x86-64 Assembly Calculator Report

By Thet Paing Zaw

Source Code:

Purpose:

The purpose of this lab is to incorporate the majority of concepts we have previously learned in this class. Instead of using integers like we have been using throughout this semester, we were to practice using floating-point numbers with certain assembly conversion instructions while also performing different arithmetic operations effectively on floating-point numbers. In addition, this lab requires students to be able to create a calculator for the user to input numbers and do arithmetic operations and find the maximum of the inputted numbers.

Planning and organization:

To write an effective program we had to plan and organize well from the beginning, which not only included just starting immediately to write up the code for it but also ways of communicating and organizing. We had agreed to meet up a couple of times throughout the week to tackle parts of the program together.

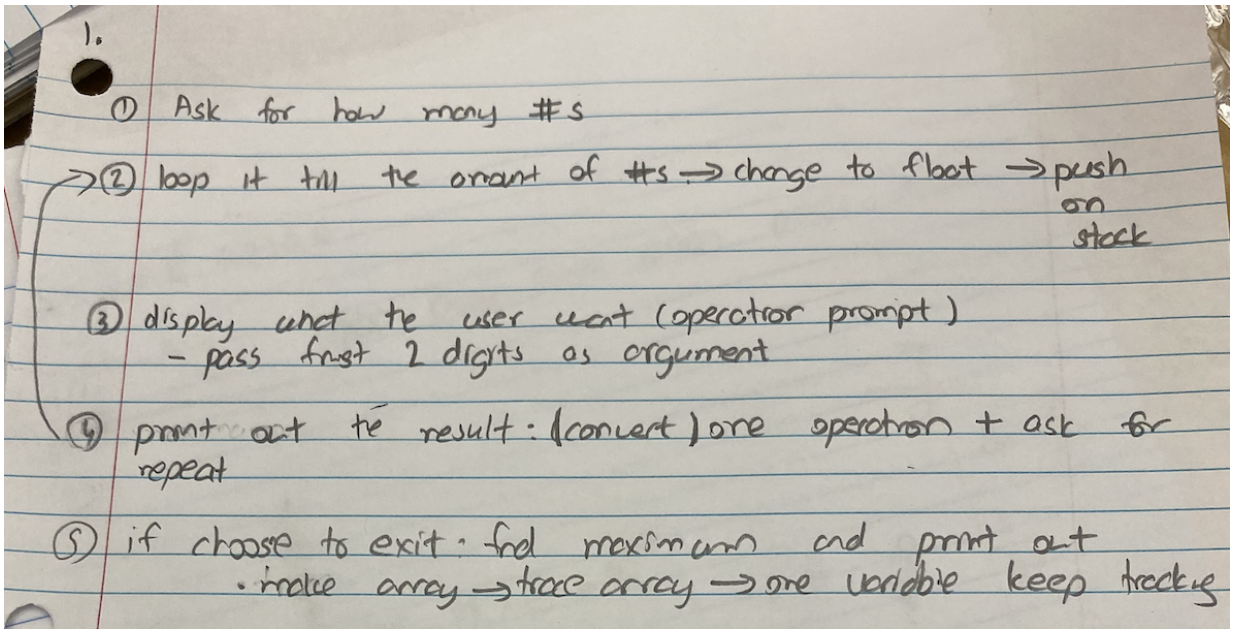

Figure 1: Planning of parts we were going to do for the project

- Initial planning:
  - ➔ We initially made a rather simple list of the things we would need to complete for the program (as shown above).

➔ The first two parts were completed between when our classes meet, and we were actually a little ahead of schedule since we initially said we would only complete one on the list.
➔ The next part was meeting up during the Thanksgiving break holiday to work together on completing the rest of the program and write out the functions and algorithms we should use together.

- Take up 5 floating-point numbers from the user
  ➔ ask the user for the amount of numbers to input
  ➔ store the amount inputted and loops it to take in the inputs

- Convert hex input into actual floating-point numbers onto the stack

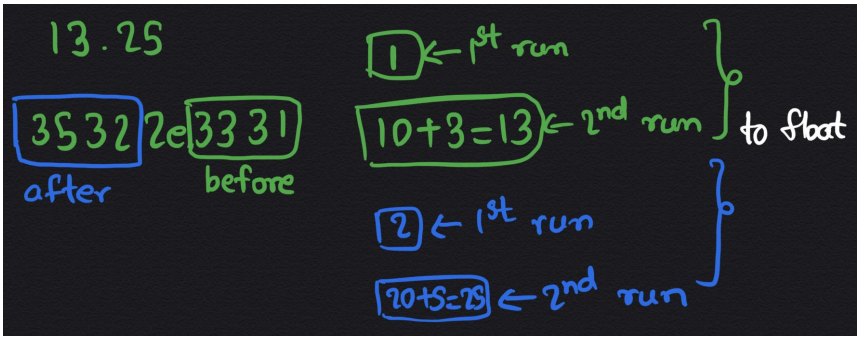

Figure 2: converting from ASCII to hex

➔ we first start by separating the integer part and decimal parts of the user input
➔ We will first subtract 30 from ASCII to integer and do multiplication with 10 repeatedly using loops to get the decimal value of the integer part

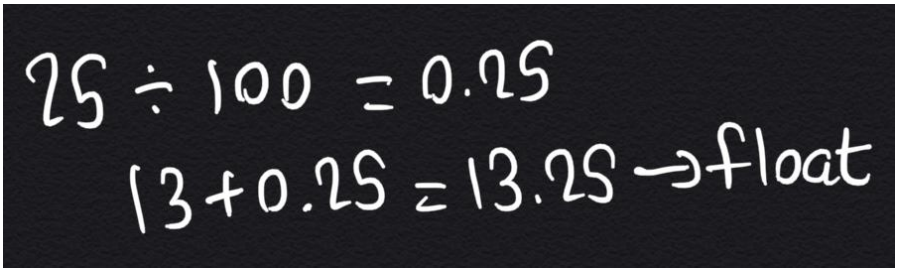

Figure 3: Getting the final float number

➔ While looping for the decimal part, we will need to keep a counter. And after that, we can move both the converted integer part and the decimal part to “xmm” registers to change into float. We need a counter while processing the decimal part because when we have converted the

decimal part it would be in whole number representation. We then need to divide (depending on the counter) with powers of ten so we can get the actual decimal value.
    - ➔ Then we can simply add the decimal part and integer part together and the result will be the float we can use for our calculations
    - ➔ Case 1: negative numbers--we first need to have a flag to check for negative sign ASCII and using a flag, we can process the same way as we process our positive numbers and at the end of the convention, we simply multiply with negative one (-1) to change from positive to negative float.
    - ➔ Case 2: for other user input validation, we will process while doing the conversion using conditional jump statements to compare for anything other than numbers

- Allow the user to choose arithmetic operations using the first two numbers from the user input
    - ➔ We have four choices for the user: addition, subtraction, multiplication, and division. The user will need to input 1, 2, 3 or 4 respectively in order for our program to perform the operation. We can check the user input by using conditional jump statements and then, the program can jump to the operations the user wants to perform.
    - ➔ After we have obtained numbers from the user, they will be stored on the stack. So we can simply access the stack and use arithmetic operation commands of xmm registers and get the results

- Continue until the user decides to quit the program
    - ➔ The program will continue to loop through and continues to do operations with the inputted data as long as the user decides to choose anything other than given options

- Finding a maximum number from all floating-point inputs:

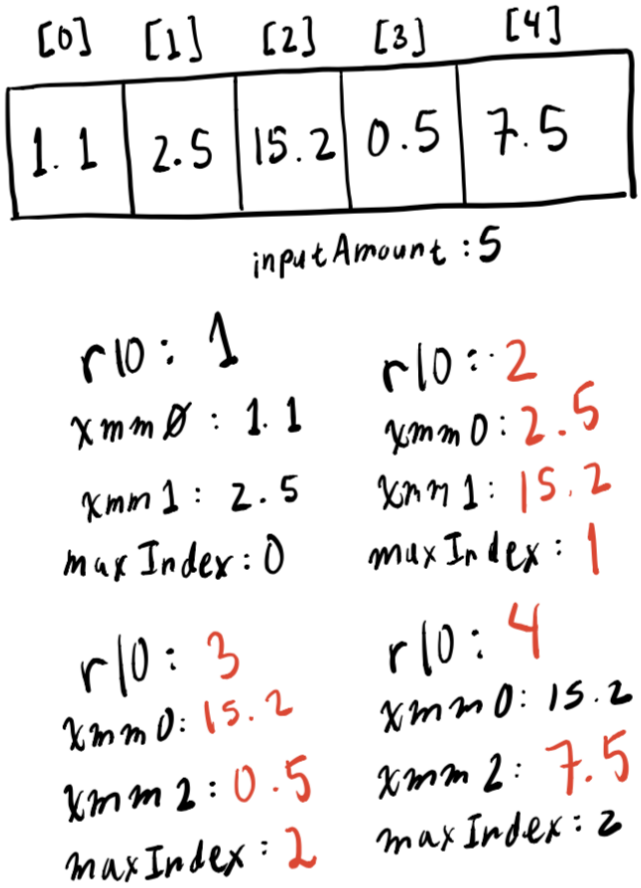

Figure 4:Algorithm to find maximum

➔ We stored all the float inputs from the user into an array on the stack, where index 0 would start at offset of -40 from %rbp. Using an array was thought out way before any coding started, as we all decided it was going to be the most effective way of storing and retrieving our values when having to do such things like looking for max or min or sorting and simply just walking through and checking each input value.

➔ Utilizing registers for xmm0 and xmm1, we iterate through the array looking for maximum as depicted in the above image (red value means there was a change in that field).

➔ A comparison is done on the two registers and if xmm1 is greater than the value in xmm0, then at that point we update the maximum index position to the current xmm1 index that holds the bigger value and also set the new maximum value into xmm0, then continue the loop again.

- ➔ If xmm0 is still holding the biggest value, as shown in the fourth iteration, then there is no change to xmm0 or maxIndex and we continue with our loop until we hit the end of array.
- ➔ Once the loop is completed and the resulting maximum index position is stored in our variable of maxIndex, all there was left to do was to give the offset of where the maxIndex position in our array is to get that floating point, convert it to ASCII, and output it.

- Converting from float to ASCII: We planned different ways to convert from float to ASCII most of which had some problems discussed later in pitfalls section:
  - ➔ Firstly we tried to truncate the whole part of the number and subtract it from the original to get the whole part and fractional part in different registers. But, the value was rounding off instead of being truncated
  - ➔ Second, we tried to transfer the hexadecimal value of the float to just convert it to ASCII, but it was stored in IEEE format, so we could not use that either.
  - ➔ Lastly, we thought about converting the IEEE format in hex to decimal format, just as we did in class. We thought about using bit shifting to get the different parts: sign, exponent, and mantissa.
- Adding input validation for the number of digits, negative numbers, and precision error.
  - ➔ data validation to check user only enters numbers Ex: 1a.2b.3c
  - ➔ data validation to check for invalid form of input Ex: 1.2.222
  - ➔ Handling zeros for both user inputs and results
  - ➔ Check for dividing with zero

  All these will result in an error message and prompting the user to enter again.

Product:

- **Program Runs**

```
Please input number of floats you would like to enter (min of 2 and max of 5): 3

Enter your floating point values (8 characters maximum): 4.3
You have a precision error.
1.4
You have a precision error.
1.5
```

Figure 5:Taking multiple inputs from the user and displaying a precision error if any

```
Choose one option
1. Addition
2. Subtraction
3.Multiplication
4. Division
or any key to quit program: 1

Your result is: 5.700000
```

Figure 6: Output of the first two float addition (4.3+1.4=5.7)

```
Choose one option
1. Addition
2. Subtraction
3.Multiplication
4. Division
or any key to quit program: 2

Your result is: 2.900000
```

Figure 7: Output of the first two float subtraction (4.3-1.4=2.9)

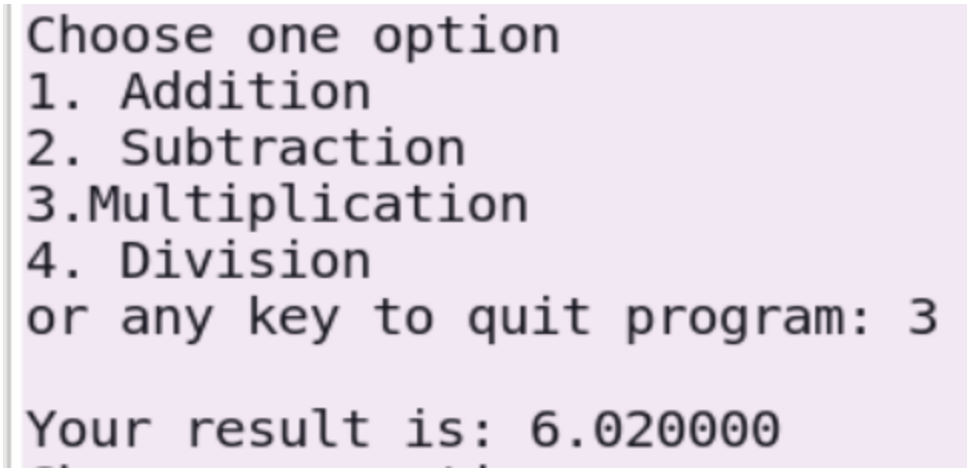

Figure 8: Output of the first two float multiplication(4.3*1.4=6.02)

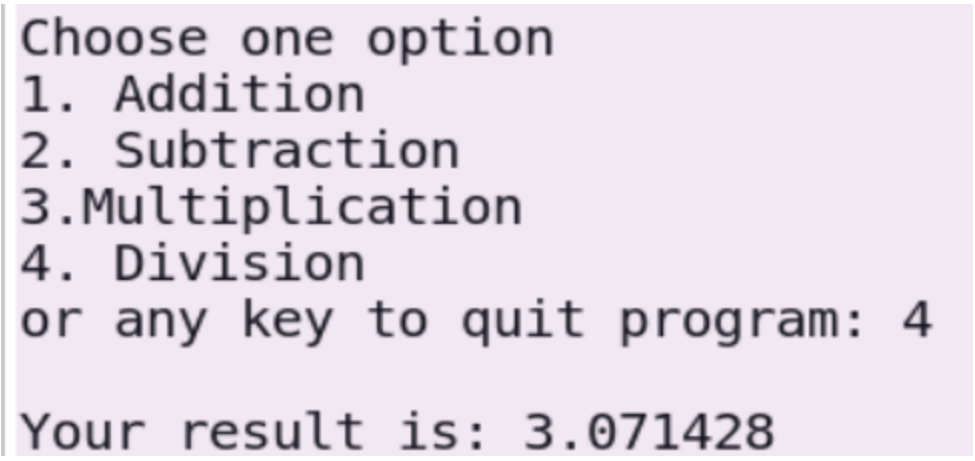

Figure 9: Output of the first two float multiplication(4.3/1.4=3.071428)

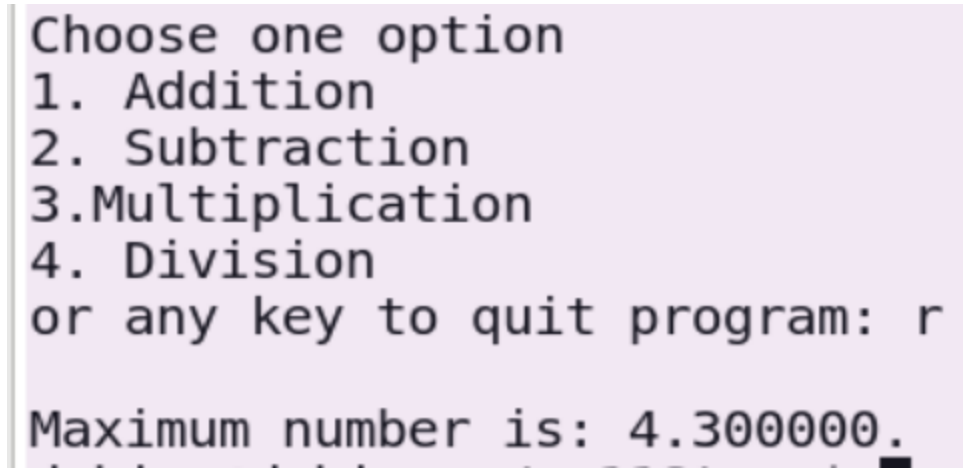

Figure 10: The user quits the program and output the maximum number of inputs

Call Stack Changes

- At the beginning of our program, since we did not have any data inputted yet, our call stack is empty as shown in the Figure below.

```
(gdb) x/11xg $rsp
0x7fffffffe1f0: 0x0000000000000000      0x0000000000000000
0x7fffffffe200: 0x0000000000000000      0x0000000000000000
0x7fffffffe210: 0x0000000000000000      0x0000000000000000
0x7fffffffe220: 0x0000000000000000      0x0000000000000000
0x7fffffffe230: 0x0000000000000000      0x0000000000000000
0x7fffffffe240: 0x0000000000000000
(gdb)
```

Figure 11:Call stack at the beginning

- We then ask the user for input

```
Please input number of floats you would like to enter (min of 2 and max of 5): 4

Enter your floating point values (8 characters maximum): -1.2
You have a precision error.
2.2
You have a precision error.
0.5
5.6
You have a precision error.
```

Figure 12:Getting input to run through using gdb

- The converted ASCII results are stored at offset of rbp-88 and the sign is stored in the offset next to it at rbp-80 as shown in the figures below.
- We store the new results from all the arithmetic operations at the same offsets along with the result for maximum.

```
(gdb) x/4xg $rbp-88
0x7fffffffe1f0: 0x0000000000000000      0x3030303030302e31
0x7fffffffe200: 0x000000003f800000      0x0000000000001f80
(gdb) i r rax rdx
rax            0x3030303030302e31       3472328296227679793
rdx            0x0      0
(gdb) c
Continuing.

Your result is: 1.000000
```

Figure 13:Addition result stored in registers and call stack

```
(gdb) x/4xg $rbp-88
0x7fffffffe1f0: 0x000000000000002d      0x3030303030342e33
0x7fffffffe200: 0x00000000c059999a      0x0000000000001f80
(gdb) c
Continuing.

Your result is: -3.400000
```

Figure 14:Subtraction result stored in registers and call stack

```
(gdb) x/4xg $rbp-88
0x7fffffffe1f0: 0x000000000000002d      0x3030303034362e32
0x7fffffffe200: 0x00000000c028f5c3      0x0000000000001f80
(gdb) c
Continuing.

Your result is: -2.640000
```

Figure 15:Multiplication result stored in registers and call stack

```
(gdb) x/4xg $rbp-88
0x7fffffffe1f0: 0x000000000000002d      0x3435343534352e30
0x7fffffffe200: 0x00000000bf0ba2e9      0x0000000000001f80
(gdb) i r rax rdx
rax            0x3435343534352e30       3761970466851728944
rdx            0x2d     45
(gdb) c
Continuing.

Your result is: -0.545454
```

Figure 16:Division result stored in registers and call stack

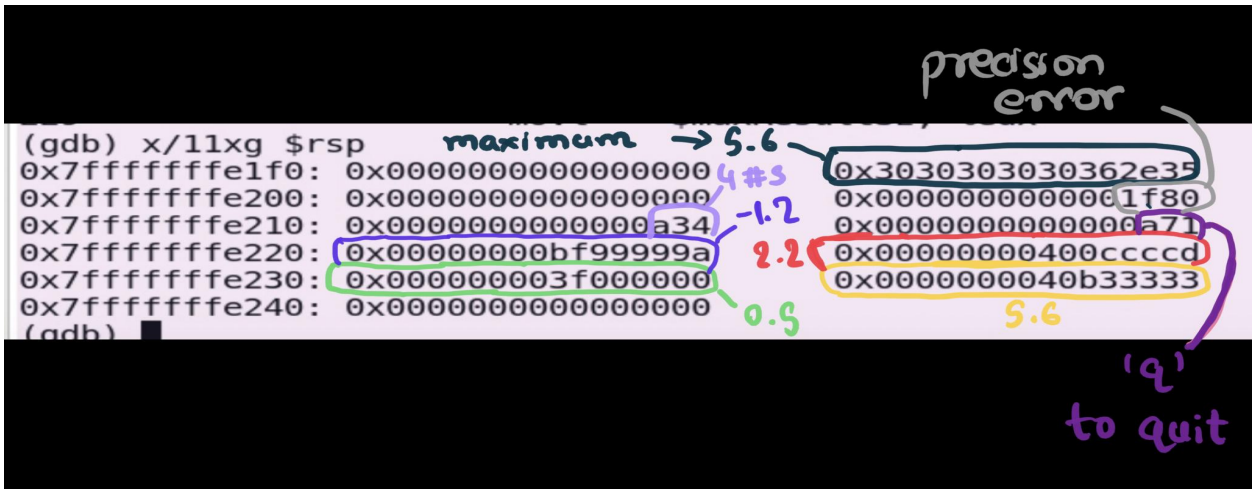

Figure 17:Call stack with data

RIP changes

- Initially stepping into a function from the main, we have rip as 0x4001e5

```
165                             call    addition
(gdb) i r rip
rip            0x4001e5 0x4001e5 <additionfun>
(gdb)
```

Figure 18:checking RIP before calling addition function

- After stepping into the function, before returning back to the main, we can see the rip in the function(0x40043f) which is very different compared to the rip in the main(0x4001e5). We can also see in the call stack of the function, the rip of the return is stored (0x4001ea) which is 5 bytes increment from our previous main rip 0x4001e5.

```
357                         ret
(gdb) i r rip
rip            0x40043f 0x40043f <addition+12>
(gdb) x/3xg $rsp
0x7fffffffe1b8: 0x00000000004001ea      0x0000000000000000
0x7fffffffe1c8: 0x0000000000000000
```

Figure 19:Call stack before returning back to main

- And when we have returned to the main, we can now see that the rip is now changed into (0x4001ea) which was previously stored in the call stack of the function

```
166                             movss   %xmm0, -72(%rbp)
(gdb) i r rip
rip            0x4001ea 0x4001ea <additionfun+5>
```

Figure 20:RIP after returning back to main function

Input validations:

- Validation for invalid number of float numbers: If the number of digits trying to enter is more than 5 or less than 2 then we provide an error message about it and prompt users to enter again

```
debian@debian:~/teamproj$ ./team

Please input number of floats you would like to enter (min of 2 and max of 5): 7

You entered less than 2 or more than 5 numbers.
Please input number of floats you would like to enter (min of 2 and max of 5): 1

You entered less than 2 or more than 5 numbers.
Please input number of floats you would like to enter (min of 2 and max of 5):
```

Figure 21:Input validation for number of floats

- Validation for precision error: If there was any precision error during calculation we manually check for it and print an error message for that error

```
Please input number of floats you would like to enter (min of 2 and max of 5): 2

Enter your floating point values (8 characters maximum): 1.22
You have a precision error.
```

Figure 22:Input validation for precision error

- Validation for floating-point numbers: If the user enters anything other than floating-point number or characters in between numbers, then we provide an error message and prompt the user to enter the number again.

```
2.3.4

You entered something other than numbers or did not enter a floating point number.
Enter again:
```

Figure 23:Input validation for anything other than floats

Conversion algorithm

- Float to ASCII algorithm: The steps we followed in creating the algorithm for conversion, for example, 1.5 is stored as 0x3fc00000
  - ➔ First of all, we figured out how many places we need decimal. For that, we divided the number by 10 until it became less than 1 and counted how many times we divided. Then subtracted it from 7 because we have 7-digit printing precision.

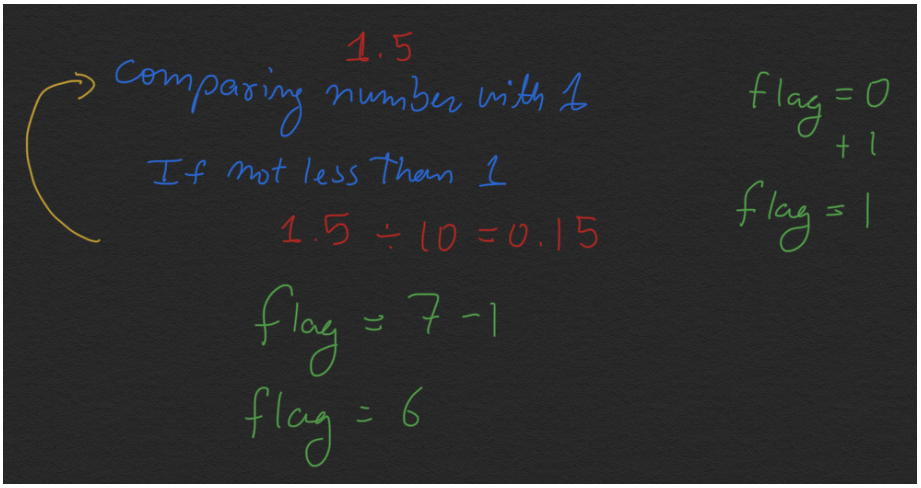

Figure 24:Getting number of decimal places to shift

➔ To get rid of the sign bit we shifted 1 bit to the left

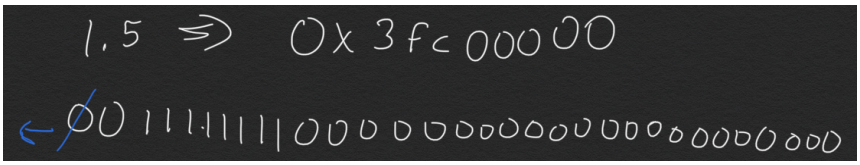

Figure 25:Removing the sign bit

➔ Next, we moved the number left to another register (rcx) and shifted right 24 times to get the 8-bit exponent

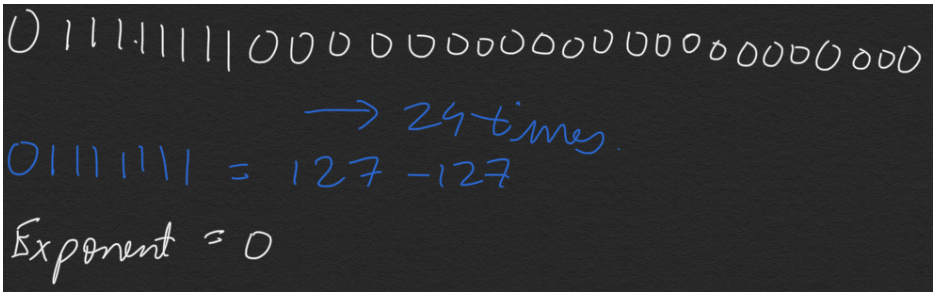

Figure 26:Getting the exponent

```
#calculating the exponent
        shll    $1, %ebx
        movq    $0, %rcx
        movl    %ebx, %ecx
        shll    $8, %ebx
        shrl    $24, %ecx
        subl    $127, %ecx
        movl    %ecx, deccount
```

Figure 27:Code to find the exponent

Above is the code portion for figures 25 and 26 and after we shifted right 24 bits, we get the exponent by subtracting the exponential bias.

➔ Then from the original value stored in ebx I shifted 8 bits left to get the mantissa part. And then shifted left one bit at a time and jumped if there is carry flag. Along with that loop we multiplied to 2 to a register and whenever there is carry flag, we divided by the power of 2.

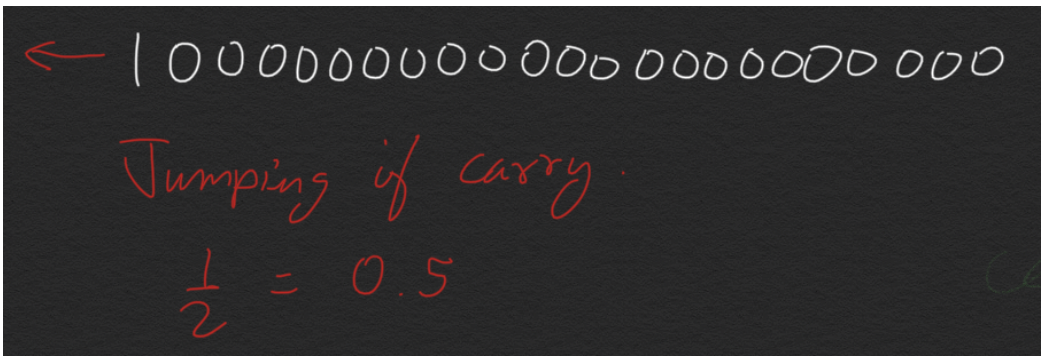

Figure 28:Shifting bits to find mantissa

➔ After converting the mantissa part we add 1 to the fraction and then multiply by 100000000 to make the number at least 7 digits

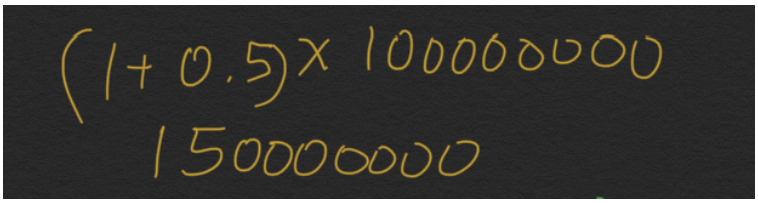

Figure 29:Getting the number

➔ Then we multiply the resulting number by 2 the number of times equal to the exponent, but in this case, the exponent is zero so we do not multiply by 2.

➔ After this step, we check if the number is 7 digits, if not, we divide it by 10 until it becomes 7 digits number because we can only print 7 digits in total

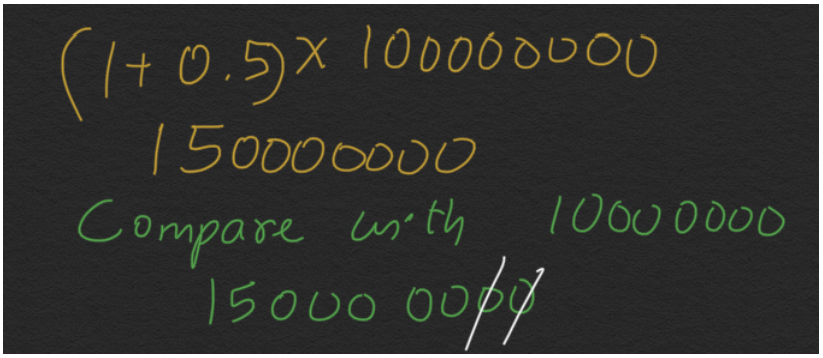

Figure 30:Reducing the number to 7 digits

➔ In the next step we convert the number to ASCII by the method we used in previous labs, i.e. dividing by 10 and adding 0x30 to the numbers. However, in this case, because we have floating point numbers so we have to put a decimal point in printing. So, we check the flag we calculated before to count after how many places we need decimal.

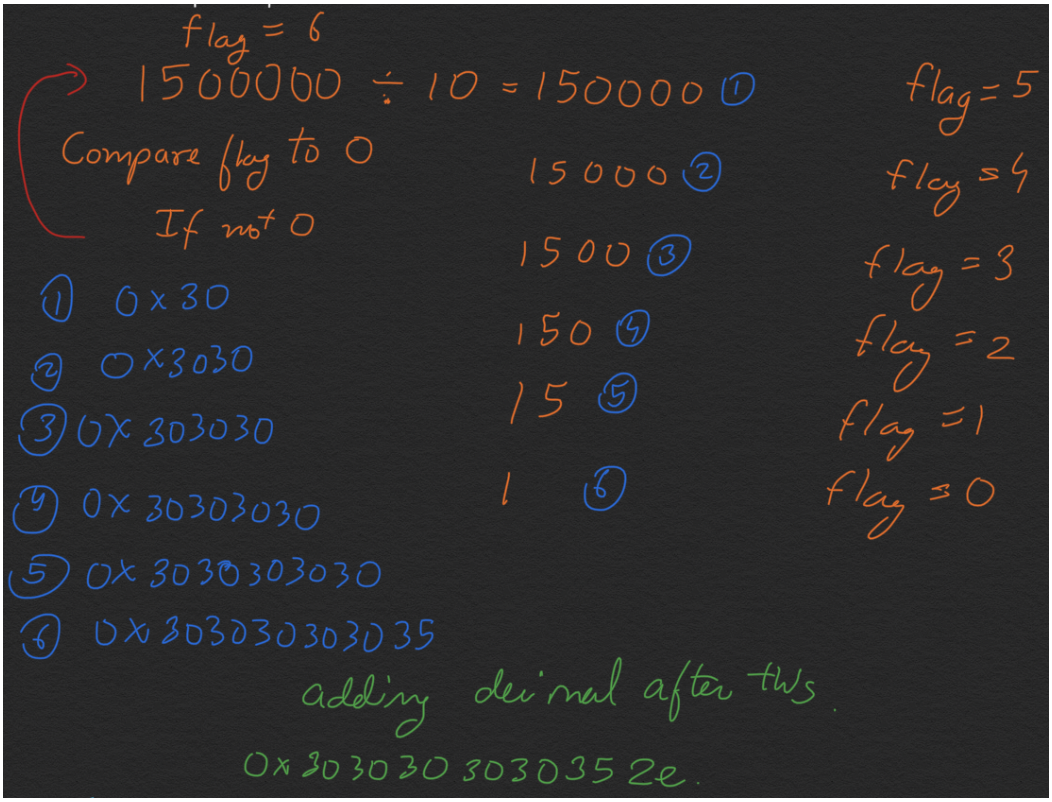

Figure 31:Converting from decimal to ASCII to print and adding decimal

➔ After adding the decimal, the last step is to add the remaining whole part and print.

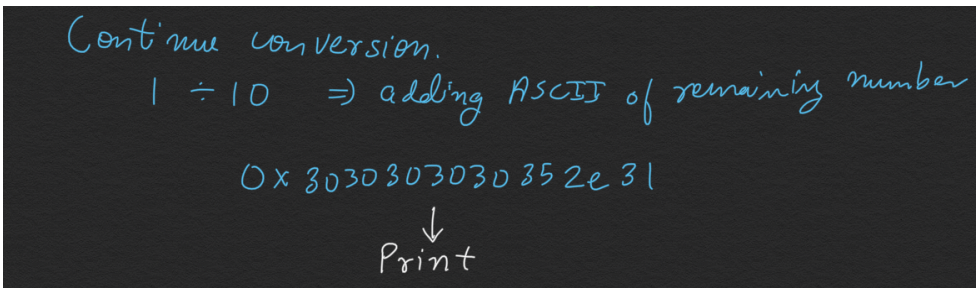

Figure 32:Converting the whole part and adding it to the ASCII string and printing

Pitfall:

- Initialization registers
  - ➔ Because of not initializing the xmm registers and using the same ones for the next calculation, it was giving the wrong result.

- Failure to mention the precision error of the floating-point number
  - ➔ Also, mxcsr flags operating differently than the r-flags were used to. So we had to look up why we were seeing multiple-precision error outputs, but thanks to the internet we were able to find out that the mxcsr flags do not change back by themselves.
  - ➔ Resolution was to use bitwise and operation on the fifth bit where the PE flag was turned on without changing the bits of the other flags, then load it back so that next time PE flag would be turned on and invoke our precision error output correctly instead of spitting it out repeatedly.
- Not being able to convert from float to ASCII properly due to:
  - ➔ IEEE format is used for floating-point numbers which didn't cross our minds until later.
  - ➔ Using one of the conversion instructions which would start rounding and effectively make the result incorrect when we would want to output it.

- RIP Error

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000000001 in ?? ()
1: /x $rip = 0x1
```

Figure 33: Rip giving segmentation faults and changing into 0x1

➔ When we are making a function for multiplication operation, whenever we run we kept getting a segmentation fault. So we went through line by line using GDB then we found error shown in Figure 33. Everything was working the way we expected until we were about to exit the program. The very first assumption we made was our virtual machine must have run out of memory. But we tested on all three computers and they all showing the same error which is very suspicious. Then we started to look for possible syntax errors in the code.

```
movq    %rbp, %rsp
popq    %rsp
ret
```

Figure 34: Mistake on popping %rsp instead of %rbp

➔ After going through for hours we found, we have popped %rsp instead of %rbp in the function epilogue. From this mistake, we learned that we should carefully check every single line of code even the code we are very familiar and think that we know very well.

- Only being able to print until 7 digits of result

```
Enter your floating point values: 10000.0
1000.0

Choose one option
1. Addition
2. Subtraction
3.Multiplication
4. Division
or any key to quit program: 3

Your result is: 9999999
Choose one option
1. Addition
2. Subtraction
3.Multiplication
4. Division
or any key to quit program:
```

Figure 35:Only being able to print 7 digits because of error in precision

➔ Because we can only print until 7 digits (which is further discussed in the next section), when we multiply 10000 with 1000 the result should be 10000000, but this is an 8 digit values, so instead of the program prints the approximate number i.e. 9999999, which is almost correct.

Possible Improvements:

- Only being able to print until 7 digits of float: When performing arithmetic on 32-bit floats, due to precision error, the results were right only until 7 digits because of the conversion from Ascii to float. When converting from float to Ascii we can only convert 7 digits because of calculation errors after the 7th digit. Also, a register can hold only up to 8 ASCII characters and one of them is the decimal, so we were only able to print out 7 other digits including both before and after the decimal. If there was a more efficient way to avoid precision errors, we would have been able to print more digits.

```
.data
  num1: .float 992345.55
```

```
(gdb) i r rax
rax            0x5a40d8800000   99234556674048
(gdb)
```

Figure 36:Error in the result after 7 digits

For example, in this test program that we made for converting float to decimal, the float value “992345.55” was converted to decimal without any decimals. “rax” register contains the decimal value, which is “992345566..” (we multiplied the float number by some factor to remove the decimal), but the value of float goes off after 7 digits with some junk numbers, so we decided to take only until 7 digits.

- User Input Validation: We tried to think of as many cases as possible in order to validate the user data inputs as shown in the products section. But in the real world, user data can vary significantly and can be anything so we believe that there are still many more cases we need to consider to improve data handling for our program.