

Lebanese American University



COE323 - Microprocessors - 10AM Section Group 5

Instructor: Dr. Zahi Nakad

Project: Floating Point

Hisham Atieh (201800953)

Charbel Bou Maroun (201800430)

Abed Dandan (201801045)

Hazem Daher (201703266)

Date of Submission: May 13, 2020

Table of contents

Table of contents.....	1
Table of tables	2
Introduction	3
Assembly Code	4
Subroutine 1	4
Explanation.....	4
Stack	7
Subroutine 2	8
Explanation.....	8
Stack	13
Appendix A	14
<i>Project Code:</i>	14

Table of tables

Table 1: Stack while performing subroutine 1.....	7
Table 2: Stack while performing subroutine 2.....	13

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.

Assembly Code

Subroutine 1

Explanation

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 :

$$Bias = 2^{number\ of\ exponent\ bits - 1} - 1$$

After that, we store our original number that is stored in D1 in the stack frame 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 frame 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`

Stack

This is how the stack looks like while performing subroutine 1:

Registers stored using MOVEM	}	Stack frame
Original Input		
Address A0 after LINK		
Return address of BSR		
Mantissa (address)		
Exponent (address)		
Sign (address)		
Number of exponent bits(address)		
Input (address)		

Table 1: Stack while performing subroutine 1.

Subroutine 2

Explanation

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:

```
**Preparing stack frame for subroutine 1
LINK A0,#-20

MOVE.L A1, (-4,A0) **Address of input 1 on stack
MOVE.L A3, (-8,A0) **Address of number of exponent bits on stack
LEA (-12,A0),A4
MOVE.L A4, (-12,A0) **Storing sign
LEA (-16,A0),A4
MOVE.L A4, (-16,A0) **Storing exponent
LEA (-20,A0),A4
MOVE.L A4, (-20,A0) **Storing mantissa

BSR Subroutine_1
```

Storing the address of
itself on the stack.

The diagram consists of three blue arrows originating from a single point on the right side of the text box. One arrow points to the instruction 'LEA (-12,A0),A4', another points to 'LEA (-16,A0),A4', and the third points to 'LEA (-20,A0),A4'. These instructions are the ones that store the return address (the address of the instruction following the BSR) into 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.

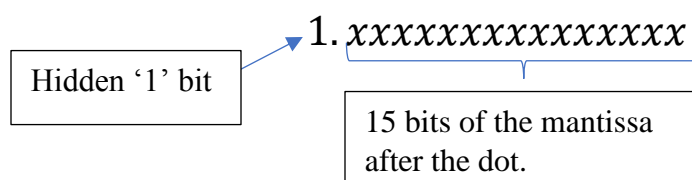
After that, we can add the exponents since it's a multiplication, we'll add D1 to D4.

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:



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:

$1x.\underbrace{\text{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}}$

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: `ASL.L #1,D5`
`ADD.L #1,D4`

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

If not, then this code would run: `ASL.L #2,D5`

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

PS: We can never have a product value of : 0.xxxxxx ... since we are multiplying two normalized mantissas. (1.xxxx ... × 1.xxxxx ...)

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:

1. First, we will shift our mantissa in D5 by 1 (And set the MSB to 0 because of the arithmetic shift right) then by the number of exponents bits to the right.
2. Then we get the bias (As in subroutine 1) and add to our exponent in D4
3. After that we shift our exponent to the left by the number of mantissa bits (So that it sits directly to the right of the mantissa).
4. Finally, we test the sign bit in D3, and set it (if D3 is set to '1'), in our MSB at D5.
5. After performing an OR.L on our 3 modified and shifted parts (Sign, exponent mantissa) we'll get our final product answer that we will store in the appropriate memory location.

The output has now been supplied; we unlink the stack frames that were created, then we return the registers as they were before the subroutine by reusing the MOVEM.L function.

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 (\$10,SP),SP

Stack

This is how the stack looks like while performing subroutine 1:

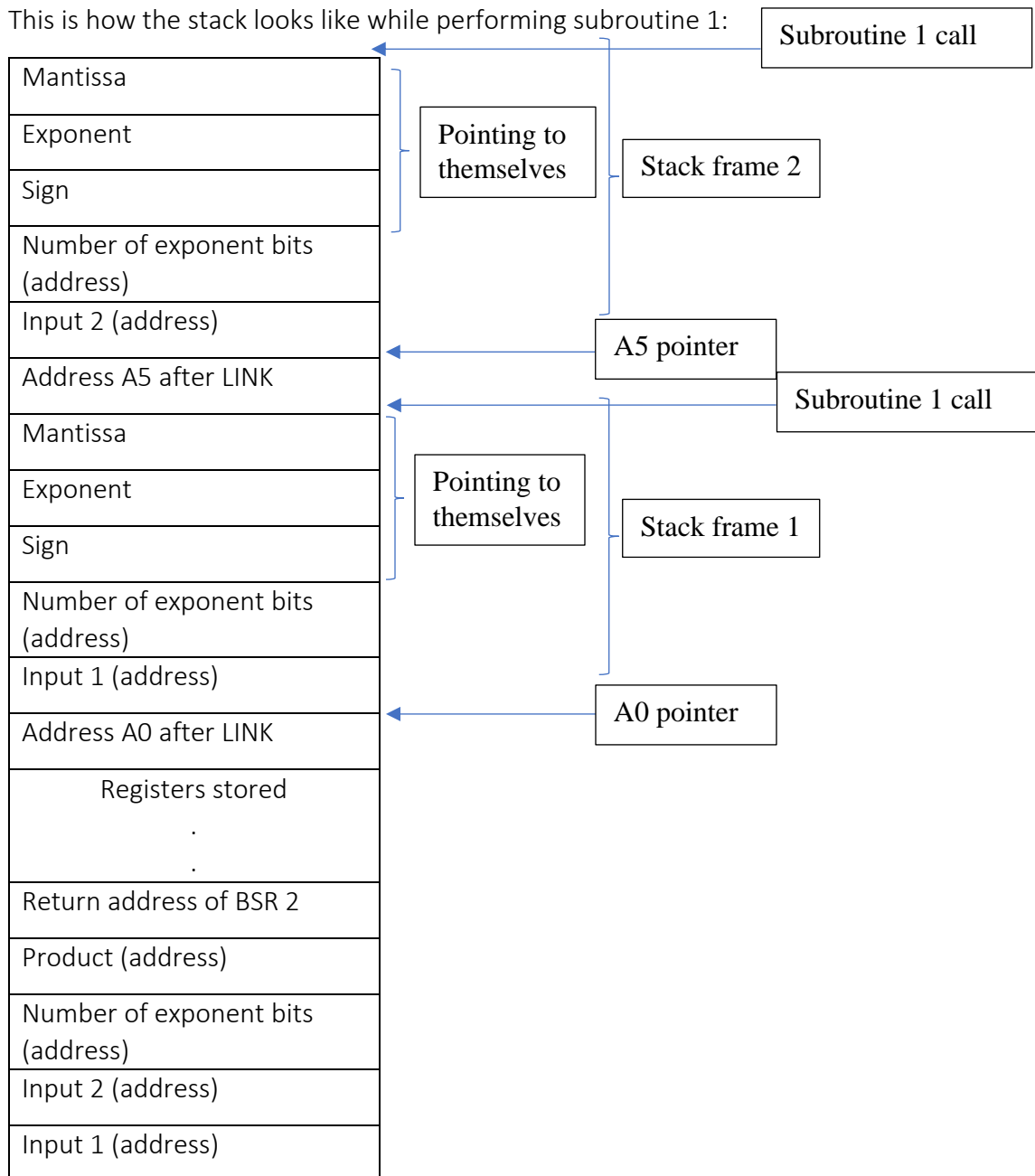


Table 2: Stack while performing subroutine 2.

PS: Subroutine 1 stack changes were not included in this table since the stack returns to its original form after performing a subroutine 1 call.

Appendix A

Project Code:

```

        ORG1000$
Input_1    DC.L ??
Input_2    DC.L ??
Exponent_Bits DC.W ??
Product    DS.L1
Sign_1     DS.W1
Exponent_1 DS.L1
Mantissa_1 DS.L1
Sign_2     DS.W1
Exponent_2 DS.L1
Mantissa_2 DS.L1

        ORG2000$
Subroutine_1 LINK A0,#-4** opening a stack frame maybe for bias

        MOVEM.L D1-D4/A1,-(SP)** should store the used ones

        MOVEQ #0,D2** Clear D2
**      all LEA +4 since based on A0 (base of stackframe)

        LEA (26,A0),A1
        MOVE.W (A1),A1** Original number address
        MOVE.L (A1),D1** Original number

        BPL.S POSITIVE

NEGATIVE    MOVEQ #1,D2** Storing sign in register

**          Maybe instead of LEA directly move.L to register
POSITIVE    LEA (22,A0),A1
        MOVE.W (A1),A1** Number of exponent bits address
        MOVE.W (A1),D3 ** number of exponent bits

        LEA (18,A0),A1
        MOVE.W (A1),A1** Address for sign
        MOVE.W D2,(A1)** Storing sign in memory

**          Until this point: The sign is stored in memory
**          D1 has the original number
**          D3 has the number of exponen bits
**          D2 was used to get the sign

        MOVEQ #31,D4 ** 32bit number - 1 for sign

```

SUB.W D3,D4** Storing the number of mantissa bits in D4(31 - number of exponent bits)

** D4 was used to get the number of mantissa bits

SUBQ #1,D3** D3 has become number of exponent bits - 1

MOVEQ #1,D2

ASL.L D3,D2** $2^{(number\ of\ exponent\ bits - 1) - 1}$

SUB.L #1,D2** D2 is the bias

** D2 has now the the bias

MOVE.L D1,(-4,A0)** Duplicating original number and storing it in the stack frame

ADDQ #2,D3** number of exponents bits + 1 (Sign)

ASL.L D3,D1** D6 is our final mantissa (Shifted the mantissa to the left by the number of exponents bits +

1

LEA (10,A0),A1

MOVE.W (A1),A1

MOVE.L D1,(A1)** Mantissa stored

MOVE.L (-4,A0),D1** Fetching the original number duplicate from the stack frame

AND.L #\$7FFFFFFF,D1** Set MSB to 0 (For ASR)

ASR.L D4,D1** Moving our exponent by the number of mantissa bits to the right

SUB.L D2,D1** Removing the bias from it

LEA (14,A0),A1

MOVE.W (A1),A1

MOVE.L D1,(A1)** Exponent stored

MOVEM.L (SP)+,D1-D4/A1

UNLK A0

RTS

ORG3000\$

Subroutine_2 MOVEM.L D1-D7/A1-A6,-(SP)

MOVEA.L (68,SP),A1** Address of Input 1

MOVEA.L (64,SP),A2** Address of Input 2

MOVEA.L (60,SP),A3** Address of number of exponent bits

MOVEA.L (56,SP),A6** Address of product result

** Preparing stack frame for subroutine 1

LINK A0,#-20

MOVE.L A1,(-4,A0)** Address of input 1 on stack

MOVE.L A3,(-8,A0)** Address of number of exponent bits on stack

LEA (-12,A0),A4

MOVE.L A4,(-12,A0)** Storing sign


```

LEA (-16,A0),A4
MOVE.L A4,(-16,A0)** Sotring exponent
LEA (-20,A0),A4
MOVE.L A4,(-20,A0)** Storing mantissa

BSR Subroutine_1
**      Stack is now pointing at the top of the stack frame(Where the mantissa is)

**      Testing if subroutine performed correctly
MOVE.W (-12,A0),D0** sign1
MOVE.L (-16,A0),D1** exponent1
MOVE.L (-20,A0),D2** mantissa1

**      Preparing stack frame for subroutine 1
LINK A5,#-20

MOVE.L A2,(-4,A5)** Address of input 2 on stack
MOVE.L A3,(-8,A5)** Address of number of exponent bits on stack
LEA (-12,A5),A4
MOVE.L A4,(-12,A5)** Storing sign
LEA (-16,A5),A4
MOVE.L A4,(-16,A5)** Sotring exponent
LEA (-20,A5),A4
MOVE.L A4,(-20,A5)** Storing mantissa

BSR Subroutine_1

**      Testing if subroutine performed correctly
MOVE.W (-12,A5),D3** sign2
MOVE.L (-16,A5),D4** exponent2
MOVE.L (-20,A5),D5** mantissa2

**      Logic of multiplication:

EOR.B D0,D3      **getting sign
ADD.L D1,D4      **getting exponent

**      Shifted by 1 to add the hidden bit, then shifted by 16 (To take a word space)
ASR.L #1,D2
ASR.L #1,D5
OR.L #$80000000,D2
OR.L #$80000000,D5

MOVE.B #16,D7
ASR.L D7,D2
ASR.L D7,D5

MULU D2,D5
BPL POS
NEG      ASL.L #1,D5
ADD.L #1,D4
BRA ENDOSUBROUTINE2

```

POS ASL.L #2,D5

**Only thing left to do is reconstruct the product
 **new sign in D3, exponent in D4, mantissa in D5

```

ENDOFSUBROUTINE2 ASR.L #1,D5
AND.L #$7FFFFFFF,D5
MOVE.W (A3),D7** D7 has become the number of exponent bits
ASR.L D7,D5** Shifted by number of exponent bits + 1 in total

SUBQ #1,D7** D7 has become number of exponent bits - 1
MOVEQ #1,D2
ASL.L D7,D2^2** (number of exponent bits -1)-1
SUB.L #1,D2** D2 is the bias

ADD.L D2,D4** Adding the bias to the exponent

MOVEQ #30,D1
SUB.L D7, D1 **30 - (number of exponent bits - 1) = number of mantissa bits
** D1 has now the number of mantissa bits

ASL.L D1,D4
OR.L D4,D5

** Checking the sign
TST.L D3
BEQ.S STOREINMEMORY
OR.L #$80000000,D5

** Storing product in memory:
STOREINMEMORY LEA (A6),A6
MOVE.L D5,(A6)
UNLK A5
UNLK A0
MOVEM.L (SP)+,D1-D7/A1-A6

RTS

ORG400$

START MOVEA.L #$4000,A7 **Only to visualize the stack
PEA Input_1
PEA Exponent_Bits
PEA Sign_1
PEA Exponent_1
PEA Mantissa_1

```

```
BSR Subroutine_1
LEA ($14,SP),SP
**      Stack pointer back to normal

**      PART 2:

MOVEA.L #$5000,A7 **Only to visualize the stack

PEA Input_1
PEA Input_2
PEA Exponent_Bits
PEA Product
BSR Subroutine_2
**      Return stack as it was
LEA ($10,SP),SP
**      Stack pointer back to normal

END START
```