EC311 – Introduction to Logic Design
Laboratory 4 (Lab 4)
Select Verilog Concepts and Standalone Practice Problems

Due 11/29

In this lab, there are 5 problems over 3 different general areas. The goal of this lab is to get you to practice creating a variety of different programs that use both different Verilog concepts as well as constructs. The deliverables for each problem are provided in each section.

Pacing

This lab is effectively available from 10/31 to 11/29. That gives you 30 days in theory to work on these problems or 6 days (almost a week) per problem. **HOWEVER**, you will also have your class Project to work on at this time. **The Project will be released on 11/13**. Therefore, a successful strategy would be to finish this in 14 days if possible allowing you 2+ days per problem. If things slip, you should still be able to comfortably finish this. Whatever the case may be, coming up with a schedule early is highly recommended.

Deliverables

Students should demo all testbenches in simulation and debouncer on the board to a lab TA; then turn in one zip file to Gradescope containing the deliverables listed for each problem.

Problem Area 1: Finite State Machines

Problem 1: Moore State Machine

The provided file **"moore_FSM.v"** is meant to simulate the finite state machine diagrammed below. However, it contains many mistakes and pieces of missing code.

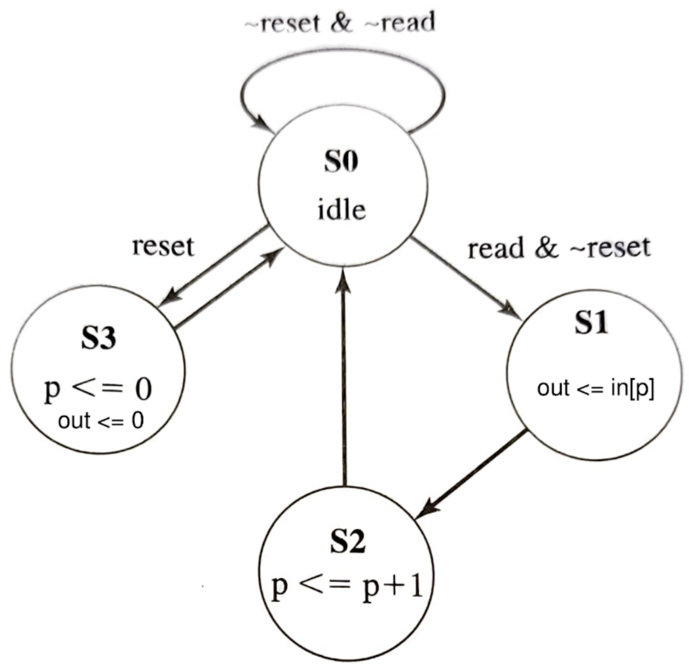
Task 1: Correct/complete the module so that it is **synthesizable** and behaves like the state machine in the diagram.

Notes:

It should update the state on the positive edge of the clock

The initial state should be S3 (be careful how you initialize the state variable)

Task 2: Create a testbench that shows each state working correctly. You may want to include your state variables in the scope of the testbench to easily demonstrate the state transitions.



Deliverables for this problem: Include 2 files in the zip file you turn in. Your corrected state machine (moore_FSM.v*)* and the testbench (moore_FSM_tb.v).

Problem 2: Mealy State Machine

In this problem, you will be creating a finite-state machine in Verilog from scratch. We recommend you use what you learned from the previous problem to structure your FSM module, but you are welcome to deviate from that structure if you wish.

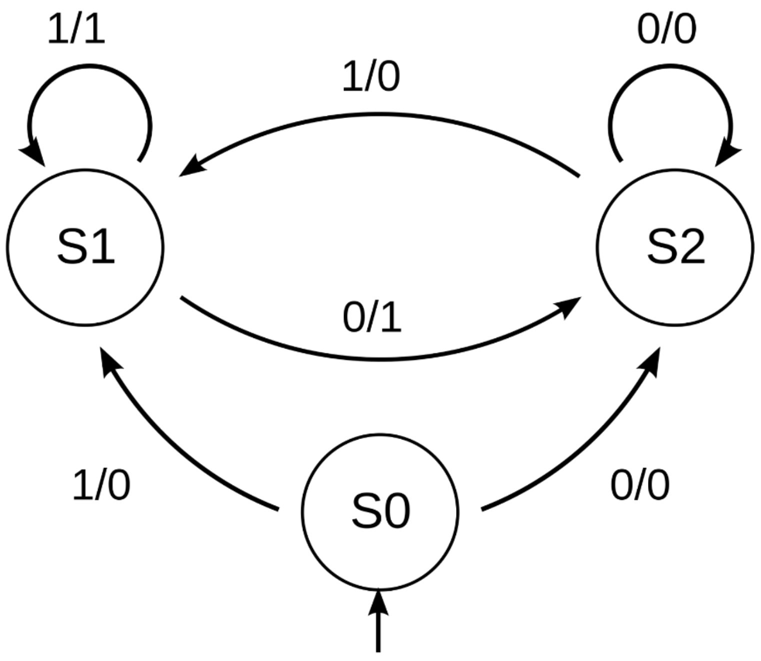
Task 1: Create a module called "mealy_FSM.v" that replicates the state diagram below

Notes:

Your module should have a 1-bit input (in) and a 1-bit output (out)

Update the state on the positive edge of the clock

Task 2: Create a testbench that demonstrates all state transitions and outputs. You may want to include your state variables in the scope of the testbench to easily demonstrate the state transitions.



Deliverables for this problem: Include 2 files in the zip file you turn in. Your state machine module (mealy_FSM.v*)* and the testbench (mealy_FSM_tb.v).

Parameterized/Generative Modules

Problem 3: Basics of Parameterized Modules and Verification Logic In this problem you will be learning the basics of creating parameterized modules in Verilog and using verification logic to improve your testbenches.

Parameterized Modules: In Verilog, you can add parameters to your modules which allows you to easily change how that module is instantiated. For example, the module below acts as a bitwise inverter of size n. Based on the value of the parameter it generates n not gates and connects them to the appropriate input and output bits.

```
module Nbit_not #(parameter n = 1)(a, out);

input [n-1:0] a;
output [n-1:0] out;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        not inverter(out[i], a[i]);
    end
endgenerate

endmodule
```

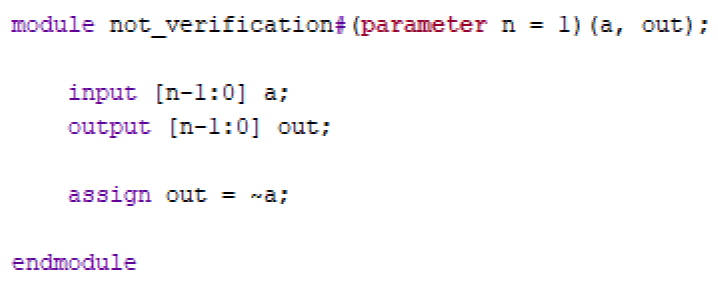
This is a good example of the basic syntax for parameterized modules. The first parenthetical in the module declaration lists the parameter's name and default value which can be overridden when instantiating the module. The generate block uses loops to generate the module logic based on the value of the parameter. Below you can see how to instantiate this module into a 64-bit inverter.

```
module inverter_64bit(
    input   [63:0] a,
    output  [63:0] out
    );

    Nbit_not #(64) inverter(.a(a), .out(out));

endmodule
```

__Verification Logic__: When creating large or complex circuits in Verilog it can be difficult to manually determine if they are working correctly from testbench waveforms alone. We can add verification logic to our testbenches to automatically flag errors in our results. In this case, we will be testing that our parameterized structural inverter works correctly by comparing it to a behavioral inverter module shown below.



```
module not_verification#(parameter n = 1)(a, out);

    input [n-1:0] a;
    output [n-1:0] out;

    assign out = ~a;

endmodule
```

In our testbench below we will instantiate the unit under test (the parameterized behavioral inverter) as well as the verification module as 64-bit inverters and connect them to the same input but separate outputs. We will then use an error flag to indicate if the output of the UUT does not equal the output of the verification module and we will print which inputs cause an error to the TCL console.

```
module inverter_tb;

    reg  [63:0] a;       //64 bit input
    wire [63:0] out;     //64 bit output for our UUT

    wire [63:0] ver_out;//output for the verification module
    wire error_flag;     //flag that will be 1 if out does not match ver_out

    //instantiate the UUT
    Nbit_not #(64) inverter(.a(a), .out(out));

    //instantiate verification module
    not_verification #(64) verification(.a(a), .out(ver_out));

    //assign error flag
    assign error_flag = (out != ver_out);

    //print errors to the console
    always@(out) begin
       if(error_flag)
          $display("Error occurs when a = %d", a);
    end

    //initialize input
    initial begin
        a = 0;
    end

    //perform exhaustive test of all possible inputs
    always begin
        #10 a = a + 1;
    end

endmodule
```

Task 1: Use the full adder module from **your previous labs** to create a parameterized n-bit ripple carry adder using a generate block to connect the full adders

Notes:
You should have two n-bit inputs and a n + 1-bit output to accommodate the final carry-out

Your initial carry-in can be hardwired to 0

Task 2: Create a verification module that uses behavioral Verilog to add two 32-bit numbers and output a 33-bit number

Task 3: Create a testbench that instantiates your parameterized module as a 32-bit ripple carry adder as well as the verification module

Notes:
Add an error flag and verification logic that will print which input values caused errors.
Perform an exhaustive test of all possible input combinations and check that this does not produce any errors

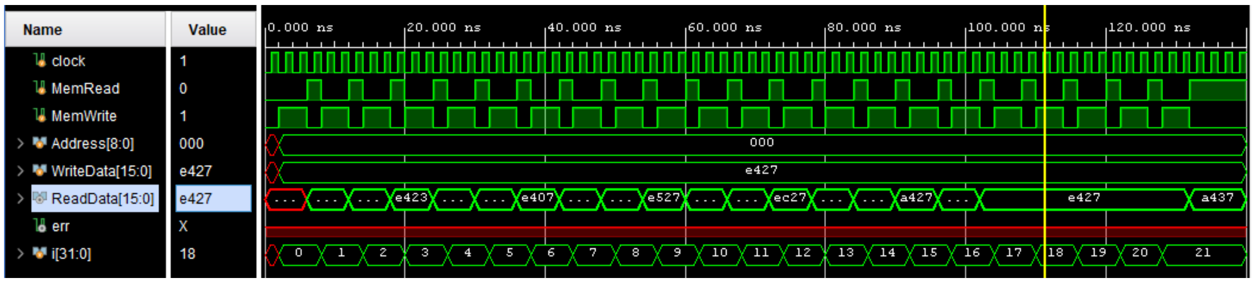
Deliverables for this problem: Include 3 files in the zip file you turn in. Your parameterized ripple carry adder (RCA_Nbit.v), the verification module you instantiate in your testbench (RCA_verification.v), and your testbench (RCA_32bit_tb.v)

Applications

Problem 4: Error Detection and Correction. *Read section 7.4 in your textbook*. In this problem, you will implement the Hamming code error correction mechanism described in this section for a simple memory module.

- **a.** The provided *hamming_memory* module implements a simple read-write memory with 512 16-bit registers. Note that these **registers are actually 22 bits**, but there are **only 16 bits of data**. The remaining 6 bits are parity bits that you will calculate. When *MemWrite=1*, the module is in write mode and writes the input *WriteData* into the memory location indicated by *Address*. When *MemRead=1*, the module is in read mode and reads the memory location indicated by *Address* into the output *ReadData* register.

- **b.** The provided *hamming_memory_test* module simulates memory corruption. **Do not modify this testbench.** It writes some data into *Address 0x0*, flips one of the bits, and then reads from the same address. Clearly, the data read back is different from the data we wrote previously. See this Figure:



- c. Your job is to **set 6 parity bits to detect and correct the bit being flipped**. There are two steps:

  . Locate the if block that handles the write mode. Instead of just writing the data into memory, calculate 5 parity bits according to the following chart:

For example, say we want to write the 16-bit data 0011 0101 1111 0000
We should populate the first 21 bits like this:

Bit #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	P_1	P_2	0	P_4	0	1	1	P_8	0	1	0	1	1	1	1	P_{16}	1	0	0	0	0

P_1 = XOR of bits (3, 5, 7, 9, 11, 13, 15, 17, 19, 21)
P_2 = XOR of bits (3, 6, 7,10,11, …)
P_4 = XOR of bits (5, 6, 11, …)
…
The XOR operands are always the bit positions whose binary representation has the corresponding bit turned on. If that's confusing, please look at the table here: https://en.wikipedia.org/wiki/Hamming_code#General_algorithm

This will give you the first 21 bits of the memory register. The final (22nd) bit is the XOR of the other 21 bits.

ii. **Locate the if block that handles the read mode.** Please change the code to do error detection and correction:

We now need to calculate the parity check bits according to the same process as the write operation:

C_1 = XOR of bits (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21)
C_2 = XOR of bits (2, 3, 6,7,10,11, …)
C_4 = XOR of bits (4, 5, 6, 11, …)
…

This will give you five check bits. If all check bits are 0, that means no error. If at least one of the check bits is 1 and only one bit was corrupted (just assume this for now), you should determine the position of the corrupted bit and flip it. Lucky for us, the check bits tell us in binary exactly which bit was corrupted. For example:

	C_{16}	C_8	C_4	C_2	C_1
No error:	0	0	0	0	0
Error in bit 1:	0	0	0	0	1
Error in bit 11:	0	1	0	1	1

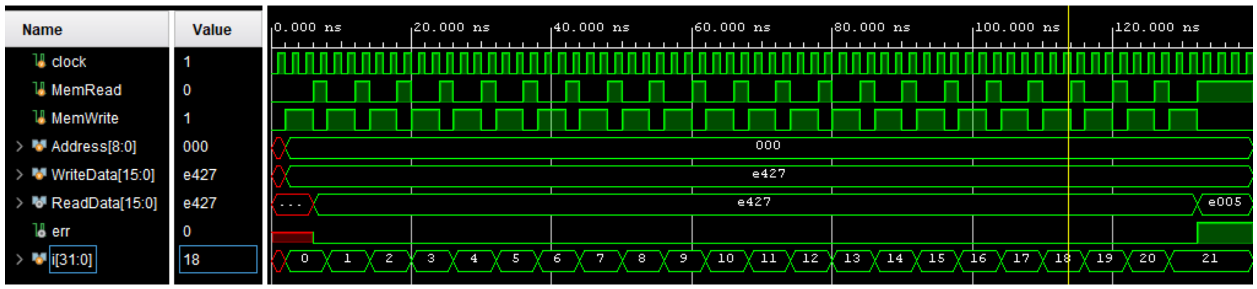
If the corrupted bit is a data bit, you should flip it before writing it to the output *ReadData* register. If the corrupted bit is a parity bit you calculated, then you don't need to do anything.

Finally, the above error correction mechanism can only handle the corruption of a single bit; it cannot handle a double-error. To mitigate this, we will set the output *err* bit if there was a double-error. You can detect a double-error using the 22nd bit you calculated in the write operation. Calculate the XOR of all 22 stored bits; let's call this P. Let's also call C = (C_{16} OR C_8 OR C_4 OR C_2 OR C_1).

- If C = 0 and P = 0, no error occurred.
- If C = 1 and P = 1, a single error occurred and you should have corrected for that already, so leave the output err as zero.
- If C = 1 and P = 0, then a double error occurred that cannot be corrected; *set the output err to one.*
- If C = 0 and P = 1, then an error occurred in the 22nd parity bit; do nothing.

Note that behavior for triple-error or higher is undefined. However, if there is a chance of the hardware making 3 errors in a 16-bit operation, you're in deep trouble anyway.

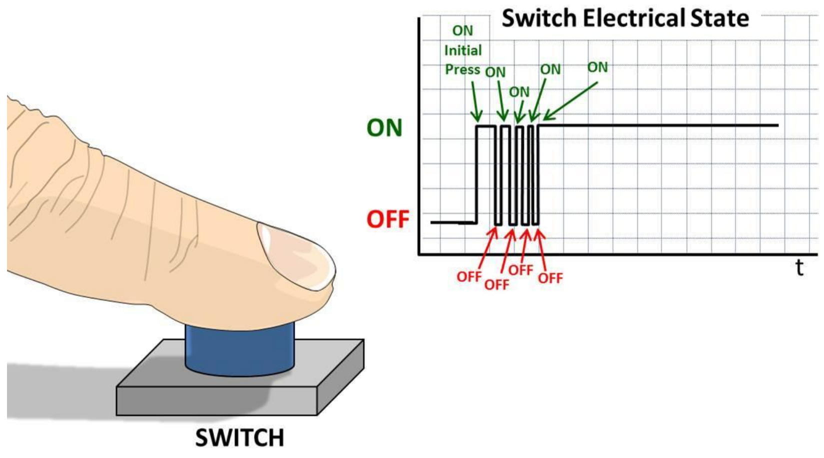
Deliverable for this problem: When finished, you should run the testbench *(hamming_memory_test.v)*. The testbench must show that the data being read back (ReadData) is the same as the data that was written (WriteData). It must also show that the *err* flag is set at the end for the double error test. See the screenshot below. Include *hamming_memory.v* and a .png screenshot of the simulation waveforms in the zip file you turn in.



Problem 5: Using Debouncers to filter noise generated by buttons: In this problem, you will learn a noise filtering technique using counters in the form of debouncers.

You will be implementing the state machine from Problem 1 in the FPGA using switches as your input "in", a button for each of the inputs "reset" and "read", and an LED for the output.

This will lead to your design to read a specific switch based on the current state and loading it into the output LED.



https://microchipdeveloper.com/xpress:code-free-switch-debounce-using-tmr2-with-hlt

As seen in the figure above, we see a limitation when pressing the button, which is that the output "bounces" for a short period of time after pressing the button. A solution to this is to use a debouncer, which counts the number of clock cycles that have passed since the initial button press until it reaches a threshold value at which the output is stabilized.

Task 1: The provided module "*debouncer.v"* contains an outline of the state machine you will need to build. Please read the comments for the specific definitions of the states and counter logic as these are what you are primarily completing. There are three states: IDLE, PRESSED, and RELEASED.

Task 2: Create a testbench to test your state machine as well. For this problem, you will need your updated code from problem 1 as well as the completed "*debouncer.v"* and the corresponding testbench.

Deliverables for this problem: Include 2 files in the zip file you turn in. Your completed debouncer module (debouncer.v*)* and the testbench (FSM_debounced_tb.v).