EE108 Lab 2

Floating Point Adder

Lab Due: October 7th at midnight

Objective: The purpose of lab 2 is to introduce you to floating point arithmetic and give you a chance to structure a complex Verilog module from scratch.

1. Introduction

So, what are we doing here?

This week we'll design a combinational circuit called a μ-*Law Floating Point Adder*. If you haven't yet, you should **read chapter 11 in the course text** to get a sufficient background in floating-point representation and the issues concerning its use and implementation.

A word on notation

In this lab, subscripts will refer to the base of the number we've written. A subscript of 16 means we're writing a hexadecimal (base 16) number. A subscript of 2 means we're writing out a binary number, and a subscript of 10 means we're writing out a plain decimal number as you normally would. For example, $A_{16} = 10_{10} = 1010_2$

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 1 or the exponent is the lowest possible exponent). It's like when you're writing decimal numbers in scientific notation and you make sure there is one digit to the left of the decimal point:

$9.05 * 10^5$

$.905 * 10^6$

Both represent the same number, but the one on the top has a single non-zero digit to the left of the point. In binary, we only have one possible non-zero digit, 1, so we say a binary floating-point number is normalized if the MSB of the mantissa is 1. For example, there are two ways to represent the number 40_{16} using the floating-point format presented in this lab:

($E=100_2$, $M=00100_2$)

($E=010_2$, $M=10000_2$)

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 (see chapter 11 for a full explanation).

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

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.

2. Lab

Specification

Design a *μ-Law Floating Point Adder* which has the following name and interface:

```
module float_add(
     input  [7:0] aIn,
     input  [7:0] bIn,
     output [7:0] result
);

//Your submodules and connecting logic go here

endmodule
```

All inputs and outputs are positive 8 bit μ-law floating point numbers with the mantissa in the 5 least significant bits and the exponent in the 3 most significant bits. 'result' is the sum of 'aIn' and 'bIn'.

First draw a block-level diagram of the system you wish to build. You will turn this diagram in as part of your lab report. Then, proceed by building and testing each of the modules in that hierarchy, starting at the bottom and working your way up to the top-level adder. Write a testbench **for each module**, making sure to test for boundary cases like the number zero, saturation, etc. Testing these modules individually will allow you to pinpoint errors at their source instead of trying to find them after they've propagated through a large system.

Assumptions

1. Both the mantissa and the exponent are unsigned, or in other words, all numbers are positive
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 sum must be retained. In other words, the sum's carry bit is the new mantissa's most significant bit, and the old mantissa's least significant bit is dropped.
4. If the result of the whole addition exceeds the maximum representable number, you should output the maximum representable number (i.e. $E = 111_2$ $M = 11111_2$). This is called **saturating**.

Suggested implementation

The table below outlines a suggested way to organize the logic of your adder into submodules. In order to properly interact with the display and input blocks we provide you during lab time, you

must declare float_add exactly as specified. We strongly suggest following this structure because it will be easier for the TA's to help you debug, if necessary.

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 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 (1 bit)	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. Additionally, think about why we do 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 adding 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* bits. It's just like how we add decimal numbers by hand: before adding we move the numbers horizontally such that their decimal points are lined up with each other, and assume there are 0s in the blank spaces.

The *adder* module performs the actual 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.

FPGA board interface

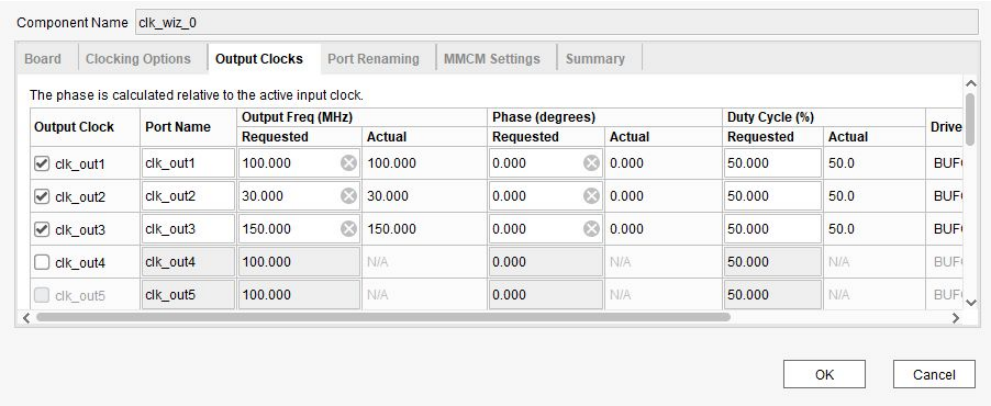
Like in lab 1 you will be provided modules that implement a video, button and switch interface for setting `aIn` and `bIn` and observing the result. In order to interface properly with the top module make sure your module name, inputs and output are consistent with the following:

```
module float_add(
      input  [7:0] aIn,
      input  [7:0] bIn,
      output [7:0] result
);
```

Setup

As a reminder, here's the things you'll need to do to setup your Vivado project:

1. Create a new project and select the PYNQ Z2 board.
2. Add simulation sources (_tb.v files), design sources (other files), and constraint source (lab2.xdc).
3. Setup github repository for source files.
4. Add IP blocks (HDMI TX).
5. Customize IP blocks (HDMI TX, and clock wizard).
    a. In the clock wizard, you will need to set clk_out1 to 100MHz, clk_out2 to 30MHz, and clk_out3 to 150MHz.



If you need reminders about the detailed process for each step, consult labs 0 and 1 for more info.

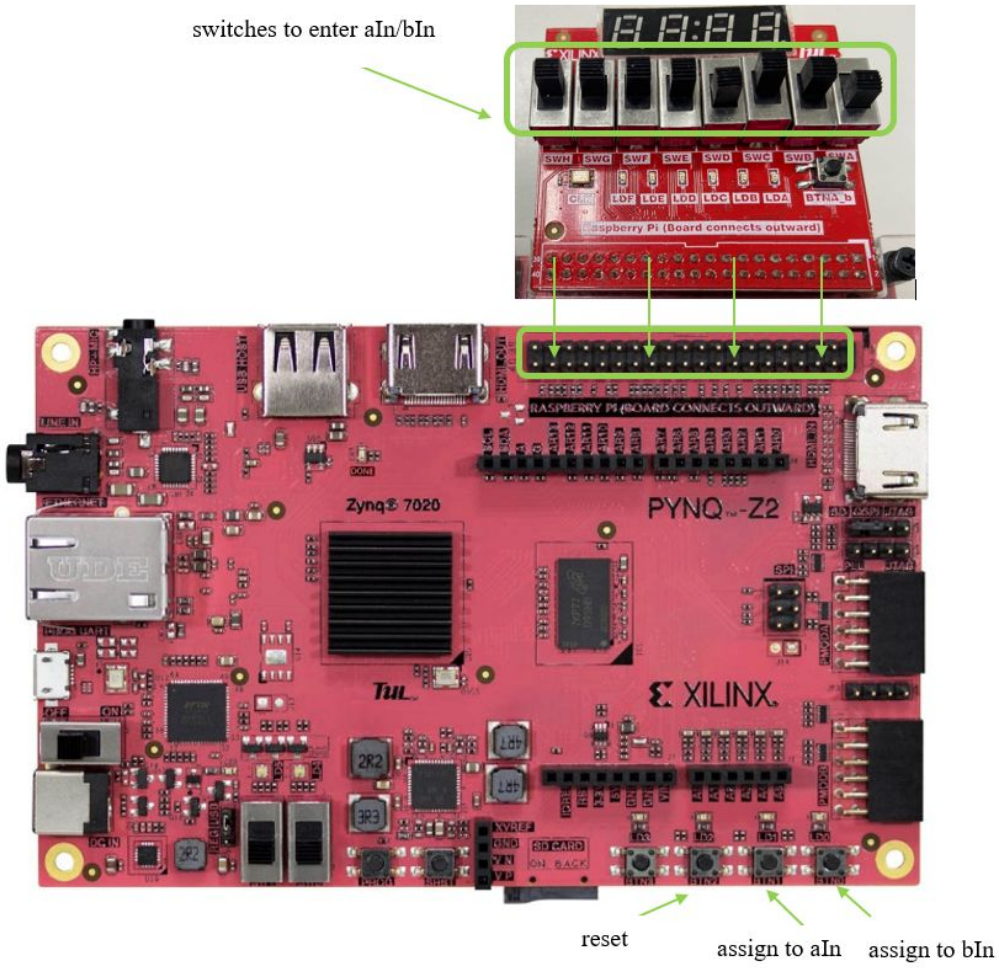
Synthesizing

Once you're in the lab with a verified design:

1. Make sure that *`lab2_top`* has been set as the top level module (as indicated with the 3 squares next to it). Double-check to make sure there are no missing modules in the hierarchy (indicated by ?'s), then generate the bit file by clicking *Generate Bitstream*
2. Zone out for a few minutes while Vivado chews up your design very thoroughly.
3. Load your design onto the board using the hardware manager.

Checking the result

Once you have synthesized your design and put it on the board, use the eight switches on the attachment card to input a floating point number, then hit BTN1 to assign the value to aIn, or BTN0 to assign the value to bIn. Check to make sure the resulting sum is correct. Denormalized numbers (which you were not required to handle) will be shown in red. Saturated results will be shown in yellow. You can reset the entire display using BTN2.



Lab deliverables

Zip or tar up all of the following files and submit them on gradescope:

1. **Block Level drawing of the FPA**. This drawing should demonstrate how you will implement the adder at a module level. Label significant signals where necessary (labeling the output of a *mux* "*mux_out*" is not necessary). You should *NOT* make a gate level schematic. Use block notation and clearly specify the inputs, outputs and the function of each block.
2. **All Verilog modules you write.** If you followed the suggested decomposition, these are:
    a. float_add.v
    b. big_number_first.v
    c. shifter.v
    d. adder.v

    Remember to include the correct versions, since sometimes Vivado caches old versions in your project directory!

3. **All testbenches** (should have **one for each module** you write).
    a. float_add_tb.v
    b. big_number_first_tb.v
    c. shifter_tb.v
    d. adder_tb.v

4. **Simulation outputs**. Convince us that you know how to write appropriately thorough testbenches, you wrote such testbenches, and your implementation of each module functions as it should.
5. **Your .bit file**. This file is generated when you synthesize.
    a. This is located at lab2/lab2.runs/impl_1/lab2_top.bit
6. **A README.txt file (or .pdf)** with:
    a. Your names
    b. Any information that the TA should know when evaluating your design.
    c. What was your testing strategy? How did you prove to yourself that your design completely satisfied the lab's requirements? Did any of your tests catch bugs you weren't previously aware of?
    d. An explanation of why we shift the input with the smaller exponent to the right as opposed as shifting the input with the larger exponent to the left.

3. Demo of your lab

Load your design onto the FPGA using Vivado, generate, and submit all of the following in a **.zip** archive on Gradescope by midnight on October 7th:

1. The generated synthesis and timing reports
    a. lab2/lab2.runs/synth_1/**lab2_top.vds**
    b. lab2/lab2.runs/impl_1/**lab2_top_timing_summary_routed.rpt**
2. The same resource usage statistics we collected in lab 1. As a reminder, go to "Window -> Reports" and double click on "impl_1_place_report_utilization_0."
    a. What fraction of the FPGA's slice LUTs did your implementation consume?
    b. How many slices are occupied?
3. A screenshot of the FPGA layout (Implementation -> Open Implemented Design) like we made in lab 1. Try to figure out what one of the slices is doing in your adder – this might be easier if you start from the **netlist** on the left side of the screen, find a signal you recognize, right-click it, and select 'Mark'. You can then zoom in to its location on the physical FPGA diagram. Take a screenshot of this and include a text file explaining what you think it's being used for.
4. A README.txt file with your partners' names and any feedback you have on the lab.
5. A video of you testing these three cases:
    a. A = 10010010, B = 01011111
    b. A = 11111110, B = 11111000
    c. A = 00001100, B = 0011001

Now is a great time to meet up with your partners and divide up the work for lab 3. It is much longer and more error-prone than this one was, so start early!