ASIC FPGA Prototyping - Sensor Error Detector, 1-, and 8-bit Adders

Jared Botte

January 29, 2022

1 Overview

In this lab, we will be implementing the Sensor Error Detector, 1-, and 8-bit adders on the DE2-115 development board. This lab will assume that these assignments have already been completed, and will walk through the steps of implementing these programs on a development board for testing.

2 Setup

To begin, open a terminal window, create a new directory, and move into it. Also, setup the directory structure and copy over the base makefile.

```
mkdir adder-error-detector
cd adder-error-detector
mkdir -p source include
cp $TOOLSDIR/makefile .
```

This will be the standard setup for most labs.

3 Demonstrating a sensor error detector

Let's demonstrate the sensor error detector we created. First, choose any of your designs and copy into your new source directory. Rename it `sensor.sv`.

Next, we'll need to modify our file to use inputs on the board. The options that come to mind when looking at the board are the buttons (or keys as they're labeled) and the switches. We'll also need to choose an output. Luckily, there are tons of LEDs to choose from.

3.1 Change the inputs to switches 3:0

To change the sensors to be switch input, we'll need to change the input declaration at the top of our module.

Change it to look like this instead:

```
input logic [3:0]SW,
```

Hopefully the transition here is clear; we've set the input type to `logic` instead of `wire`, and told it to use switches 3, 2, 1, and 0 as input.

Now, change all references in your modules to use `SW[x]` in place of `sensors[x]`.

3.2 Change the output to LEDG8

Now that we've changed the input to use the switches, we'll need to find a way to display the output. Let's output the error on green led 8 (between the hexidecimal displays). Although it's only a one bit output, we must set it to go from 8:8 in order to isolate LEDG8. Replace the output line with:

```
output logic [8:8]LEDG
```

Lastly, replace `error` with `LEDG[8]`. This will now light up LEDG8 when an error has been detected.

3.3 Program the design to the board

To program your design to the board, return to the terminal window. In your main directory for the lab (i.e. not in the `source` directory, but the parent directory), run `make sensor`. This will compile the design, create the necessary files to map the physical pins on the board to your design, and upload the design to the development board.

For more make rules, run `make help`

3.4 Try it out!

Now that you can test your sensor error detector using the switches and leds, try messing around with the development board some more.

3.4.1 Challenge 1: Use the buttons (keys) instead of the switches

Refer to the DE2-115 documentation (Specifically chapter 4) to determine the names of the keys and their functionality.

Question: Why does the error detector appear to have "flipped" behavior when using the buttons?

3.4.2 Challenge 2: Use a red LED to show the error

Now try using LEDR[8] as your error output. Also try using LEDR[0].

Question: What happens if you initialize more than one red LED by saying `output logic [8:0]LEDR`? What happens to all the LEDs that are initialized? What about the uninitialized ones?

4 1-bit adder

Next we will move onto our 1-bit adder program.

Copy your 1-bit adder into the `source` folder and make a copy of it called `adder_1bit_mapped.sv`. Use this mapped version for this part of the lab (more on why later)

Open `adder_1bit_mapped.sv` in an editor, and replace your inputs and outputs with switches and leds respectively. When finished, run `make adder_1bit_mapped` to test out your design.

Question: Did you use 3 input statements and 2 output statements? If you're using sequential Switches/LEDs, do you need 3/2 statements respectively? How can you condense these into one statement each? Try it out!

5 8-bit adder using nbit adder template

Now that the 1-bit adder is working, let's add in our 8-bit full adder. remember that the 8-bit adder uses the n-bit template adder, so you'll need to copy that file over too.

5.1 Why did we need to make a copy of the 1-bit adder earlier?

Our template requires the 1-bit adder to have inputs with names `a`, `b`, and `carry_in`, and outputs with names `carry_out` and `sum`. On top of that, if we were to map our 1-bit adder inputs directly to switches on the board, all our adders would have the same inputs and not be connected properly! Therefore, we created a seperate file to map to our board, and leave the generic one for later use. In this case, we're using a wrapper file to create the 8-bit adder, so we'll map our inputs and outputs there.

5.2 Setup the 8-bit adder wrapper to use switches and LEDs

Try to modify the 8-bit adder file to take input from SW[7:0] (input a), SW[15:8] (input b), SW[16] (carry_in) and to put the output on LEDG[7:0] (sum) and LEDG[8] (overflow). Once you've done this, try running it and testing that it works as expected.

5.3 Why does Quartus Prime give us the error "this block requires a name"?

Chances are if you've tried to compile this that you've gotten an error on the for loop of the nbit adder module. The error states that "this block requires a name". This is something specific to Quartus Prime (as opposed to Quartus II used in ECE 337). To fix this, simply add a name for the generate loop after the `begin` statement like so:

```
genvar i;

generate
for(i = 0; i < BIT_WIDTH; i = i + 1)
    begin: adder_generator
        <adder logic here>
    end
endgenerate
```

5.4 Try it out!

5.4.1 Challenge 1: Display the state of the switches on the red LEDs above them

Try using the red LEDs above the switches to show their current state. Your implementation should be 2 lines of code. 1 port declaration and one `assign` statement.

5.4.2 Challenge 2: Nicer displays

Although counting in binary is a favorite pastime of us computer engineering students, we have 7-segment displays on our development board! Let's use them to make our life a bit easier so we can verify that our adder works as expected. Set the leftmost displays HEX7/HEX6 to map to SW15-SW12/SW11-SW8 respectively. Map the middle displays to the other switches, and map the rightmost three displays to the carry out and sum values.

Note: It may be useful to use a 2d array to store and access the hexidecimal character values for the seven segment displays. Also note that the seven segment displays are active low.

Hint: How can you control an output value using an input value using a hardware implementation?