CE/CZ1005

Lab 4: Combination Design with Verilog

Objective

To introduce the design approach for combinational circuits using Verilog.

Equipment

Same as Lab 3. You should also have to hand your Lab 3 manual and a copy of the lecture notes on Verilog.

Part 1

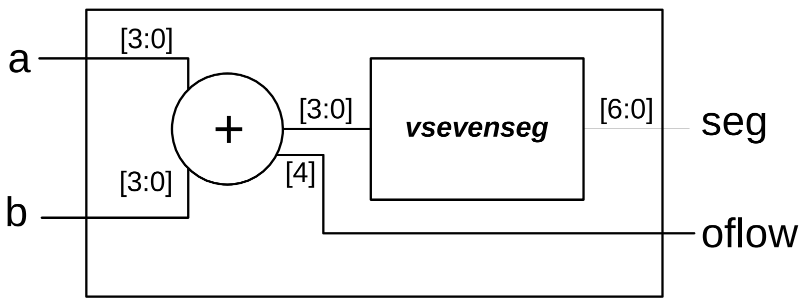
First, we will re-implement the seven segment display decoder using behavioural Verilog. This will show how much easier behavioural description can be, since we do not need to work out any logic equations. We will implement the decoder using a case statement in a combinational always block. Use your filled in Table 1 from Lab 3 to give you the required outputs.

1. Start a new project named *Lab4*. Ensure the Top Level Source Type is set to HDL
2. In the following options window, ensure the correct device details are set, and that the Preferred Language is set to Verilog. Use the same project settings as Lab 3.
3. Create a new module: Project | New Source, select Verilog Module, and name it *vsevenseg*.
4. Give the module a single input called *a*, select bus, and set the MSB to 3. Add an output called *seg*, select bus, and set the MSB to 6. This creates a module with a 4-bit input and a 7-bit output. The input is a 4-bit binary number, and the outputs are the seven LED segments.
5. You should now see a bare module declaration with the inputs and outputs as you indicated.
6. You should now fill out your module with a single **`always @ *`** block, that contains a case statement. The case statement should switch on *a*, and for each case, make a fixed assignment to the 7 bits of *seg*, e.g. **`seg = 7'b1101101`**. We assume the order of bits to be segment *gfedcba*, as in the last lab. Remember your *seg* output will need to be declared as type **reg**.
7. Your case statement should give the correct values for all 16 possible inputs to produce the outputs 0-F.
8. Double-click "Synthesize – XST", then "View Technology Schematic". Double-click the *vsevenseg* symbol and you should see 7 *lut4* elements, each one implementing the logic for one output. If you double-click a LUT, you can see its contents. Look-Up Tables are used to implement combinational logic in FPGAs, rather than gates. Check how the logic corresponds with what you determined in Lab 3. Note that the input order may be different to what you specified.
9. Add a new implementation constraints file to the project. You can use the .ucf file from Lab3, othewise, follow instruction 5.4.2 in the Lab 3 instructions. Note, you will need to remove the tglout constraint if it is there.
10. Implement your design and test it on the FPGA board, verifying correct functionality.

Here we have used behavioural modelling to implement this combinational circuit. You can see that even with seven outputs, we only needed to write one case statement. The tools took care of all the other steps including determining Boolean logic, simplification, and mapping to the logic available on the FPGA.

Part 2

We will now implement an adder from a high level description, letting the synthesis tools create the circuit. We will also see how to instantiate modules within our current design.



1. Create a new Verilog module called *vaddoflow* with two 4-bit inputs, *a* and *b*, a 7-bit output, *seg*, and a 1-bit output *oflow*.
2. Let us first create a simple adder using an **`assign`** statement. First declare a 5-bit internal wire, then use an assign statement to assign the sum of the two inputs to it.
3. Now, instantiate your *vsevenseg* module from Part 1, using explicit connection. Its output should be connected to the module output, as shown in the figure. Its input should be connected to the four least significant bits of the adder output.
4. The *oflow* output can be taken from the most significant bit of the adder result.
5. Set *vaddoflow* to be the top module.
6. Now modify your .ucf file so that the leftmost four switches on the board are connected to the a input, and the rightmost four switches are connected to the b input. The seg output should be the same as in Part 1. Add a constraint to connect the *oflow* output to an LED LD0 (pin U18).
7. Due to a software bug, you may need to right click your constraints file and select Remove from Project, then go to Project | Add Source… and add it back.
8. Test the adder and display circuit. You should see the *oflow* LED light whenever the result is move than F.

We were able to make use of the vector selection operations to use different bits of a vector for different uses. The adder was also synthesised entirely from a single Verilog arithmetic expression.

Part 3

In part 5.6 of the last lab, you you built a circuit that allows you to select which digit of the seven segment display to use. Here, we will create the circuit that enables us to use both at the same time. This is done by switching the segments at a high enough rate for them to both appear lit at the same time. You do not need to worry about the details inside the module yet, as this includes synchronous design, that you have not yet covered.

1. The two digits on the seven segment module cannot be controlled at the same time. Instead, we need to send the two numbers to display separately at high speed in a repeating manner. As long as this happens at around 50Hz, the digits won't blink.
2. Download the *segtoggle.v* file from edveNTUre.
3. Within the file, you will find a comment, where you should copy your seven segment decoder always block from Part 1. You only need to copy the always block. Modify the signal that controls the case statement to be *numsel*, as indicated.

4. Add the following pin constraints to your constraints file, and remove the *oflow* signal constraints:
   ```
   NET "tglout" LOC = T4;
   NET "clk" LOC = L15;
   NET "rst" LOC = T15;
   ```
   And remember to add the corresponding IO standard lines. Repeat the remove and add on the constraints file, as you did in the last part.
5. Generate the Programming File and test your circuit. It should now display two-digits, each controlled by four of the switches on the board.

Information on the provided module: The segtoggle module implements a 20-bit counter and uses its 20th bit as a slowed down version of the clock. The board clock runs at 100MHz, and this bit toggles every 1,048,576 clock cycles, so it slows the switching down to about 50Hz for a full cycle, which is fast enough to make a digit look permanently lit. The other function of this module is to alternate one of the two 4-bit digit inputs in time with the toggling strobe tglout. So the first digit is always output with tglout set to zero, and the second digitis output with tglout set to 1; a simple multiplexer. The tglout signal is also used by the board to enable the corresponding digit.

Optional Part 4

1. For this optional final part, we will implement a binary multiplier and display the result on the two digits. You may use the same source file as in Part 3, or save a copy.
2. Within the module, we would now like to multiply the *a* and *b* inputs together to give an 8-bit number x:
   ```
   wire [7:0] x = a * b;
   ```
3. This 8-bit number can then be displayed on the seven segment display by using the first four bits as the first digit and the second four as our second digit into the multiplexer. You will need to edit this line:
   ```
   assign selnum = div20 ? a : b;
   ```
   Such that *a* and *b* are replaced with the two parts of the multiplier result.
4. Implement and test your design on the board. Since the module port names have not changed, there should be no need to change the UCF file. Though you may need to remove and re-add.

Multiplying two 4-bit numbers together produces an 8-bit result, which we have called x, and assigned implicitly in its wire declaration. We are now able to take the four least significant bits and use them to drive the rightmost digit, while the four most significant bits drive the leftmost digit. Remember 0 multiplied by anything will give zero. You can also test that the multiplier gives the same result for two operands in either order.