Lab 6

MUX and 7-segment Decoder

Updated: February 18, 2020
Version: SystemVerilog 2017

6.1 Introduction

In this lab, we will be setting up a circuit to display an 8-bit number on two 7-segment displays (i.e. two hexadecimal digits). You will design a couple of combinational logic modules in Verilog and then implement your design on an FPGA.

6.2 Objectives

After completing this lab, you should be able to:

- Write a multiplexer in Verilog using the conditional operator
- Write combinational Verilog components using an `always` block
- Define and use multi-bit signals (vectors)
- Instantiate and connect components in a top-level module
- Use provided constraint files to specify package pins
- Implement design on FPGA (Digilent Basys3 board)

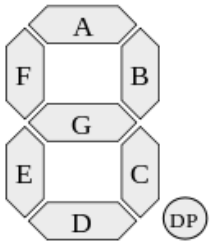

Figure 6.1: Segments of a 7-segment display

6.3 Pre-Lab

1. Read Part 2 of the SystemVerilog Tutorial, and be prepared to answer the following questions:
   (a) What is the conditional operator equivalent to?
   (b) What data type must be used when assigning the value inside a procedural block?
   (c) What are these and when are they used: `=` and `<=`?
   (d) What is different about *procedural* statements?
   (e) What does @* mean/indicate?
   (f) When do you need `begin` and `end`?
   (g) Both `if` and `case` statements infer multiplexers in hardware. What is the difference?
   (h) What are three common errors with `always` blocks?
2. Develop a truth table for a decoder that converts a 4-bit binary number into a hexadecimal digit on a 7-segment LED display. The segments of the display are labeled "a" through "g" (plus a decimal point, "dp"), as shown in Figure 6.1. It's important to note that the display on the Basys3 board is active low (or uses "negative logic") meaning 0 = On and 1 = Off. More information can be found in the Basys3 reference manual.

   Your truth table should include the binary value, its hexadecimal equivalent, and the output values for the 7-segment display, as shown in Table 6.1. The first line, representing zero, is completed for you. Only the center segment (g) is turned off. Note that g comes first in order

to match with the typical bus signal numbering scheme (0 corresponds to the LSB). Use upper case letters for all but 'b' and 'd.'

Bin	Hex	g	f	e	d	c	b	a
0000	0	1	0	0	0	0	0	0

Table 6.1: 7-segment display table

6.4 Procedure

As we will do for all labs, create a new project with the same properties as before (Type = RTL project, Board = Basys3).

6.4.1 Multiplexer

We will need a multiplexer or "MUX" to switch between the digits of the display. Create a new SV file and name it `mux2_4b`. The "2" is because this will be a 2-input MUX and "4b" stands for "4-bit".

You will have two inputs `in0` and `in1` that need to be 4 bits. Another input, `sel`, should be one bit and is the "select" line that chooses between `in0` (when `sel=0`) and `in1` (when `sel=1`). Finally, you need a 4-bit output, `out`. Use the conditional operator to implement your 2:1 MUX. Listing 6.1 shows part of what your file should look like.

Listing 6.1: Unfinished 2:1 4-bit MUX

```
module mux2_4b
    (
    //inputs and outputs
    );

   assign out = sel? //... ;

endmodule
```

Create a test bench for your module and verify that it behaves as expected. Use at least two different sets of input values and observe the output for `sel` = both 0 and 1. That is, you should have at least 4 test cases (input set 1, `sel`=0,1; input set 2, `sel`=0,1).

Note: You need an expected results table (ERT) for *every* test bench you write. It won't always be listed explicitly in the instructions.

Listing 6.2: Unfinished 7-segment decoder

```
module sseg_decoder(
   // input
   output reg [?:?] sseg;
   );

// 4-bit to 7-segment decode logic
// (note: output is active low)
always @*
   case(num)
      4'h0: sseg = 7'b1000000;
      // Fill in remaining cases here
   endcase

endmodule
```

The diagram for this module would simply be that of a 2-input MUX, shown in Figure 6.2.

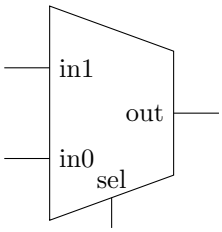

Figure 6.2: 2-input MUX

6.4.2 Seven-segment Decoder

Create a new SV file and name it `sseg_decoder`. It should have a 4-bit input `num` and a 7-bit output `sseg`. Use an `always` block with a `case` statement to implement the table you developed in the pre-lab. Listing 6.2 shows part of what your file should look like. Note that the output `sseg` is type `reg`.

To test your decoder, you could write out separate commands to implement your truth table from the pre-lab, but that is rather tedious. We can be more efficient by recognizing that there is a pattern to the inputs, namely counting by 1 in binary (or hex). This can be done with a `for` loop, as shown in Listing 6.3. (See For loop.)

Note the `<=` inside the condition (parentheses). In this case, it means "less-than-or-equal," but it is the same symbol as a non-blocking assignment.

Listing 6.3: Unfinished 7-segment decoder test bench

```
module sseg_decoder_test();

// I/O signals...

integer i;  // Declare loop variable

// Instantiate module...

initial begin
   for (i=0; i<=8'hF; i=i+1) begin
      num = i;
      #10;
   end
   $finish;
end

endmodule // sseg_decoder_test
```

It will also be easier to verify correct operation using hexadecimal instead of binary, so when you write your ERT, use hex for both input and output, as shown in Table 6.2.

Table 6.2: `sseg_decoder` ERT format

	Time (ns):	0	10	20	30
Input	num (hex)	0	1	2	...
Output	sseg (hex)	40	79	24	...

6.4.3 Top-level Module

Now create another SV file named `sseg1`. This file will connect your decoder module and MUX to the actual hardware on the FPGA board. You need to define the I/O ports listed in Table 6.3.

Name	Type	Width	Description
sw	input	16	switches
an	output	4	7-seg digits
seg	output	7	7-seg segments
dp	output	1	decimal point

Table 6.3: I/O signals for `sseg1`

The `an` signal is for the *anode* of the 7-segment display, and each bit turns on one of the 4 digits.

Listing 6.4: Partial `sseg1` test bench

```
initial begin
   // Initialize
   sw = 16'h0000; #10;
   // Test case 1
   sw[7:0] = //some#;
   sw[15] = 1'b0; #10;
   sw[15] = 1'b1; #10;
   // Test case 2
   // repeat with a different #
end
```

Instantiate your MUX and 7-segment decoder, then make connections as shown in Figure 6.3.

Create a test module for `sseg1`. Start by assigning all of the switch (`sw`) inputs a value of 0. Assign the lower 8 two different hex values (e.g. `A` and `B`). Set `sw[15]` to 0 and then 1. Repeat this process with a different value for the lower 8 switches. This is partially shown in Listing 6.4. Note that the output should match your hex output from the decoder (i.e. `sw = A` means `seg = 08`).

Make a list of all of the errors you find (and fix) when running this simulation.

6.4.4 Constraints Files

Download from Canvas the "constraints" files (`.xdc`) provided for the Basys3 board into a folder *outside* of your project. In Vivado, use the Add Sources dialog to add *only* the `switches.xdc` and `sseg.xdc` to your project. Check the box that says "Copy constraints files into project". This will automatically copy the files you selected into the "constrs_1" folder in your project folder.

These files tell Vivado how to connect your signals in Verilog to actual hardware pins on the FPGA. The I/O ports on the top module (`sseg1` in this case) **must** match those in the constraint files, which is how the names in Table 6.3 were chosen. Do **not** include extra constraint files, as Vivado has to match *all* constraints to signals in your Verilog code.

6.4.5 Implement and Test

Generate a programming file ("bitstream") and program the Basys3 with it. Test the operation of your circuit by entering an 4-bit number on the lowest slide

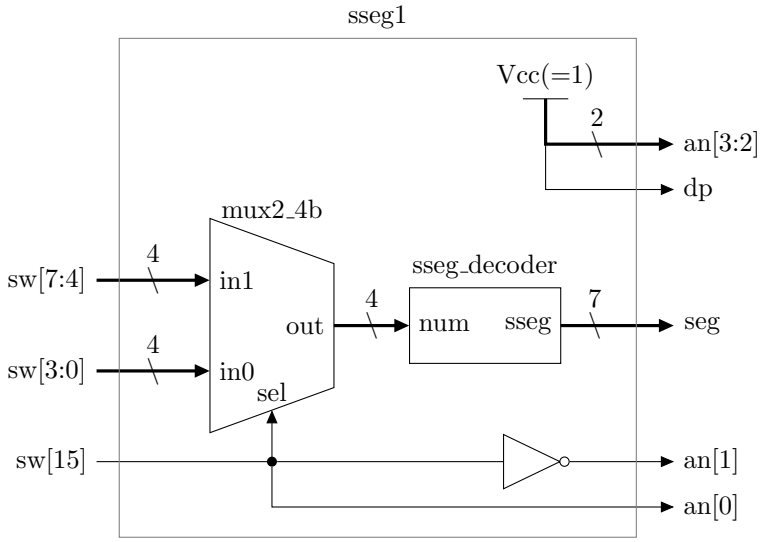

Figure 6.3: Block diagram for `sseg1`

switches and ensuring that the same number (in hex) appears on the first digit of the 7-segment display. Check all of your values to make sure your decoder is correct. Now change switch 15 and observe how it changes the digit. Use switches 7:4 to change the number shown on the second digit.

6.5 Deliverables

Submit a report containing the following:

1. Your Verilog source files (3 design, 3 test)
2. Two pictures of your board, one showing a value on the first digit, the other showing a different value on the second digit
3. List of errors found during simulation. What does this tell you about why we run simulations?
4. How many wires are connected to the 7-segment display? If the segments were *not* all connected together, how many wires would there have to be? Why do we prefer the current method vs. separating all of the segments?