Digit Design: A Systems Approach
Prelab 2: *Floating Point Adder*

Introduction/Overview

This week's laboratory assignment is to design a combinational circuit called a μ-*Law Floating Point Adder*. By the end of this prelab, you will become more familiar with building and debugging a digital system with hierarchical building blocks, in addition to gaining more experience with the tool chain. If you haven't yet, you should read Chapter 11 in the book to get a sufficient background in floating-point representation and the issues concerning its use and implementation.

Note, that in this lab subscript will refer to the base of the number. Base 16 means that we're writing the number in hexadecimal notation. A subscript of 2 would mean that we're writing out the number in binary, and a subscript of 10 would mean that we're writing out the number in straight decimal notation (as you normally would).

Background

Modern telecommunication systems use an 8 bit floating-point number representation called *μ-law*. In this representation a number is represented as $f = M \cdot 2^E$ where M is a 5 bit mantissa, $M = (m_4, m_3, m_2, m_1, m_0)$, and E is a 3 bit exponent, $E = (e_2, e_1, e_0)$.

A normalized floating point representation implies that the exponent is as small as possible (either the MSB of the mantissa is a 1 or the exponent is the lowest possible exponent). For example, there are two ways to represent the number 40_{16} using the floating-point format presented in this lab:

- ($E=4_{16}$, $M=04_{16}$)
- ($E=2_{16}$, $M=10_{16}$)

The latter representation is normalized due to the presence of a 1 as the MSB of the mantissa, which means that we cannot increase the exponent without changing the value of the number. By normalizing the floating point representation we increase its precision (again, see chapter 11 for a discussion of this).s

A floating point adder (FPA) takes two numbers in this format and calculates their sum, as shown in the table below:

input A	input B	Result
$E = 000_2$ $M = 01000_2$	$E = 000_2$ $M = 00011_2$	$E = 000_2$ $M = 01011_2$
$E = 001_2$ $M = 10001_2$	$E = 000_2$ $M = 01100_2$	$E = 001_2$ $M = 10111_2$
$E = 100_2$ $M = 10010_2$	$E = 010_2$ $M = 11111_2$	$E = 100_2$ $M = 11001_2$
$E = 111_2$ $M = 11110_2$	$E = 111_2$ $M = 11000_2$	$E = 111_2$ $M = 11111_2$

© 2012 All Rights Reserved

Note that in the third example the result isn't accurate. This is because some of the least significant bits of input b were truncated. Since we are constrained to a fixed number of bits for the mantissa, we lose precision as the exponent increases. Also, in the last example the FPA is saturated and therefore outputs the maximum possible number, as opposed to wrapping around.

Prelab Specification

Design a simple *μ-Law Floating Point Adder*, satisfying the specifications in this document. Luckily for you, a list of building blocks is given (below), which should provide hints on how to construct the FPA. All you have to do is implement the various building blocks and connect them together to form the FPA. The decomposition below is only a **recommendation**, however. As long as the interface to your *float_add* module remains as specified, you are welcome to decompose the adder into a greater number of modules than those listed below. You almost certainly do not want to use fewer modules, however.

Regardless of your choice of decomposition, the systematic way to approach this problem is to begin first by constructing a block-level diagram of the system you wish to build. Then, proceed by building and testing each of the modules in that design, starting at the bottom and then working your way up in the hierarchy. This requires you to write a testbench to verify the functionality **for each module**. You should remember to test for special boundary cases like the number zero, saturation, etc. These tests will allow you to pinpoint the errors at their source, instead of trying to find them after they've propagated through a large and complicated system. Also, be very careful when selecting the inputs in your testbenches, as some choices of inputs tend to make your module look like it is working when it is, shockingly, not.

Throughout this lab, you should assume the following:

1. Both the mantissa and the exponent are unsigned
2. Both inputs are normalized, as described above and in the course reader
3. If after adding the mantissas there is overflow, the exponent must be adjusted accordingly, and the 5 most significant bits of the mantissa must be retained.
4. You are building a saturating adder, meaning that if the result of the addition exceeds the maximum allowed number you should output the maximum (i.e. $E = 111_2$ $M = 11111_2$).

More on the Implementation

We will provide files for the input and display interfaces later, during lab. However, for the prelab, you will need to write and test each of the modules and building blocks listed below.

The *float_add* module is the highest level module in the hierarchy and takes two inputs of 8 bits each, labeled *aIn* and *bIn*, and outputs an 8 bit *result*. The operation is *result = aIn + bIn*. As a true combinational circuit it calculates the result continuously, does not rely on a start signal, and has no state/memory associated with it. As the top module it may have some combinational logic within it, but keep this to the minimum necessary. Most of the logic should be within its submodules. This is the only module that must be written as specified, in order to properly interact with the display and input blocks we provide you during lab time. You may deviate from the specification for the remaining modules if you can come up with a better decomposition.

Module Name	Inputs	Outputs	Function
float_add	aIn (8bits) bIn (8 bits)	result (8 bits)	This is the highest module in the hierarchy and realizes the prelab specification using the modules listed below.
big_number_first	aIn (8 bits) bIn (8 bits)	aOut (8 bits) bOut (8 bits)	Output number with bigger exponent as *aOut* and number with smaller exponent as *bOut*
shifter	in (5 bits) distance (3 bits) direction	out (5 bits)	The bits of *in* shifted by *distance* to the left (if *direction* = 0) or to the right (if *direction* = 1)
adder	a (5 bits) b (5 bits)	sum (5 bits) cout (1 bit)	Essentially a full adder without a carry in (same as having cin = 0)

big_number_first is used to guarantee that the exponent of the *aIn* signal to the FPA is greater than or equal to the exponent of the *bIn* signal. Initially, the provided numbers, *a* and *b*, are not ordered in any way. However, organizing them based on the magnitude of the exponent in the *float_add* module can simplify the logic inside the FPA. You'll see why as you move through the lab. Additional thing to keep in mind: why do we not bother to order by the mantissa as well?

The *shifter* module is fairly straightforward – it takes a 5 bit input and shifts the bits of that input either to the right or left a number of times specified by distance, padding with zeros where necessary. This is because before performing the actual addition of the two mantissas they must be aligned according to their exponents. If the difference between the exponents is *n*, then the mantissa of the smaller exponent must be shifted right by *n*, thus the bits being added will have the same weight. Think about why you would shift the input with the smaller exponent to the right as opposed as shifting the input with the larger exponent to the left (hint: it has to do with precision).

The *adder* module performs the addition on the mantissas. Note that the exponent of the result is not the sum of the two input exponents. Adding one exponent to another results in multiplication.

Provided Files

We will provide **no files** for the prelab portion of this lab; you are to create the project from scratch (we will provide you the wrapper for your files once we're in lab). Make sure that you follow the specification we've given you for the *float_add* module exactly or else you'll have to change your code during lab to fall in line with what the other blocks expect from your design.

Also, learn how to organize your code in a sensible way and remember the good coding conventions we've emphasized in section and class.

Prelab Submission

Compress and upload the following items into a zip archive with the naming convention:

lastname-firstname-prelab2.zip

1.**Block Level drawing of the FPA**. This drawing should demonstrate how you will implement the FPGA at a module level, and should always be the first step in working on a design. Label significant signals where necessary (labeling the output of a *mux* "*mux_out*" is not necessary). You should *NOT* make a gate level schematic. It is sufficient to use block notation as long as you clearly specify the inputs, outputs and the function of each block. Use .pdf format.

2.**All your Verilog modules**, including:

a. float_add.v
b. big_number_first.v
c. shifter.v
d. adder.v

3.**All of your testbenches** (should have one for EACH of the files listed above).

4.**Simulation outputs**. Basically, convince us that a) you know how to write appropriately thorough testbenches, b) you wrote such testbenches, and c) the modules, as you've implemented them, function as they should.

5.**A questions.txt file** following the example format with the following questions and their answers:

1. Is there any information the TA should know while evaluating your design?
2. Does your design completely meet the assignemnt's requirements? Explain.
3. If it does, what tests did you run to prove to yourself that the design worked?
4. Do you have any feedback on the lab?

