ECE 4514 Fall 2019: Homework 1

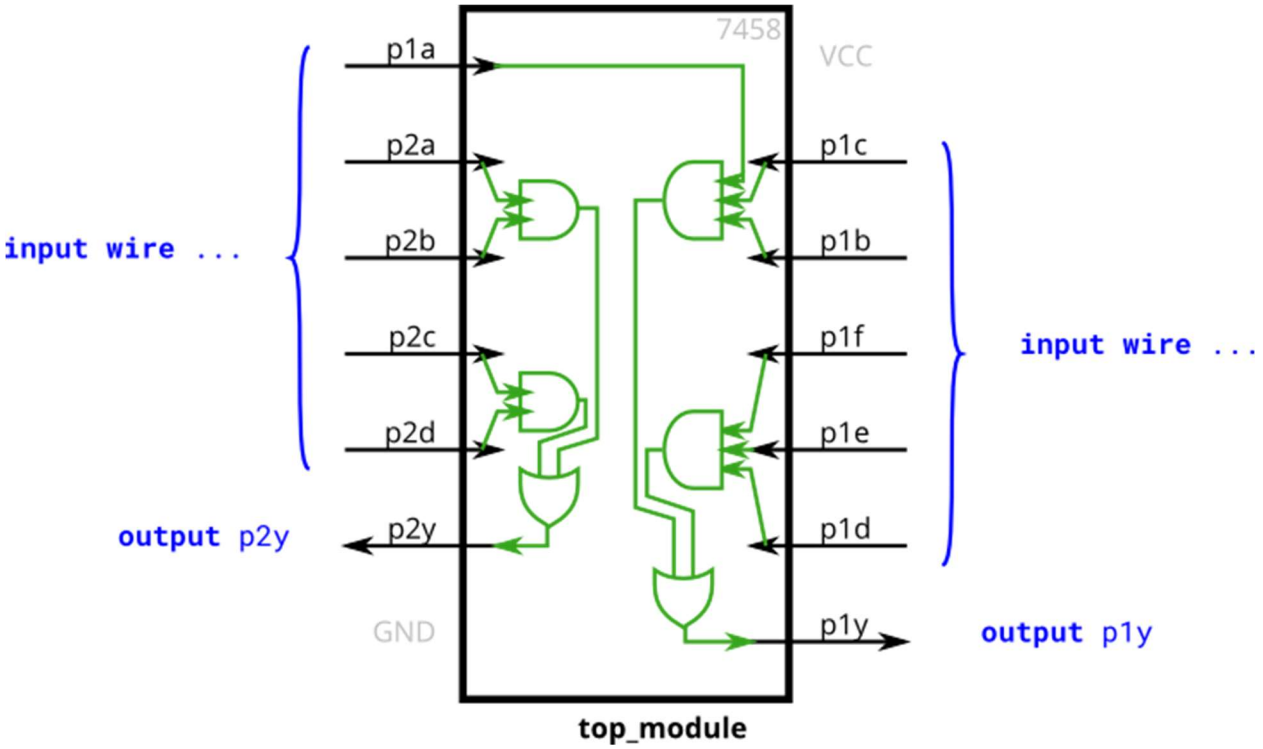
Assignment posted on January 24
Assignment due on January 31 11:59 PM

- Homework 1 is a recap on basic Verilog coding. Each problem requires you to design a Verilog module. The Verilog module has to be called **question*x*** with x corresponding to a number (1, 2, 3, ..).
- For each question, you receive a testbench. You are not allowed to make modifications to the testbench; we will verify your design with the testbench exactly as provided.
- You can evaluate the results in Modelsim, in the GUI or on the command line. The testbench will either print ERROR or else DONE depending on the correctness of your solution.
- The command line sequence to simulate the answer to question1 would be as follows:
    - Create a working library
      ```
      vlib work
      ```
    - Compile the testbench
      ```
      vlog question1tb.v
      ```
    - Compile your answer
      ```
      vlog question1.v
      ```
    - Simulate and verify
      ```
      vsim -c question1tb -do "run -all"
      ```
- Rubric

Question	Points
1	5
2	5
3	5
4	10
5	10
6	10
7	15
8	10
9	15
10	15

Question 1. 7458 Chip (5 points)

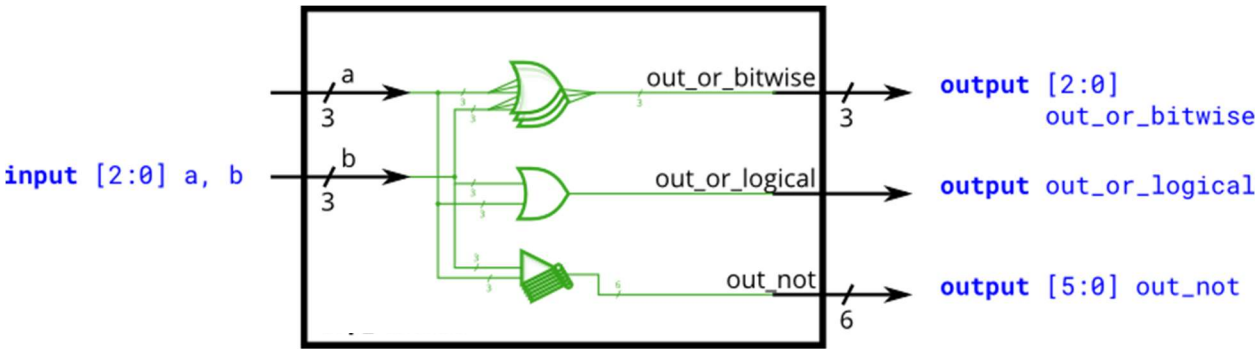
The 7458 is a chip with four AND gates and two OR gates. Create a module with the same functionality as the 7458 chip. It has 10 inputs and 2 outputs. You may choose to use an assign statement to drive each of the output wires, or you may choose to declare (four) wires for use as intermediate signals, where each internal wire is driven by the output of one of the AND gates.




```
module question1 (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y );
   // …
endmodule
```

Question 2. Bitwise vs Logical Operators (5 points)

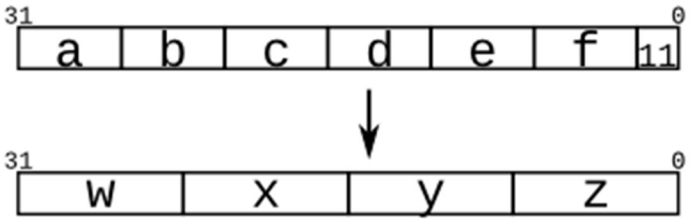
Build a circuit that has two 3-bit inputs that computes the bitwise-OR of the two vectors, the logical-OR of the two vectors, and the inverse (NOT) of both vectors. Place the inverse of b in the upper half of out not (i.e., bits [5:3]), and the inverse of a in the lower half. A bitwise operation between two N-bit vectors replicates the operation for each bit of the vector and produces a N-bit output, while a logical operation treats the entire vector as a boolean value (true = non-zero, false = zero) and produces a 1-bit output.




```
module question2(
    input  [2:0] a,
    input  [2:0] b,
    output [2:0] out_or_bitwise,
    output       out_or_logical,
    output [5:0] out_not
);
```

Question 3. Vector Concatenation (5 points)

Given several input vectors, concatenate them together then split them up into several output vectors. There are six 5-bit input vectors: a, b, c, d, e, and f, for a total of 30 bits of input. There are four 8-bit output vectors: w, x, y, and z, for 32 bits of output. The output should be a concatenation of the input vectors followed by two 1 bits:




```
module question3 (
    input [4:0] a, b, c, d, e, f,
    output [7:0] w, x, y, z );
);
```

Question 4. Sign Extension (10 points)

The concatenation operator allowed concatenating together vectors to form a larger vector. But sometimes you want the same thing concatenated together many times, and it is still tedious to do something like assign `a = {b,b,b,b,b,b};`. The replication operator allows repeating a vector and concatenating them together:

```
{num{vector}}
```

This replicates vector by num times. num must be a constant. Both sets of braces are required. One common place to see a replication operator is when sign-extending a smaller number to a larger one, while preserving its signed value. This is done by replicating the sign bit (the most significant bit) of the smaller number to the left. For example, sign-extending 4'b0101 (5) to 8 bits results in 8'b00000101 (5), while sign-extending 4'b1101 (-3) to 8 bits results in 8'b11111101 (-3).

Build a circuit that sign-extends an 8-bit number to 32 bits. This requires a concatenation of 24 copies of the sign bit (i.e., replicate bit[7] 24 times) followed by the 8-bit number itself.

```
module question4 (
    input [7:0] in,
    output [31:0] out);
```

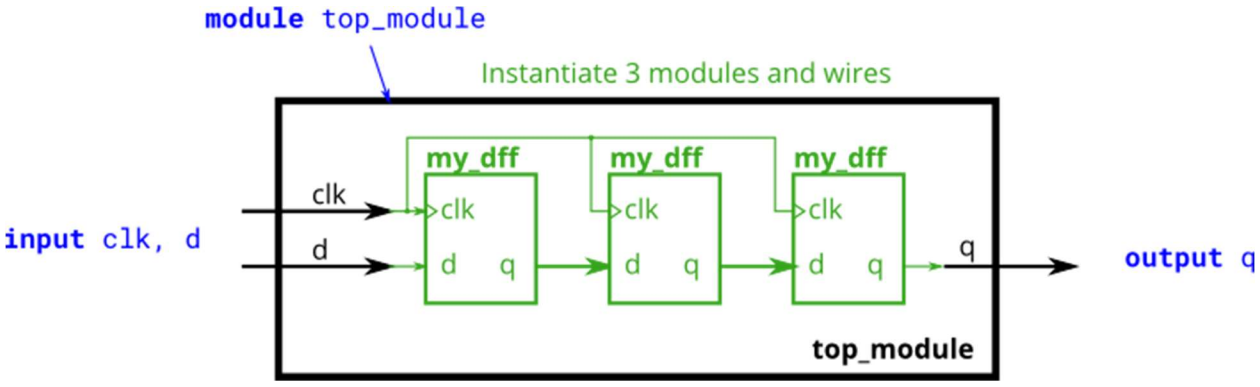
Question 5. Connecting Modules (10 points)

You are given a module my_dff with two inputs and one output (that implements a D flip-flop). Instantiate three of them, then chain them together to make a shift register of length 3. The clk port needs to be connected to all instances.

You first have to develop a module that implements the following flip-flop. The flip-flop is positive-edge triggered on the clock, and it has a negative asynchronous reset.

module my_dff (input clk, input reset, input d, output q);

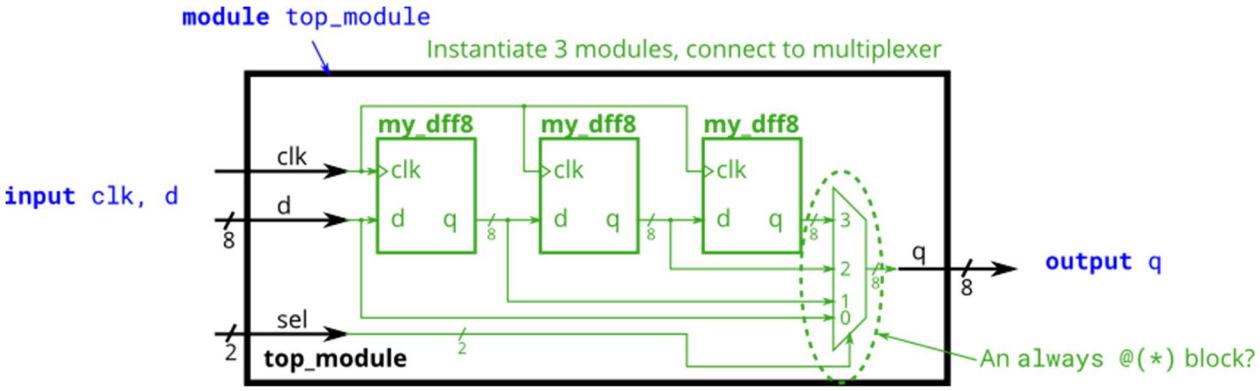
Then you can build the schematic below. Note that to make the internal connections, you will need to declare some wires. Be careful about naming your wires and module instances: the names must be unique.




```
module question5 (input clk, input reset, input d, output q);
```

Question 6. Module shift8 (10 points)

You have to design a module my_dff8 with two inputs and one output (that implements a set of 8 D flip-flops). my_dff8 has an asynchronous negative reset. Instantiate three of them, then chain them together to make a 8-bit wide shift register of length 3. In addition, create a 4-to-1 multiplexer that chooses what to output depending on sel[1:0]: The value at the input d, after the first, after the second, or after the third D flip-flop. (Essentially, sel selects how many cycles to delay the input, from zero to three clock cycles.)




```
module question6 (
    input clk,
    input reset,
    input [7:0] d,
    input [1:0] sel,
    output [7:0] q
);
```

Question 7. Module fadd (15 points)

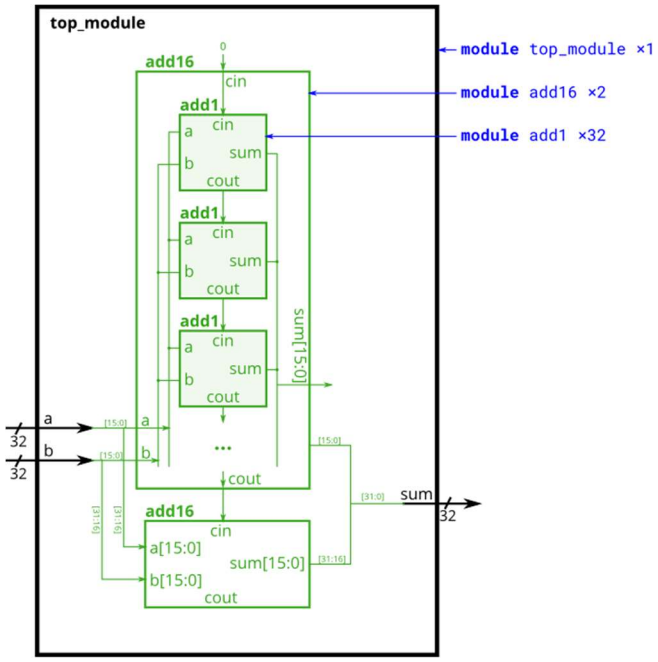
In this exercise, you will create a circuit with two levels of hierarchy that results in a 32-bit adder. Connect the modules together as shown in the diagram on the right. The full adder module has the following declaration:

```
module add1 (input a,
             input b,
             input cin,
             output sum,
             output cout );
```

Recall that a full adder computes the sum and carry-out of a+b+cin.

Within each add16, 16 full adders (module add1, not provided) are instantiated to perform the addition. Module add16 has the following declaration:

```
module add16 (input[15:0] a,
              input[15:0] b,
              input cin,
              output[15:0] sum,
              output cout );
```

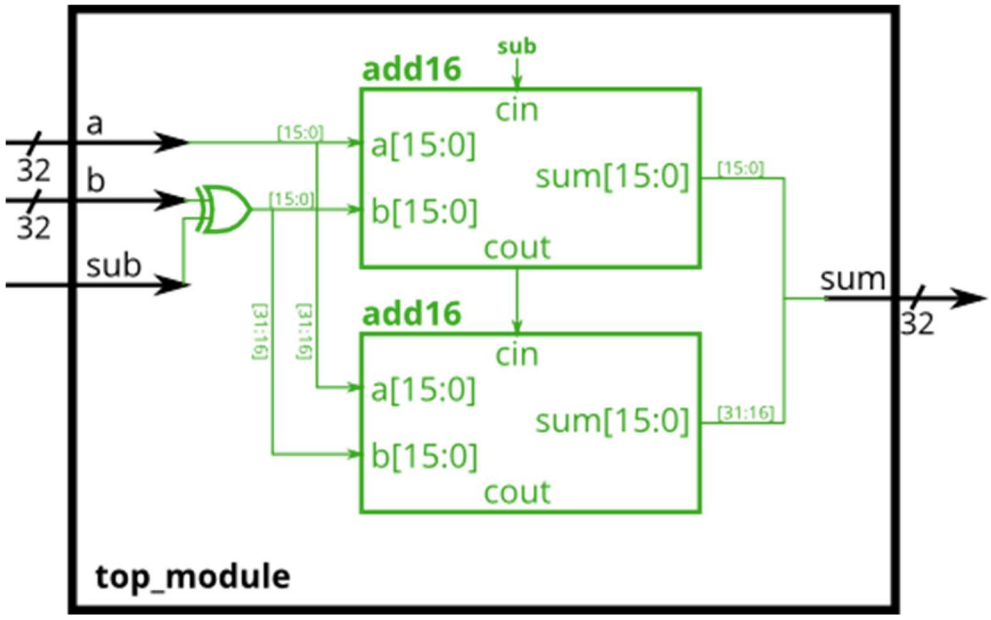



```
module question7(input [31:0] a, input [31:0] b, output [31:0] sum);
```

Question 8. Module addsub (10 points)

An adder-subtractor can be built from an adder by optionally negating one of the inputs, which is equivalent to inverting the input then adding 1. The net result is a circuit that can do two operations: (a + b + 0) and (a + ~b + 1).

Build the adder-subtractor below. Use a 32-bit wide XOR gate to invert the b input whenever sub is 1. (This can also be viewed as b[31:0] XORed with sub replicated 32 times). Also connect the sub input to the carry-in of the adder.




```
module question8(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] result
);
```

Question 9. Always block (15 points)

For hardware synthesis, there are two types of always blocks that are relevant:

Combinational: `always @(*)`
Clocked: `always @(posedge clk)`

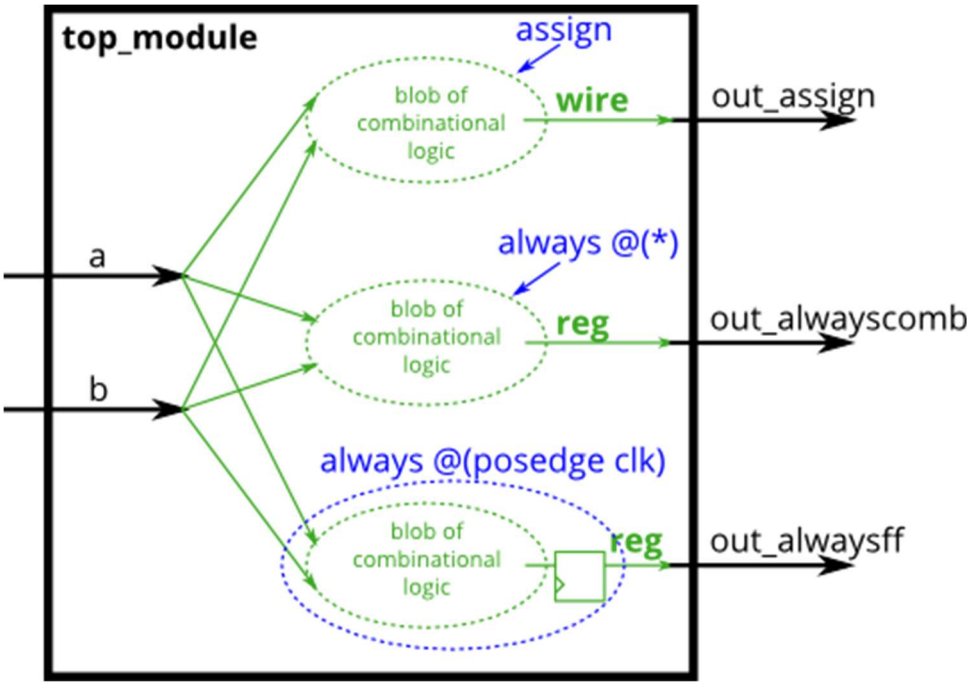
Clocked always blocks create a blob of combinational logic just like combinational always blocks, but also creates a set of flip-flops (or "registers") at the output of the blob of combinational logic. Instead of the outputs of the blob of logic being visible immediately, the outputs are visible only immediately after the next (posedge clk).

There are three types of assignments in Verilog:

- Continuous assignments (`assign x = y;`). Can only be used when not inside a procedure ("always block").
- Procedural blocking assignment: (`x = y;`). Can only be used inside a procedure.
- Procedural non-blocking assignment: (`x <= y;`). Can only be used inside a procedure.

In a combinational always block, use blocking assignments. In a clocked always block, use non-blocking assignments. We will discuss the reasons why that is, in one of the following lectures. Not following this rule results in extremely hard to find errors that are both non-deterministic and differ between simulation and synthesized hardware.

Build an XOR gate three ways, using an assign statement, a combinational always block, and a clocked always block. Note that the clocked always block produces a different circuit from the other two: There is a flip-flop so the output is delayed.



```
module question9 (
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff   );
```

Question 10. Priority Encoder (15 points)

A priority encoder is a combinational circuit that, when given an input bit vector, outputs the position of the first 1 bit in the vector. For example, a 8-bit priority encoder given the input 8'b10010000 would output 3'd4, because bit[4] is first bit that is high.

Build a 4-bit priority encoder. For this problem, if none of the input bits are high (i.e., input is zero), output zero. Note that a 4-bit number has 16 possible combinations.

```
module question10 (
    input [3:0] in,
    output reg [1:0] pos);
```