EE357 Lab 3 –FP Implementation

Prof. Annavaram

DUE: 5 PM., Friday, 11/02/2012

1 Introduction

You will work as a team to design a software implementation of IEEE Shorthened Format compliant FP adder/subtractor. In this project you will be responsible for creating an assembly emulator of floating point addition/subtraction.

2 What you will learn

This lab will help give you experience decomposing a problem into subroutines to make the design process more manageable. Further, you will better understand the IEEE floating point standard and the details surrounding its implementation.

3 Background Information and Notes

IEEE Shortened Floating Point Format:

12-bit format defined just for this class is shown below.

1	5	6
Sign	Exponent (Excess-15)	Fraction

Figure 1 – IEEE Shortened Format

General purpose processors as well as high-end embedded processors usually contain a hardware implementation of a FP arithmetic unit. However, many processors (especially low-end embedded cores) do not include floating point HW, but instead emulate the operations in SW (using only integer operations/instructions) if needed. You will implement this kind of emulation for FP addition and subtraction.

To reduce complexity, we will make the following design assumptions:

- The only special value of FP numbers (i.e. 0, inf., NaN, and Denormalized) that can be presented as an input is the '+0' value.
- The inputs will never cause an output to be one of the special values except 0.
- We will only implement the round-to-nearest method.

FP Addition/Subtraction Algorithm (for normal inputs):

1. Determine smaller number and shift its fraction right to make exponents equal
2. Sign of result = sign of larger
3. If p+p or n+n, magnitude of result = sum of magnitudes
4. If p+n or n+p, magnitude of result = difference of larger mag. – smaller mag.

5. Normalize resulting magnitude.
6. Round-to-nearest, and re-normalize if necessary
7. Check for the special case of 0.

Identifying Intermediate Signals/Variables

While the data registers (D0 ~ D7) are the fastest storage units, due to the limited number of the units, you may need to create several intermediate variables to store interim results as you compute the final FP result. For example, you will likely find it helpful to start your code by separating the sign, exponent, and fraction fields for each input into separate variables: signa, signb, expa, expb, fraca, and fracb. Similarly, when determining the larger operand, it may help to store the comparison result in a variable (e.g. a_ge_b=> A greater than or equal to B) so that you can reference that result later when determining what code your program should execute.

4 Procedure

1. Create a new project (call it "ee357_lab3_FP_emulation") and then import/setup the necessary assembly files into your project.
2. We assume that two floating point operands are in D0 and D1 using the 12 bits left to the LSB.
3. FP addition and subtraction results are calculated.
4. The two results are displayed repeatedly (infinitely) on the LED's from the MSB to the LSB as follows: two seconds for each 4-bit binary number; two seconds between 4-bit binary numbers; and four seconds between the addition and subtraction results – e. g. if the addition and subtraction gave us 1000 0101 1000 and 0011 0011 1111 respectively, first 1000 is displayed (1(MSB) -> LED4, 0 -> LED3, 0 -> LED2, 0 -> LED1) for 2 sec.'s; all the LED's are off for2 sec.'s; 0101 is displayed for 2 sec.'s; …; all the LED's are off for 4 sec.'s after the 0011 of the addition result is displayed for 2 sec.'s; and the pattern repeats forever until interrupted.

5 Requirements

1. Instead of packing all the instructions in the main function, you must use subroutine calls as follows:
   a. In 'main', we call the following subroutines (must use the same names) after the two FP operand assignments to D0 and D1.

```
main:
    move.l #_______,D0 // The first FP operand filled in by your TA.
    move.l #_______,D1 // The second FP operand filled in by your TA.
    <BSR or JSR> LED_initialization // Initialize the LED's for output.
    <BSR or JSR> FP_addition
    <BSR or JSR> FP_subtraction
```

   <BSR or JSR> Display_result

b. The final results for the addition and subtraction must be stored in the last (LSB) 12 bits of D2 and D3 respectively. The other bits in the registers (bit 31 ~ 12) must be cleared as 0's.
c. We further divide the FP_addition and FP_subtraction into the following subroutines (must use the same names):

   Conv_sub // Convert subtraction to addition.
   Cal_expo // Calculate the exponent.
   Cal_frac // Calculate the fraction.
   Cal_sign // Calculate(determine) the sign.
   Do_normal // Do the (re)normalization if necessary.
   Do_round // Do the round to the nearest.
   Final_result // Finalize and format the result into D2 or D3.

   As a result, in the FP_addition and FP_subtraction subroutines, only the above subroutines should appear.

2. Add a comment to every assembly line.

6 Submission Instructions

1. You will submit main.s including all the comments **per team on Blackboard**. The file format must be exactly the same as the original one so it can be re-run by your TA. Different results from Demo may lead to 0 credits.
2. Demonstration will be Friday, 11/02/2012 after the discussion.
3. To avoid the late penalty, both the demo and the submission must be completed before 5 PM. on the due date.

Name(s): ______________________________________

Item	Outcome	Score	Max.
Demo result correctness:			
• Sign for addition in D2	Yes / No		1
• Exponent for addition in D2	Yes / No		1
• Fraction for addition in D2	Yes / No		1
• Sign for subtraction in D3	Yes / No		1
• Exponent for subtraction in D3	Yes / No		1
• Fraction for subtraction in D3	Yes / No		1
• Output display	Yes / No		1
Implementation requirements (Code Examination):			
• LED_initialization	Yes / No		1
• FP_addition	Yes / No		1
• FP_subtraction	Yes / No		1
• Display_result	Yes / No		1
• Conv_sub	Yes / No		1
• Cal_expo	Yes / No		1
• Cal_frac	Yes / No		1
• Cal_sign	Yes / No		1
• Do_normal	Yes / No		1
• Do_round	Yes / No		1
• Final_result	Yes / No		1
Documentation – comments in the source code: • (2 = Complete; 1 = Partially complete; 0 = Missing)			2
SubTotal			20
Late Deductions (30% per day)			
Total			
Open Ended Comments:			