



Course Name: *Computer Architecture Lab*
Course Number and Section: **14:332:333:04**

Experiment: *3 – Arithmetic Operations (Basic and Float) & Combinatorial Logic*

Lab Instructor: *Christos Mitropoulos*

Date Performed: *October 13th, 2016*

Date Submitted: *October 27th, 2016*

Submitted by: *FAHD HUMAYUN – 168000889 – fh186*

-----For Lab Instructor Use ONLY-----

GRADE: _____

COMMENTS:

Electrical and Computer Engineering Department
School of Engineering
Rutgers University, Piscataway, NJ 08854

Introduction:

The assignments of this lab are focused on how to do arithmetic operations on integers & floating point operations, and combinatorial logic. The instructions used in the assignments are for reading, printing, loop and branch, arithmetic and floating point operations.

Note: Codes of each assignment are attached after the conclusion of each assignment.

Assignment - 1:

The assignment is based on reading three integer numbers from the user, then calculating the given logic function and printing the output.

Logic function: $F = (A \text{ OR } C)' \text{ AND } (B \text{ AND } C)'$ (assuming A, B, and C are the three numbers entered).

Pseudocode:

Prompt user for three integer numbers

Function:

Logic OR the first and third number

Logic NOT the above result (or Logic NOR with 0)

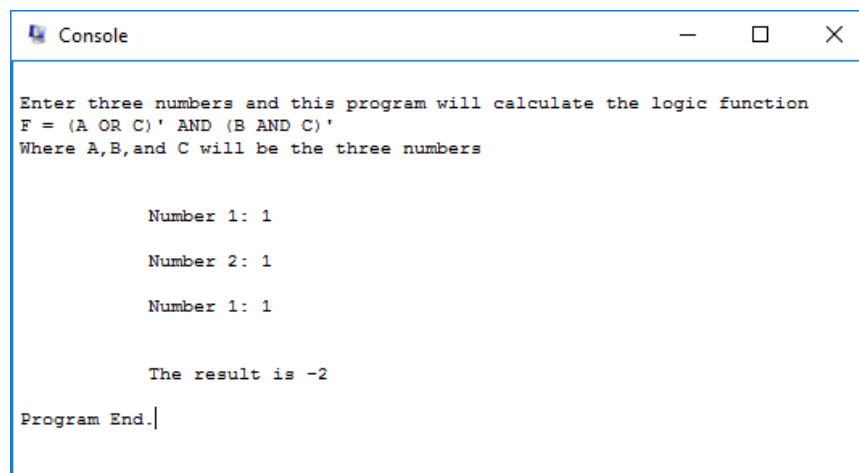
Logic AND the second and third number

Logic NOT the above result

Logic AND the two above results

Print the result

Output:



```
Console
Enter three numbers and this program will calculate the logic function
F = (A OR C)' AND (B AND C)'
Where A,B,and C will be the three numbers

      Number 1: 1
      Number 2: 1
      Number 1: 1

      The result is -2

Program End. |
```

```
Console
Enter three numbers and this program will calculate the logic function
F = (A OR C)' AND (B AND C)'
Where A,B,and C will be the three numbers

Number 1: 0
Number 2: 0
Number 1: 0

The result is -1
Program End.
```

Conclusion:

The numbers entered are decimal representation of integers, the result is obtained by performing equivalent binary bit by bit operation on each of the 32 bits of the integer numbers stored in the registers. So, in the first output integers 1, 1, 1 were entered, so in 32-bits binary representation is 0000 0000 0000 0000 0000 0000 0000 0001. Then the operations are performed like A OR C which results in 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001, after the operation of (A OR C)' it becomes 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110. Similarly, for the other operations and in the end we get the result of -2 which is the logical AND of two results from the two operations and which gives 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 which is binary equivalent of $(-2)_{10}$. The result obtained in the second console by entering 0, 0, 0 for the three numbers is obtained in the same manner.

Assignment – 2:

The assignment is based on reading two integer numbers from the user, and then computing the product of those two integers, but, without using the *mult* instruction.

The method/procedure of multiplication algorithm 1 was used for multiplication of numbers by using *shift*, and *add* instructions.

Pseudocode:

Prompt user for two integer numbers

Function:

While (all of the bits of the multiplier has been shifted/tested)

{

If (multiplier0 == 1)

Add multiplicand to product and place the result in product register

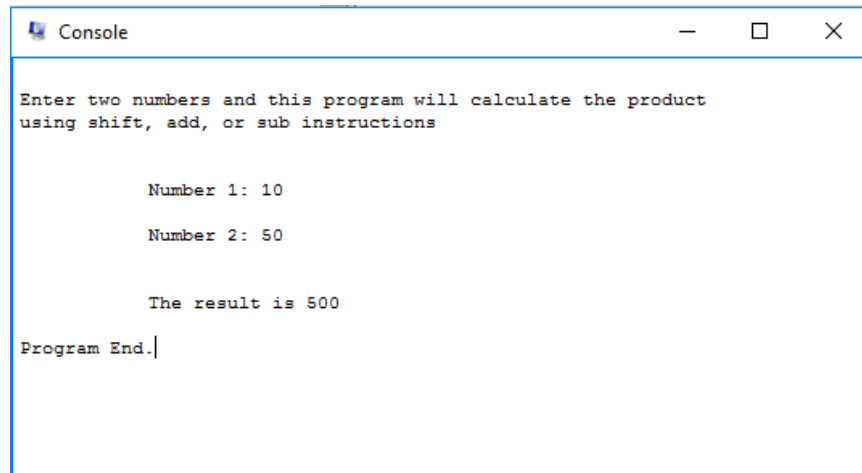
Shift the multiplicand register left 1 bit

Shift the multiplier register right 1 bit

}

Print the result

Output:



```
Console
Enter two numbers and this program will calculate the product
using shift, add, or sub instructions

Number 1: 10
Number 2: 50

The result is 500
Program End.|
```

```
Console

Enter two numbers and this program will calculate the product
using shift, add, or sub instructions

    Number 1: -10

    Number 2: 7

    The result is -70

Program End.
```

```
Console

Enter two numbers and this program will calculate the product
using shift, add, or sub instructions

    Number 1: 5

    Number 2: -3

    The result is -15

Program End.
```

```
Console

Enter two numbers and this program will calculate the product
using shift, add, or sub instructions

    Number 1: -8

    Number 2: -15

    The result is 120

Program End.
```

Conclusion:

Output for different numbers with different signs have been tested as can be seen in the console images attached above. For keeping track of the signs, some of the registers were used as flags to determine if the number is negative or positive, and then according to those flags the result was determined. If both the numbers were negative then two's complement of both of the numbers was used to compute the product, if one of the number was negative then the result is supposed to be negative as two opposite sign numbers give negative result and two same signs give positive result, so, the two's complement of result was taken if the result was expected to be negative.

Assignment – 3 (A):

The assignment is based on conversion of floating points from binary to decimal equivalent and vice versa. For converting from binary to decimal the 32-bit number in the register is split into three sections – *sign (1-bit)*, *exponent (8-bits)*, *fraction (23-bits)*, and the following equation/formula is used:

$$(number)_{10} = (-1)^{sign} * (1 + fraction) * 2^{exponent - bias}$$

Where fraction is given by $\sum_{i=1}^{23} b_{-i} 2^{-i}$, and bias = 127 for single precision and 1023 for double precision.

The 32-bit binary number given:

<i>Sign</i>	<i>Exponent</i>	<i>Fraction</i>
1	0001 1110	0011 0101 1111 0011 0011 000

$$sign = 1$$

$$exponent = 2^1 + 2^2 + 2^3 + 2^4 = 30$$

$$exponent - bias = 30 - 127 = -97$$

$$fraction = 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} + 2^{-19} + 2^{-20} = 0.210741997 = 0.2107$$

By using the above formula,

$$(number)_{10} = (-1)^1 * (1 + 0.2107) * 2^{-97}$$

$$(number)_{10} = -7.64 * 10^{-30}$$

Assignment – 3 (B):

To convert decimal number to binary the number after the decimal point is multiplied by 2, then the number before the decimal of the result is kept (i.e. either 0 or 1) and the remaining decimal fraction is multiplied by 2 again, the process is repeated till 23-bits of fraction are obtained. The number before the decimal point is converted to binary by dividing the number by 2 and then storing the quotient (i.e. either 0 or 1). After the number has been changed to binary then in order to represent in single precision binary format the number is normalized and separated in the general format that is represented by a floating point register. The exponent is determined by the number of places the decimal point is moved during the normalizing.

The decimal number given is 7.56_{10}

$$(7)_{10} = (0111)_2$$

To get the fraction,

$$\begin{aligned} 0.56 * 2 &= \mathbf{1} .12 * 2 = \mathbf{0} .24 * 2 = \mathbf{0} .48 * 2 = \mathbf{0} .96 * 2 = \mathbf{1} .92 * 2 = \mathbf{1} .84 * 2 = \mathbf{1} .68 \\ 1.68 * 2 &= \mathbf{1} .36 * 2 = \mathbf{0} .72 * 2 = \mathbf{1} .44 * 2 = \mathbf{0} .88 * 2 = \mathbf{1} .76 * 2 = \mathbf{1} .52 * 2 = \mathbf{1} .04 \\ 1.04 * 2 &= \mathbf{0} .08 * 2 = \mathbf{0} .16 * 2 = \mathbf{0} .32 * 2 = \mathbf{0} .64 * 2 = \mathbf{1} .28 * 2 = \mathbf{0} .56 * 2 = \mathbf{1} .12 \\ 7.56_{10} &= 111.100011110101110000101_2 \end{aligned}$$

Normalize,

$$\begin{aligned} &1.11100011110101110000101 * 2^2 \\ \text{power} &= \text{exponent} - \text{bias} \rightarrow \text{exponent} = \text{power} + \text{bias} = 2 + 127 = \mathbf{129} \\ \text{sign} &= \mathbf{0} \\ \text{fraction} &= \mathbf{11100011110101110000101} \end{aligned}$$

The 32-bit binary representation of 7.56_{10}

<i>Sign</i>	<i>Exponent</i>	<i>Fraction</i>
<i>0</i>	<i>1000 0001</i>	<i>1110 0011 1101 0111 0000 101</i>

There are no unsigned floating point numbers in IEEE 754 standard.

Assignment – 4:

The assignment is based on calculating cube root of a float positive number, by using the Newton method for cube root that uses the recursive formula:

$$x_{i+1} = \frac{2x_i + \frac{N}{x_i^2}}{3}$$

Where x_1 is initial guess for the cube root and in the code written it is initialized to 1. The recursion ends when the condition $|x_{i+1} - x_i| < \varepsilon$ is not true anymore, and ε is a small number initialized to 0.00001 for this assignment.

Pseudocode:

Prompt user for positive float numbers

Input validation

Function:

While ($|x_{i+1} - x_i| < \varepsilon$)

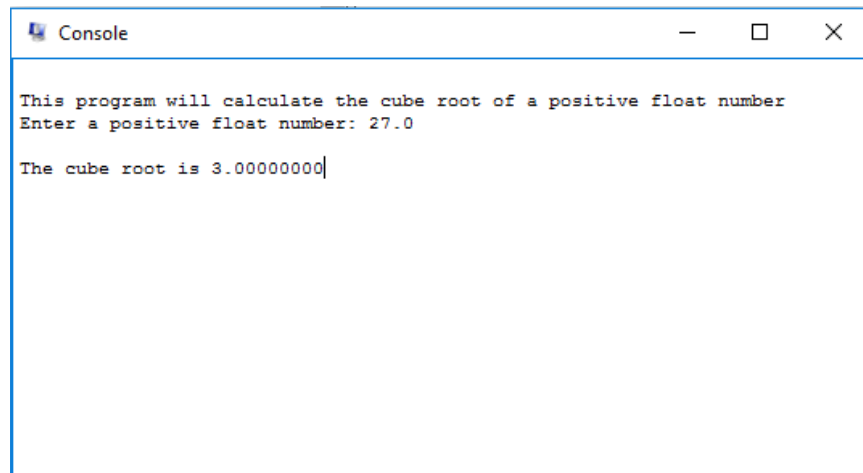
{

Compute x_{i+1} using the formula

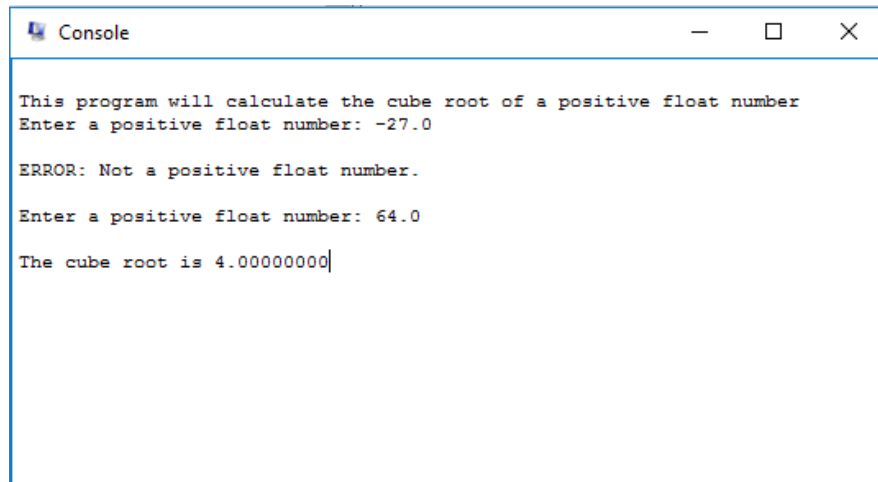
}

Print the result

Output:



```
Console
This program will calculate the cube root of a positive float number
Enter a positive float number: 27.0
The cube root is 3.00000000|
```



```
Console
This program will calculate the cube root of a positive float number
Enter a positive float number: -27.0

ERROR: Not a positive float number.

Enter a positive float number: 64.0

The cube root is 4.00000000|
```

Conclusion:

The formula is computed by breaking it into small segments and then computing the small individual segments and combining them to obtain result. Several instructions for floating point are used, and the coprocessor registers are used to store the floating point numbers. The memory is used to store the floating point values for initial values 0.0, 1.0, and $\varepsilon = 0.00001$.

Assignment – 5:

The assignment is based on calculating volume of a circular cone. User is asked for radius and height as floating point values and the volume is calculated as follow:

$$Volume = \left(\frac{1}{3}\right) * \pi * radius^2 * height$$

Pseudocode:

Prompt user for radius and height as floating point numbers

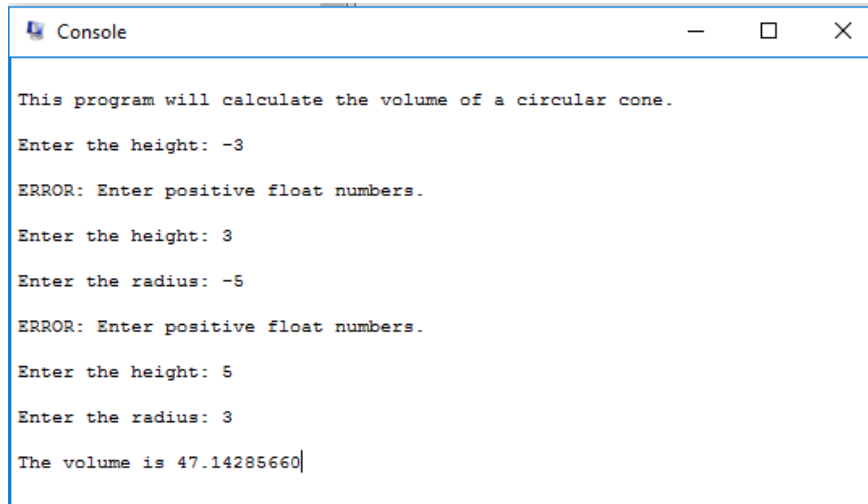
Input validation

Function:

Compute the volume using the formula

Print the result

Output:



```
Console

This program will calculate the volume of a circular cone.

Enter the height: -3
ERROR: Enter positive float numbers.

Enter the height: 3
Enter the radius: -5
ERROR: Enter positive float numbers.

Enter the height: 5
Enter the radius: 3

The volume is 47.14285660|
```

Conclusion:

Multiple operations are performed using instructions for single precision floating point. The output contains the result of user entered height and radius of 5 and 3 respectively, while, also the negative input has been rejected by the program and the user is asked to enter positive numbers for height and radius.

Assignment – 6:

The assignment is based on calculating sum of all the numbers of a floating point array what are bigger than the user input floating point. The array given is:

Array = {1.35, 2.67, 3.566, 4.56, 5.98, 9.43, 12.34, 15.54, 23.87, 34.33}

Pseudocode:

Prompt user for floating point number

Function:

For (run the loop 10 times)

{

If (number entered is greater than the number Array[index])

Add the Array[index] to the value in sum and store the result in sum (i.e. sum += Array[i])

Else

Jump to next iteration and check next number in array

}

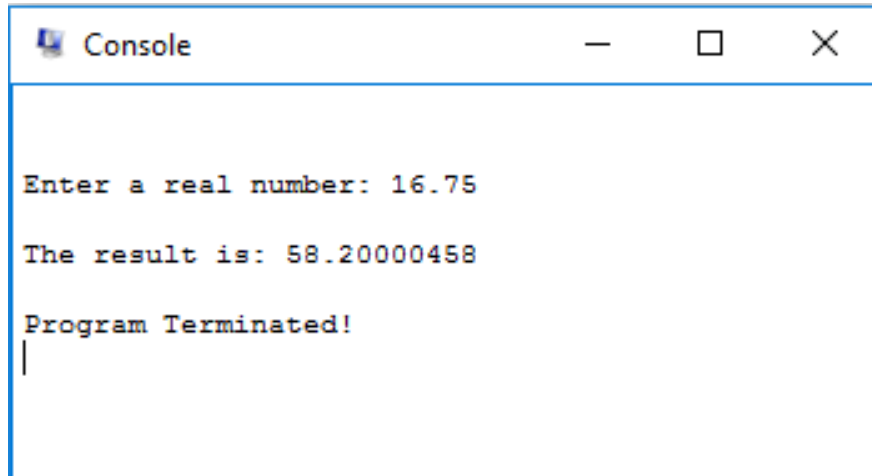
If (sum is zero)

Display the number is greater than the numbers array

Else

Display the sum

Output:

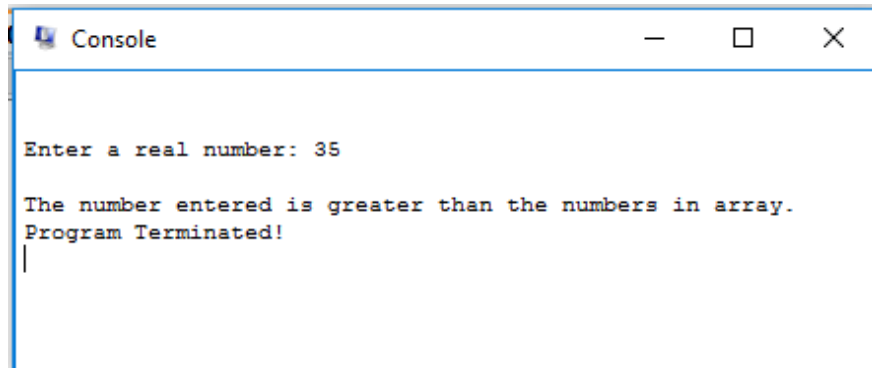


```
Console

Enter a real number: 16.75

The result is: 58.20000458

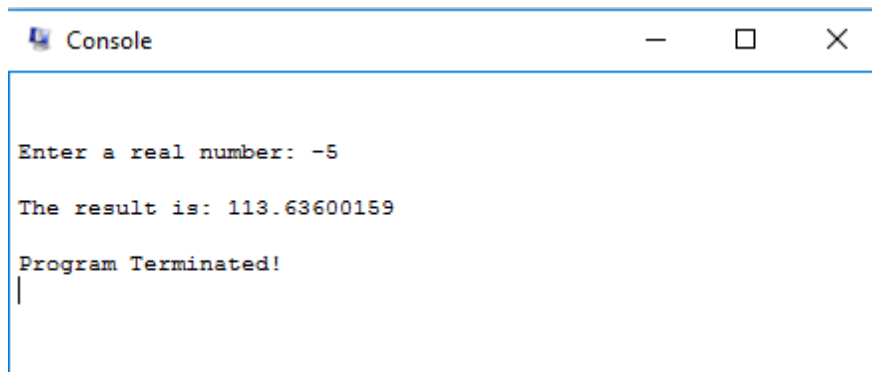
Program Terminated!
|
```



```
Console

Enter a real number: 35

The number entered is greater than the numbers in array.
Program Terminated!
|
```



```
Console

Enter a real number: -5

The result is: 113.63600159

Program Terminated!
|
```

Conclusion:

The output has been displayed for each scenario. The array has been stored in memory, and then using instructions for single point precision floating point the numbers have been loaded from the array to floating point register.

Conclusion:

Floating point instructions has been used for single precision i.e. for reading and printing floating points, arithmetic operations on floating points by using instructions like *add.s sub.s mul.s div.s*, for moving or storing the floating point numbers from one register to another by using instructions like *li.s, mov.s, l.s*, the comparison instructions like *c.xx.s* where *xx* can be *le, lt* that sets a conditional bit in the coprocessor1 to 1 (true) or 0 (false) depending upon the instruction and condition used, and then *bc1t, bc1f* are used as branch instruction to jump to specific address depending upon if the conditional bit (which is stored in FCCR) is true or false, respectively.
