NAME: SATYAJIT GHANA REG NO: 17ETCS002159

Laboratory 2

Title of the Laboratory Exercise: Arithmetic Operations

1. Introduction and Purpose of Experiment

   Students will be able to perform all arithmetic operations using assembly instructions

2. Aim and Objectives

   Aim

   To develop assembly language program to perform all arithmetic operations.

   Objectives

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

   - Identify the appropriate assembly language instruction for the given arithmetic operations
   - Perform all arithmetic operations using assembly language instructions
   - Understand different data types and memory used
   - 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 + c)-d + (b*c) / 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. For the following values of A and B, predict the values of the N, Z, V and C flags produced by performing the operation A + B. Perform the following operation. Load the values of A and B into two registers. Perform an addition of the two registers. Read the flags after each addition and compare those flag values with your predictions. Comment on the results. When the data values are signed numbers, what do the flags mean? Does their meaning change when the data values are unsigned numbers?

0xFFFF0000	0xFFFFFFFF	0x67654321 (A)
+ 0x87654321	+0x12345678	+ 0x23110000 (B)

5. Calculations/Computations/Algorithms

The FLAGS register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16 bits wide. Its successors, the EFLAGS and RFLAGS registers, are 32 bits and 64 bits wide, respectively.

The ADC instruction can be used to add two unsigned or signed integer values, along with the value contained in the carry flag from a previous ADD instruction.

The SBB instruction utilizes the carry and overflow flags in multibyte subtractions to implement the borrow feature across data boundaries.

The INC and DEC instructions are used to increment (INC) and decrement (DEC) an unsigned integer value.

```
dec destination

inc destination
```

The format for the MUL instruction is `mul source`

The format of the DIV instruction is `div divisor`

6. Presentation of Results

```
# Arithmetic Operations
.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 $10,%ebx    # b = 10
    movl $20, %ecx   # c = 20
    addl %ebx, %ecx  # b + c
    movl $15, %edx   # d = 15
    subl %edx, %ecx  # (b + c) - d
    movl %ecx, %eax

    movl $15, %ebx   # b = 15
    movl $15, %eax   # c = 15
    mull %ebx        # (b * c)
    movl $5, %ecx    # d = 5
    movl $0, %edx    # zero out edx
    divl %ecx        # (b * c) / d

    syscall
    call _ret            # exit

```

Figure 1 Source Code

```
(gdb) info registers rax rbx rcx rdx
rax            0x0       0
rbx            0xa       10
rcx            0x1e      30
rdx            0x0       0
(gdb) info register eflags
eflags         0x206     [ PF IF ]
(gdb)
```

Figure 2 info registers after b + c

```
(gdb) info registers rax rbx rcx rdx
rax            0xf       15
rbx            0xa       10
rcx            0xf       15
rdx            0xf       15
(gdb) info register eflags
eflags         0x216     [ PF AF IF ]
(gdb)
```

Figure 3 info registers after (b + c) - d

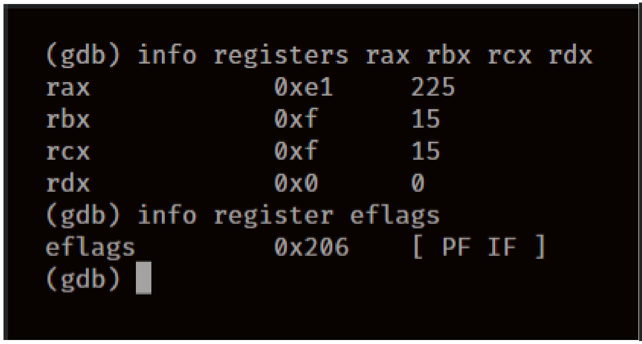

*Figure 4 info registers after b * c*

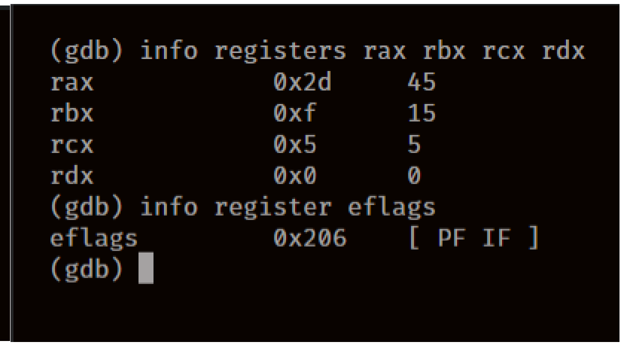

*Figure 5 info registers after (b * c) / d*

Figure 6 info registers after signed addition

7. Analysis and Discussions

Abbreviation	**Description**	**Category**	**=1**	**=0**
CF	Carry Flag	Status	CY (Carry)	NC (No Carry)
PF	Parity Flag	Status	PE (Parity Even)	PO (Parity Odd)
AF	Adjust Flag	Status	AC (Auxiliary Carry)	NA (No Auxiliary Carry)
SF	Sign Flag	Status	NG (Negative)	PL (Positive)
IF	Interrupt Enable Flag	Control	EI (Enable Interrupt)	DI (Disable Interrupt)

Code	`add <source> <destination>`
Example	`addl $20, %ebx`
Explanation	Performs:`Destination = Destination + Source` Description: Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

Code	`sub <source> <destination>`
Example	`subl $20, %ebx`
Explanation	Performs:`Destination = Destination - Source` Description: The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Code	`mul <multiplicand>`
Example	`mull $20`
Explanation	Performs: `eax = eax * multiplicand` Description:

	Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location.

Code	`div <divisor>`
Example	`divl $20`
Explanation	Performs : `eax = eax / divisor` Description: Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

8. Conclusions

To perform arithmetic operations, we have operators such as `add, sub, mul and div`, that perform addition subtraction, multiplication and division. add and sub take two arguments, which are the source and the destination, while mul and div take only one parameter which is the multiplicand or the divisor, the operation is performed and the result is stored in eax register.

Errors encountered during execution:

SIGFPE, usually encountered when there is a division by zero error when using div.

You need to zero edx before calling div ecx. When using a 32-bit divisor (e.g, ecx), div divides the 64-bit value in edx:eax by its argument, so if there's junk in edx, it's being treated as part of the dividend.

9. Comments

1. Limitations of Experiments

The mul and div operations have only one argument, hence their destination registers are fixed, this reduces the number of registers we can use to store values for operations, the operation is not as flexible since we do not have control of where the value is stored after the operation.

2. Limitations of Results

None

3. Learning happened

We were able to perform basic arithmetic operations such as addition, subtraction, multiplication and division in x86_64/x86 assembly language

We also learnt the different status codes encountered during execution of these operations.

4. Recommendations

While running the assembly code, make sure that the registers are cleared before performing a new operation, sometimes there’s junk in the register that can cause faults like SIGFPE.

Signature and date

Marks