiplies the number by 2 and shifting the bits to the right divides the number by 2.

9. Comments

1. Limitations of Experiments

The Experiment is limited to multiplying and dividing numbers using bitwise shifting operator by only powers of 2, such as 1, 2, 4, 8 .

2. Limitations of Results

Shit Left and Shift Right instructions can only multiply the operand by a positive value, i.e. the operand can only be multiplied by a positive number or divided by a positive number.

3. Learning happened

We learnt how to use bitwise logical operators on values stored in registers and also learnt how to multiply and divide numbers by using bitwise shifting operations.

4. Recommendations

Since shifting operations take way less machine execution cycles, they are preferred over `div` and `mul` instructions.

Signature and date

Marks