Lebanese American University

A close up of a logo

Description automatically generated

COE3232 - 10AM Section

Dr. Zahi Nakad

Project: Floating Point

Hisham 201800953

Charbel Bou Maroun 201800430

Abed

Hazem

May 11, 2020

# Introduction

In this project, we will create an assembly code that is able to interpret and decode floating point numbers and perform multiplications. We will use the Easy68K IDE to assemble and run our 68000 program.

# Table of contents

[Introduction 2](#_Toc40001956)

[Table of contents 3](#_Toc40001957)

[Table of figures 4](#_Toc40001958)

[Assembly Code 5](#_Toc40001959)

[Subroutine 1: 5](#_Toc40001960)

[Subroutine 2: 6](#_Toc40001961)

[Conclusion 7](#_Toc40001962)

# Table of figures

[Figure 1: Half-Wave Rectifier. 5](#_Toc36459032)

[Figure 2: PSPICE circuit of a half wave rectifier. 5](#_Toc36459033)

[Figure 3: Simulation of the half-wave rectifier. 6](file:///C:\Users\Intel\Desktop\LAU_Courses\Electronics_I%20_Lab\Lab4\Lab4.docx#_Toc36459034)

[Figure 4: Peak of the half-rectifier signal. 6](file:///C:\Users\Intel\Desktop\LAU_Courses\Electronics_I%20_Lab\Lab4\Lab4.docx#_Toc36459035)

[Figure 5: PSPICE circuit of the half-wave rectifier with a 1μF capacitor. 7](#_Toc36459036)

[Figure 6: Simulation of the half-wave rectifier with a 1μF capacitor. 8](#_Toc36459037)

[Figure 7: Simulation of the half-wave rectifier output with a 1μF capacitor. 9](#_Toc36459038)

[Figure 8: PSPICE circuit of the half-wave rectifier with a 10μF capacitor. 10](#_Toc36459039)

[Figure 9: : Simulation of the half-wave rectifier with a 10μF capacitor. 10](#_Toc36459040)

[Figure 10: Simulation of the half-wave rectifier output with a 10μF capacitor. 11](#_Toc36459041)

[Figure 11: PSPICE circuit of the half-wave rectifier with only a 10μF capacitor as the load. 12](#_Toc36459042)

[Figure 12: Simulation of the half-wave rectifier with only a 10μF capacitor as the load. 12](#_Toc36459043)

[Figure 13: Screenshot of Vs (Red) vs Vo (Green) 14](#_Toc36459044)

[Figure 14: Screenshot of Vs (Red) and Vo (Green) with 1uF Capacitor 14](#_Toc36459045)

[Figure 15: Screenshot of Vs (Red) and Vo (Green) with a 10uF Capacitor 15](#_Toc36459046)

[Figure 16: Screenshot of Vs (red) and Vo (green) when the Resistor is removed 16](#_Toc36459047)

[Figure 17: Screenshot of Vo without the resistor, for a long time 16](#_Toc36459048)

# Assembly Code

## Subroutine 1

This subroutine dissects a 32-bit floating point number where the sign consists of 1 bit and the number of exponent bits is given. It provides the separate components of a floating point number in a word for the sign, a long for the exponent, and a long for the mantissa.

The inputs are the 32 bit floating number and the number of exponent bits and, subsequently, the outputs would be the sign, exponent and mantissa, their respective addresses are provided as such:

PEA Input

PEA Exponent\_Bits

PEA Sign

PEA Exponent

PEA Mantissa

Thus, the input of our subroutine 1 is located on the stack just before calling the subroutine: BSR Subroutine\_1:

We will create a small stack frame of size 4 at the start of our subroutine and have A0 as its pointer, we then stored the used registers on the stack: MOVEM.L D1-D4/A1,-(SP)

At the start of our subroutine:

D1 will store the original number

D2 will store the sign of the number

D3 will store the number of exponent bits

D4 will store the number of mantissa bits

A0 will be our stack frame pointer

A1 will be used to load our components

First, we move our 32 bit floating number to the register D1 and check if the number is positive or negative (Since the MSB dictates the sign of the floating point number). If it’s negative, we store a ‘1’ in D2, if not we skip this step.

We then load the number of exponent bits to D3, additionally, we get the number of mantissa bits in D4 by subtracting the number of exponent bits from 31 (Since it’s a 32 bit floating point number and the sign consists of 1 bit).

After loading our essential components, we now have the sign bit, therefore we can store it in memory.

We now need to get the bias, to do so, we will now re-use the register D2 since we’ve already stored its previous value (sign) in memory. D2 will store the bias number:

1. We first subtract 1 from D3 (Which now stores the number of exponent bits - 1)
2. Then we override the entire D2 register by a literal 1
3. After that we perform an arithmetic shift left to the entire D2 register by the number of exponent bits – 1 which is stored in D3.
4. Finally, we subtract 1 from D2 and get our bias :

//We then store the bias in our stack frame

After that, we store our original number that is stored in D1 onto the stack in order to preserve it, since it’s now time to get our mantissa:

1. We start by adding 2 to D3 (Which had the number of exponent bits -1), D3 now stores the number of exponent bits + 1 (Which includes the sign)
2. We arithmetically shift left our original number in D1 by the number of exponent bits + 1 which is stored in D3.

We now have the mantissa stored in D1, which we will store in the corresponding memory location.

Since D1 (Where the original number was stored) was destroyed, we get our original number back from the stack where we originally placed it before getting our mantissa:  
D1 now has the original number.

The final step of this subroutine is to get our exponent. We know that the biased exponent is stored 1 bit to the right of the MSB (After the sign bit) of our 32 bit floating point number, we will need to isolate that number and remove the bias:

1. We set the first bit of our 32 bit floating point number to 0 by using the using the function: AND.L #$7FFFFFFF,D1 where our original number is stored
2. The first step was necessary since we’re going to arithmetically shift right our number and we know that using this function in the 68000 language would cause problems if the MSB is set to 1 since it will push a 1 to the right every time which would distort our exponent. We shift right by the number of mantissa bits (Stored in D4).
3. We now subtract the bias from D1 (Where the biased exponent is stored).

Having done the last steps, we store our exponent (stored in D1) in the corresponding memory location.

All the outputs have now been supplied; we return the registers as they were before the subroutine by reusing the MOVEM.L function: MOVEM.L (SP)+,D1-D4/A1

Finally, we unlink the stack frame that was created.

The registers are now back to their initial value, for a better code, we return the stack pointer to its initial location in our main code: LEA ($14,SP),SP

## Subroutine 2

This subroutine calculates the product of two floating point numbers provided from Main.

The inputs are two 32 bit floating numbers and the number of exponent bits and, subsequently, the outputs would be the product, their respective addresses are provided as such:

PEA Input\_1

PEA Input\_2

PEA Exponent\_Bits

PEA Product

These addresses are provided in the stack before calling subroutine 2.

In this subroutine, we’ll have to call subroutine 1 which we created in the last part. The thing we should look out for is the correct input format for subroutine 1 which is:

PEA Input

PEA Exponent\_Bits

PEA Sign

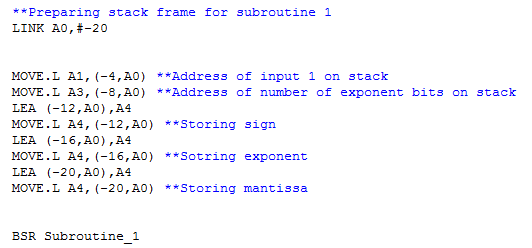
PEA Exponent

PEA Mantissa

Therefore, in order to make the subroutine 1 function properly, we will have to have these inputs before calling subroutine 1 inside subroutine 2. In order to do so, we will create a new stack frame of size 20 (5 longs) which will act as the input for our subroutine 1 calls.

Just before that step we will store our used registers on the stack using the MOVEM function.

We will get the input and the number of exponent bits from the original inputs of subroutine 2, but for the addresses of “sign”, “exponent” and “mantissa” which have not been provided in the main code, we will store them directly on the stack frame (The content of the stack frame at “sign”, “exponent” and “mantissa” inputs will be their own location on the stack).

**Ex:**

Storing the address of itself on the stack.

Subsequently, after running the above code, the sign, exponent and mantissa of input 1 will respectively be stored in : (-12,A0), (-16,A0), (-20,A0) on the stack frame.

We’ll do the same for input 2 (Having A5 as its respective stack frame pointer), and we will store all of our subroutine 1 results in the following registers:

* D0: Sign of input 1
* D1: Exponent of input 1
* D2: Mantissa of input 1
* D3: Sign of input 2
* D4: Exponent of input 2
* D5: Mantissa of input 2

After having sliced our inputs, it is now time for the multiplication logic.

We’ll first get the sign by using the exclusive or function between D0 and D3.

After that, we can add the exponents since it’s a multiplication, we’ll add D1 to D4.

Now we aren’t yet sure that D4 is our final answer for the exponent value of the multiplication since it could increase by 1 due normalization of the mantissa.

For the mantissa, we know that they are stored from the most significant bit downwards (Based on what was needed in subroutine 1), the process will be as such:

1. We first need to add the hidden mantissa bit, in order to do so we need to arithmetically push to the left of the mantissa a ‘1’ : this is done by arithmetically shifting the mantissa to the right by 1 and using the OR function with #$80000000 which will make the newly added bit a 1.
2. Since we’re going to perform a multiplication, it will have to be done between two **WORDS**, therefore, we will move our mantissas to the word position by arithmetically shifting right by 16 bits. (We do not care about the 1’s that will be added on the left since they will be overridden during the multiplication).
3. Then, we will multiply both modified mantissas in D2 and D5 and store the result in D5. Here we will have to check if the result needs to be normalized which will push our mantissa ‘dot’ to the left and we’ll have to add 1 to the exponent:

Some concepts regarding the multiplication: we are multiplying two 16 bit numbers, and we consider them to have the format:

Hidden ‘1’ bit

15 bits of the mantissa after the dot.

We know that there are 15 bits after our ‘imaginary’ dot, and that we are multiplying two 16 bit numbers which will lead to a number which has 30 bits after the ‘imaginary’ dot.

To know if our product needs normalization we just check if the MSB of our register (The 32nd bit) is set to ‘1’, the number would look like:

30 bits of the mantissa after the dot.

Therefore, we know our mantissa should include the 31st bit and the exponent should be incremented by 1. If that MSB bit was set to 0, then we know that our mantissa is just made out of the 30 bits after the ‘imaginary’ dot.

If normalization was needed, this code would run: 

Where D4 stores the exponent and after shifting left our register D5 would have the correct mantissa.



If not, then this code would run:

Where D5 would be shifted to the left by 2 leaving the correct mantissa.

Having done the multiplication logic, we will have to reconstruct the product from its sign, exponent and mantissa into its hexadecimal 32 bit floating point representation and store it as the output of the subroutine in memory.

For that we will need the number of exponent bits (which we have), the number of mantissa bits and the bias:

# Conclusion

In conclusion, one of the main uses of diodes is rectifier circuits. As we have seen during this lab, two basic configurations are the half-wave and full-wave rectifiers. Each has unique specifications and use cases, additionally, we’ve learned how adding a capacitor in parallel to our load or even removing the load resistance all together affects our output (To ultimately reach as close to a horizontal output as possible).