STARS 2024

Lab Experiment 6: Design and Verification of Combinational Logic in SystemVerilog

---

Objectives

- To learn how to convert various combinational circuit designs into Verilog.
- To evaluate these modules on the physical FPGA development board.
- To learn new Verilog/SV syntax for designing and verifying combinational modules.

Work with a partner for this lab!

Step 1: Implement and verify a seven-segment decoder

ssdec is perhaps one of the most important modules you will use (for this week, at least) because it allows you to see values on the development board's seven segment displays, and so implementing it is not as important as implementing it correctly. You'll use this module throughout the lab to see the outputs from various modules.

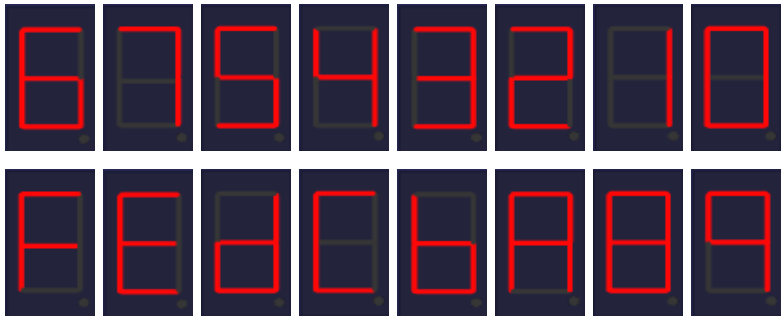
1.1: Implementation

Using the hints provided in lecture, create the **ssdec** module with the following ports:

Port	Type	Length
in	input	4
enable	input	1
out	output	7

Keep in mind that in order to ensure that **ssdec** is displaying correctly, you need to connect the output to a seven-segment display. Each display is defined as 8 bits long because it also includes the decimal point, which **ssdec** is not responsible for, which is why the output is 7 bits and not 8. Make sure to only connect the segments.

For the inputs, use push buttons. Make sure that your display is turned OFF when the push button is not pressed (or pressed, depending on whether you invert it when connecting it to the instance). Try all combinations of **in**, from 0 to 15, so that you see 0-9, A-F on the display. They should look like the following:



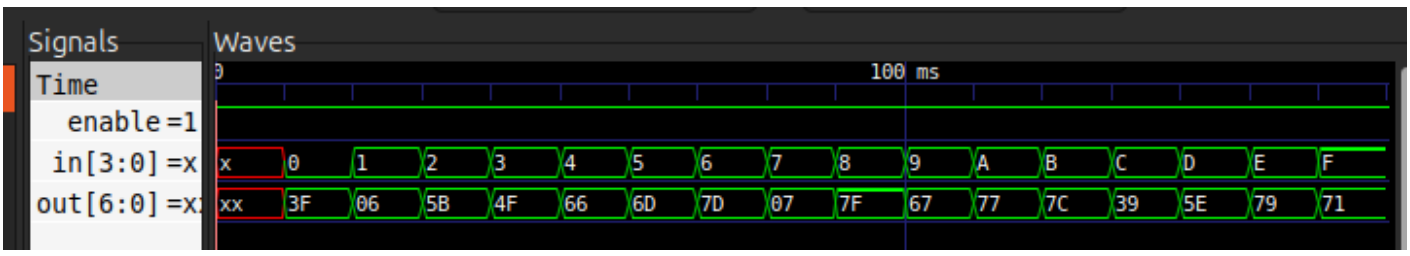
Take particular care with 9 - the D segment must not be lit! In some designs, that could be considered a small "g".

1.2: Verification

Once you are sure your module works, implement the testbench for it. The bulk of it has been provided to you in the slides - you just need to fill out the rest of the testbench function `ss_to_int`, which is just restating the seven-segment bit patterns for each digit.

This may seem redundant since you are writing a testbench for your own design, which is why **you should exchange your testbench with someone else's** and see if your **ssdec** module works with their testbench and vice versa! **You have to do both the module and the testbench to get full credit, not one or the other.**

Make sure that the testbench **tb_ssdec** goes in **tb.sv**, and your **ssdec** module is still in **top.sv**. Run **make sim** to open GTKwave, add all the signals from the **tb_ssdec** object in the left sidebar, click the "Zoom Fit" magnifying glasses to see the whole simulation timeframe, and you should hopefully see something like this:



out is in hexadecimal notation in GTKwave, so if you want to see the individual binary bits, double-click it to expand the bus into the constituent signals.

Note the part in the beginning where **in** and **out** are red. When you test your modules, you should always ensure that your inputs have been assigned values. The reason **enable** is 1 but **in** is a red X is because we didn't assign a value to the latter at the same time as **enable**. Go ahead and put that into the testbench - start it at 0.

Checkpoint L6C1: **Demonstrate to your peer mentor/instructor** that you can display the hex digits 0-F on your physical FPGA board, that you can turn off the enable and the display goes blank as a result, and that your testbench does not print out any error messages in the console when you run **make sim**. Show that your testbench works on your peer's **ssdec** and vice-versa. **Make sure to document everything in your lab journal. Include the name(s) of the peer(s) you worked with.**

Step 2: Implement and verify a single-digit BCD adder

As discussed in lecture, a BCD adder is one that takes two inputs of 4 bits each, a carry-in bit, and produces the BCD sum and carry-out of the three inputs.

To implement this module, you'll need the 4-bit full adder from the last lab (**fa4**).

The BCD sum is different from a regular sum in that the value 6 is added to the original sum if that original sum is higher than 9. Therefore, the carry-out becomes 1, and the new sum becomes the "ones" digit. E.g. if the two inputs were 9 and 5, whose sum is 14 (4'b1110), 6 is added to make it 20 (5'b10100), where the 5th bit is split off into the carry-out bit (which is the tens place "1"), and the remaining bits are 4 (4'b0100), which, together, form the sum "1 4".

2.1 Implementation

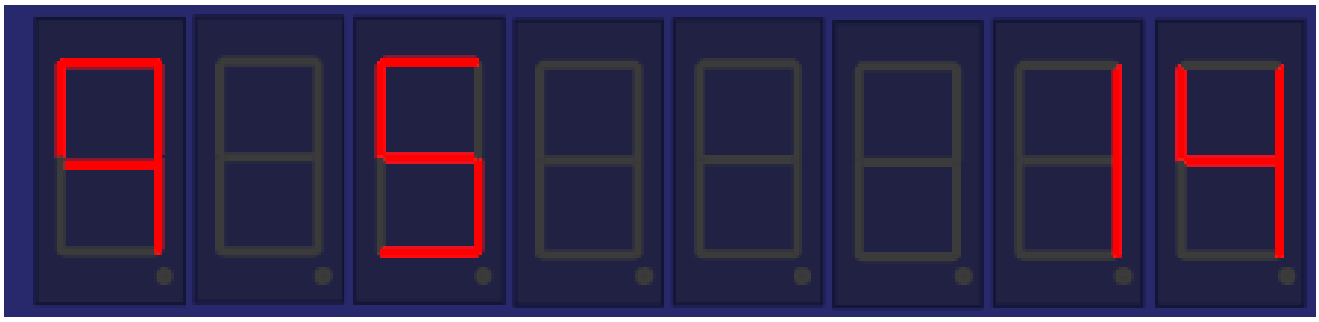
Implement the **bcdadd1** module so that it has the following ports:

Port	Type	Length
A	input	4
B	input	4
Cin	input	1
S	output	4
Cout	output	1

To test this module, you'll need to create four instances of **ssdec**, so that they display the following:

- The 4-bit input A.
- The 4-bit input B.
- The 1-bit carry-out Cout (so the display should show 1 OR 0 only)
- The 4-bit BCD sum S.

Connect them in a way so that there's space between **A** and **B**, and the **Cout** and **S** are next to each other. An example is shown below.



2.2 Verification

Take one of the testbenches you were given for an adder module and adjust it so that it checks for BCD addition rather than regular binary addition. If the sum and carry out are not as expected, print out both the incorrect output and the expected value.

A simple approach may be to use a nested for loop inside the initial block, changing both A, B and Cin.

Your **bcdadd1** only needs to have correct sums from 0-19 - it's only capable up to that point. In a design, you would only use a BCD adder to test BCD inputs (i.e. digits from 0 to 9) anyway, so your testbench only needs to check those inputs, not 10-15.

Calculating the BCD sum can make the testbench write-up a little hard to follow. Remember to use functions! That way, if you wrote your check condition as:

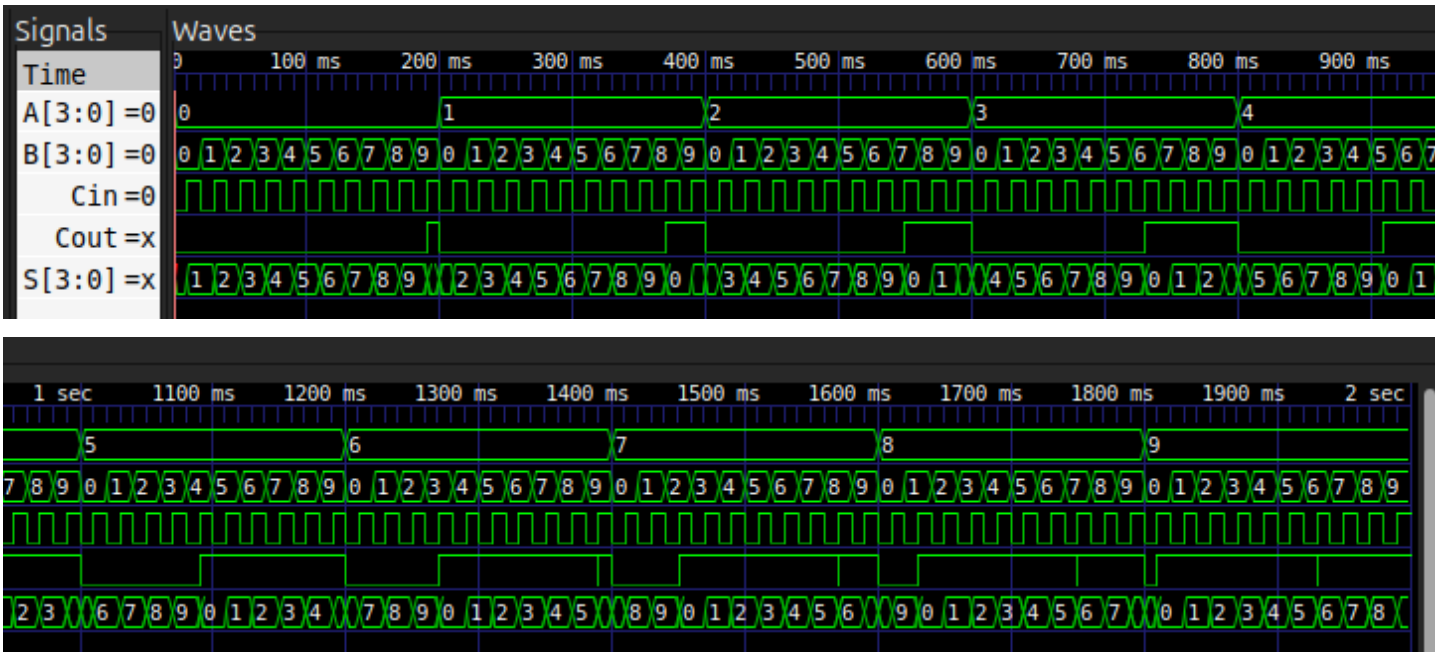
```
if ({Cout, S} == A + B + Cin))
```

And you created a new function **bcdsum**, you could merely change that to:

```
if ({Cout, S} == bcdsum(A, B, Cin))
```

Remember to add the function **inside** the module, not outside!

An example of what you might see (this is from our solution testbench, cropped into two pictures so you can read it):



Because our **input space** (the number of input bits) has become so large, we need to rely more on the print-outs from our testbench. Look back at the terminal to determine if there were any issues. That being said, an absence of print-outs does not necessarily mean it is working correctly - use the waveforms to additionally confirm that the outputs make sense for those inputs (and that you are cycling through all possibilities of the inputs).

Troubleshooting: If you see an error that says like:

```
*** These modules were missing:
      bcdadd1 referenced 1 times.
***
```

Even when the module is already in top.sv, you may have to comment out **ssdec** in **top.sv**. This has to do with the compiler not understanding which module it should compile to be simulated. You can also explicitly tell it to compile **bcdadd1** if you open the Makefile and edit the "`$(YOSYS) -p`" line to add "`-top bcdadd1`" after "`synth_ice40`".

Checkpoint L6C2: On the FPGA simulator, **demonstrate to your peer mentor/instructor** how your new **bcdadd1** module implements the "BCD correction". Demonstrate your new testbench, and show the waveforms to prove that you cycled through all possible inputs from 0-9 for A and B, and 0-1 for Cin, and that your BCD adder correctly handled all of them. Show how your testbench works against your peer's BCD adder, and vice versa. **Make sure to document everything in your lab journal.**

Step 3: Implement and verify a 20-to-5 priority encoder

No, this is not a typo - instead of 32 (2^5) inputs, we only need **20** to connect the 20 push buttons on the FPGA board (**pb[19:0]**). With this module, we can create designs that need to know what button was pushed.

*FYI - the **top** module has 21 inputs (**pb[20:0]**) in the port header, but only 20 push buttons. There exists a 21st pin on the FPGA, **pb[20]**, that we never actually connected to anything. You should ignore it.*

3.1 Implementation

The module name should be **enc20to5**, and it should have the following ports:

Port	Type	Length
in	input	20
out	output	5
strobe	output	1

The **strobe** output should go high any time if any of the bits of **in** goes high. (In case you forgot - see the lecture for an easy way to write this.)

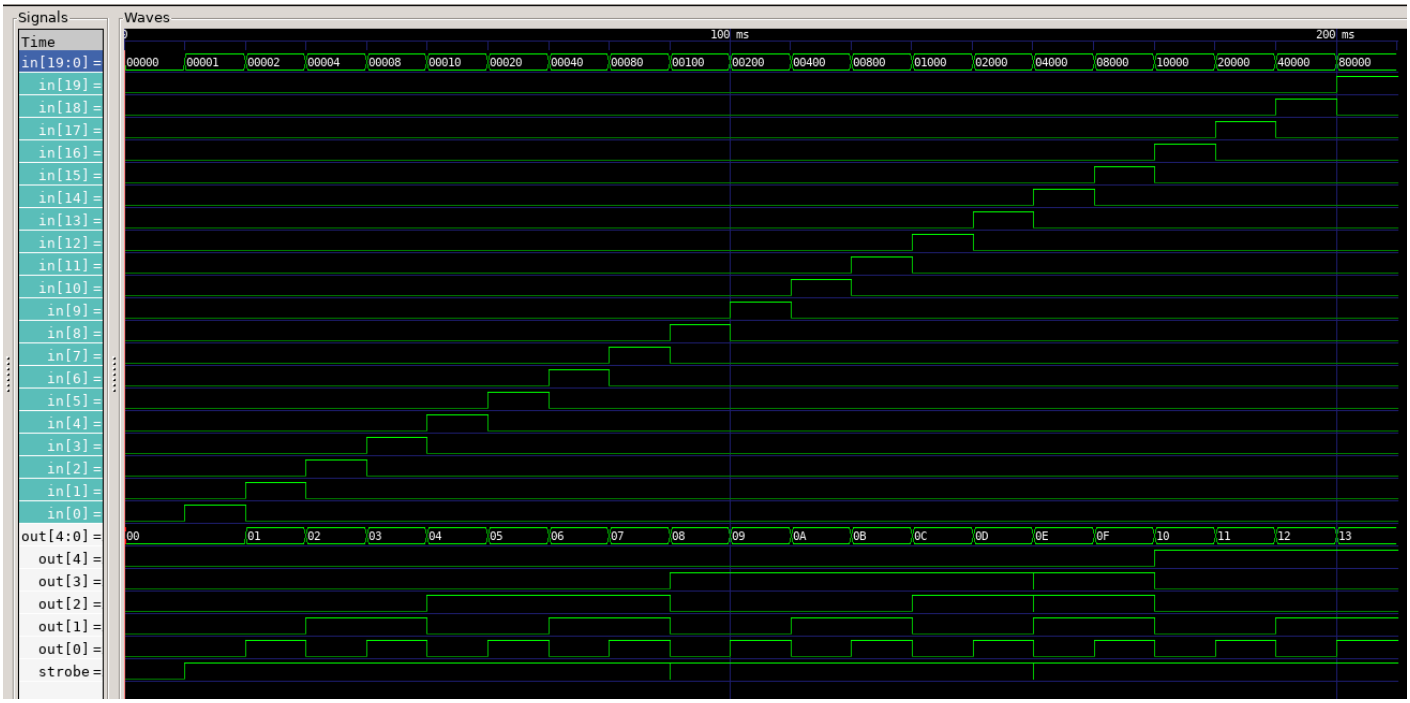
To start testing, instantiate it in your **top** module, with all the push buttons as inputs, **out** to **right[4:0]**, and **strobe** to **red**. (Consider using two **ssdec** instances to display **out** with BCD correction so that you see the decimal value, not the hexadecimal).

Your encoder should only set **out** to the index of the highest selected bit of **in**, so if you held down **pb[19]** (Z on the physical board) and **pb[4]** at the same time, the value of **out** should be 19 (5'b10011), and not 5'b10111 (which is what would happen with a regular encoder).

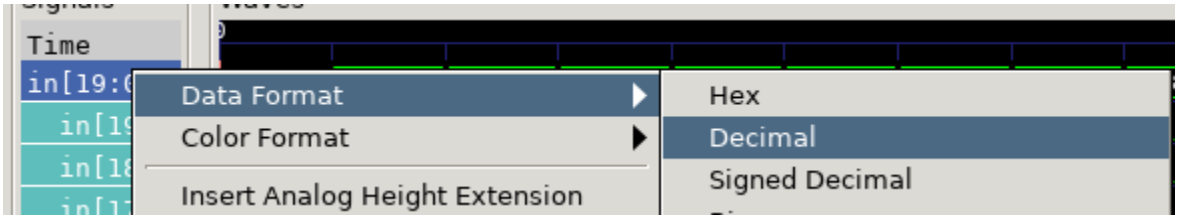
3.2 Verification

Create a testbench to test "pressing" each button individually, and to test multiple buttons. **Don't try to test the entire input space, or you will run 1048576 cases!** Instead, consider adding 2-3 cases to the testbench after the loop. Consider sets of inputs where the priority and regular encoders have different outputs (like the one above where you press 19 and 4).

Here's how our testbench looks (without the custom cases). You may have to zoom in for this one.



Keep in mind that by default, signals/buses in GTKwave are displayed as hexadecimal. Change this by right-clicking the signal added to the Waves viewer, clicking Data Format, and then clicking Decimal. You should be seeing a range of values from 0 to 19.



We'll work on this module a bit more when we get to sequential logic.

Checkpoint L6C3: Demonstrate to your peer mentor/instructor that your encoder correctly produces the corresponding binary value on **out**, and that the strobe only goes high when a button is pushed. Show that your testbench exercises all pushbuttons, and explain your custom cases and why you chose them. Test it against your peer's encoder and vice versa. **Make sure to document everything in your lab journal.**