# Laboratory 2

Title of the Laboratory Exercise: Arithmetic Operations

1. Introduction and Purpose of Experiment

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

1. 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

1. 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

1. Questions
2. 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 is* in registers and *c*, *d* in memory. Develop an assembly language program to perform this assignment statements.

Value of b= 7654321

Value of c= 3110000

Value of d=2344

1. Consider the following source code fragment

*Int a,b,c,d;*

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

Perform multiplication and division by shift operations

1. Calculations/Computations/Algorithms

5.1 For performing *a= (b + c)-d + (b\*c) / 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 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

1. Presentation of Results

6.1 For performing *a= (b + c)-d + (b\*c) / d;*



Figure 1 ASM code

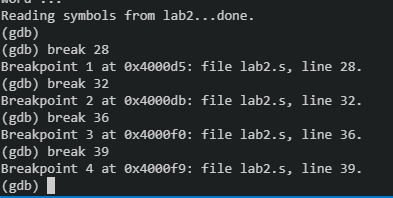


Figure 2 adding breakpoints

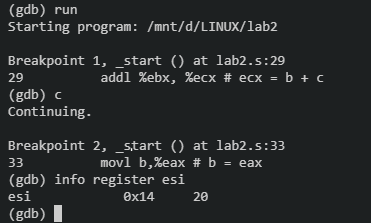


Figure 3 info register after ( b + c ) – d

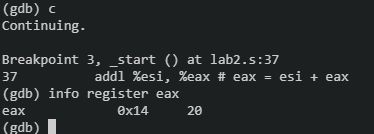


Figure 4 info register after ( b \* c ) / d

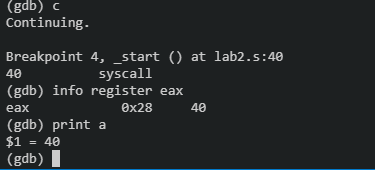


Figure 5 info register after assigning value from eax to a

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

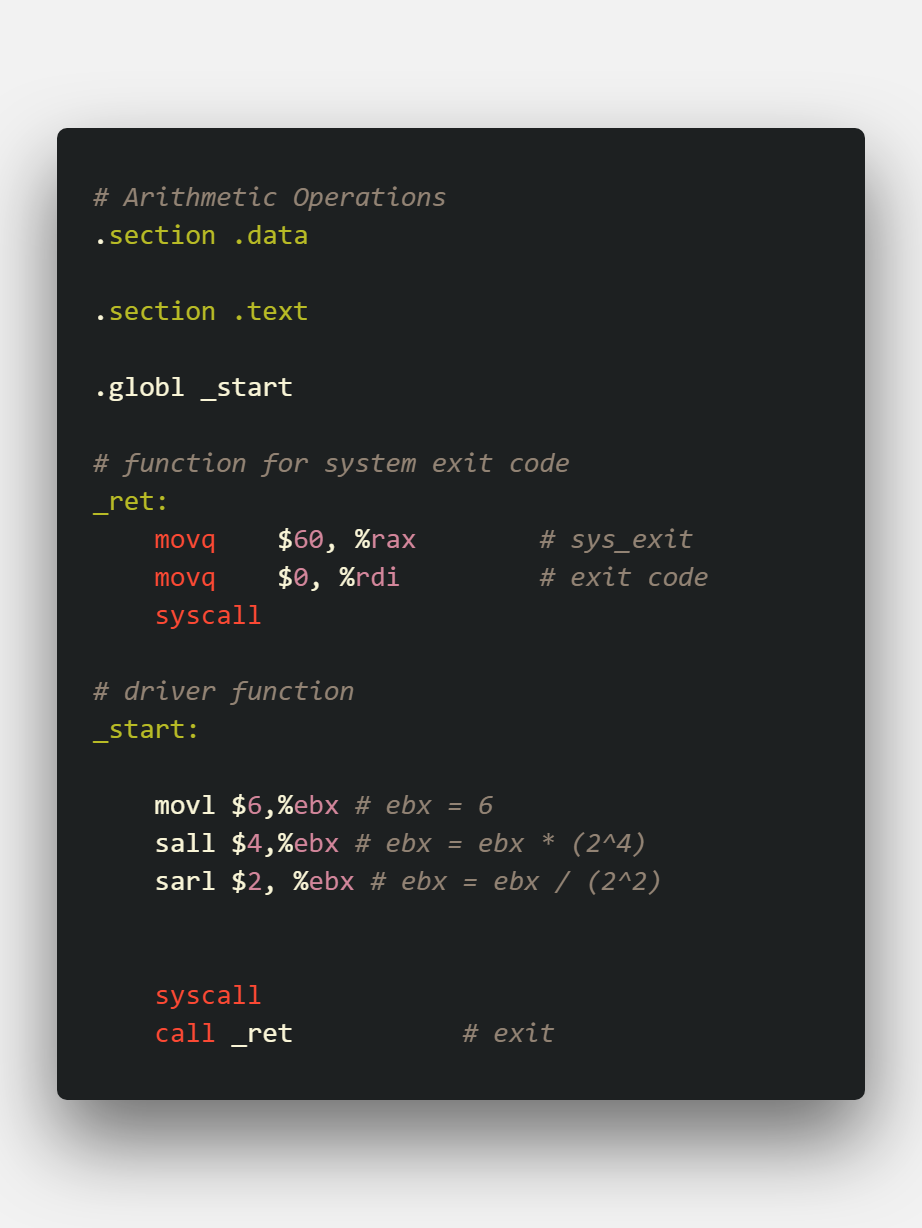


Figure 6 ASM code

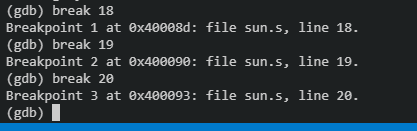


Figure 7 Adding breakpoints

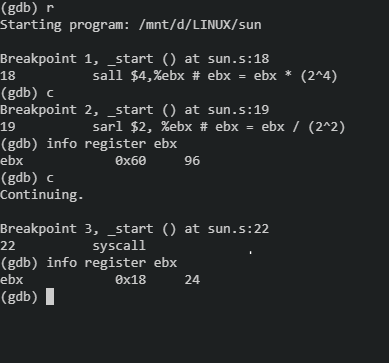


Figure 8 at breakpoint 2 , sall operates. at breakpoint 3 sarl operates

1. Analysis and Discussions

|  |  |
| --- | --- |
| 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.) |

|  |  |
| --- | --- |
| 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. |

|  |  |
| --- | --- |
| 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. Overflow is indicated with the #DE (divide error) . |

|  |  |
| --- | --- |
| 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). The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2. |

1. 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.

1. 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

Shift 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 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.

1. 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.

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

Signature and date Marks