EECS 151/251A Homework 2

Due Friday, Sept 18th, 2020

For this HW Assignment

You will be asked to write several Verilog modules as part of this HW assignment. You will need to test your modules by running them through a simulator. A useful tool is https://www.edaplayground.com, a free, online Verilog simulator.

For all problems, include your Verilog code, test bench, and test results (including the simulation output and a waveform). Also explain what aspects of your design are being verified by your testbench.

Version: 1 - 2020-10-10 03:51:45Z

Problem 1: Find the Errors [4 pts]

For each module, identify the common mistake in the code and explain how it will cause the module to function incorrectly. Some useful resources:

- Verilog Primer Slides
- wire vs reg
- always@ blocks

(a) The following module is meant to be a simple 2-to-1 multiplexer with the output being set to input `a` when `sel=0`, and `b` when `sel=1`. [1 pt]

```
module mux2to1(
    input a, b, sel,
    output reg out
);
    always @(a or b) begin
        if(sel)
            out = b;
        else
            out = a;
    end
endmodule
```

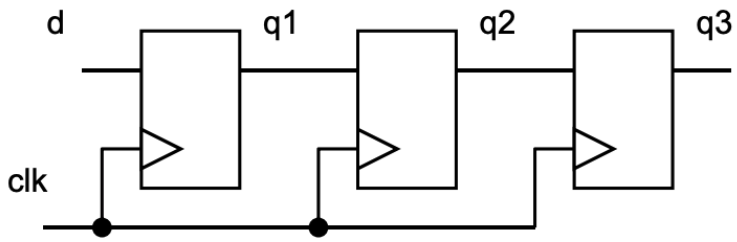
(b) In the following module, input `a` can have any value. `priority` should take the most significant bit of `a` that is 1 (if any) and set the corresponding bit of `out` to 1. All other bits should be set to 0. [1 pt]

```
module priority(
    input [3:0] a,
    output reg [3:0] out
);
    always @(a) begin
        if (a[3]) out = 4'b1000;
        else if (a[2]) out = 4'b0100;
        else if (a[1]) out = 4'b0010;
        else if (a[0]) out = 4'b0001;
    end
endmodule
```

(c) The following module is meant to be a 3-input `and` gate built from 2-input `and` gates. [1 pt]

```
module and3(
    input a, b, c,
    output reg y
);
    reg tmp;
    always @(a or b or c) begin
        tmp <= a & b;
        y <= tmp & c;
    end
endmodule
```

(d) The following code is meant to be for a sequential pipeline register. Diagram of the intended design: [1 pt]




```
module pipeline_reg (
    input clk,
    input[7:0] d,
    output [7:0] q3
);
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q1 = d;
        q2 = q1;
        q3 = q2;
    end
endmodule
```

Solution:

(a) Incorrect sensitivity list.

`sel` is missing from the sensitivity list. When `sel` changes, the output of the mux won't change.

(b) Inferred latch.

There is no if statement for when `a` is 0. Because we are using a combinational `always` block, all cases of `a` should be accounted for. Since there is a case missing, the software will introduce a latch where there was previously no state, so that `out` keeps its previous value when `a` is 0. This is called an inferring or inferred latch.

(c) Incorrect non-blocking assignments.

Because the module uses non-blocking assignments, if `a` or `b` changes, `y` will be set based on the old value of `tmp`, rather than the current value of `a & b`. This will result in incorrect `and` gate functionality.

(d) Incorrect blocking assignments.

Because blocking assignment is used, the assignments effectively wire `d` to `q1` to `q2` to `q3`. The synthesized design will end up behaving like a single register with input `d` and output `q3`.

Problem 2: One-hot to Binary Encoder [4 pts]

One-hot encoding is an alternative to binary encoding. For example, in 8-bit one-hot encoding, the number 0 is represented as 00000001, the number 1 is 00000010, 2 is 00000100, 3 is 00001000, ..., 7 is 10000000. It is called "one-hot" because only 1 bit is ever on at a time. Compared to other encodings (like binary), one-hot encoding in hardware trades additional bits of state for reduced decode logic. In particular, this is useful for finite state machines where one flip-flop can represent each state. Consider a circuit to convert 2-bit binary numbers to one-hot code.

(a) Write a Verilog module that implements a one-hot code to binary number converter. This converter takes in 8 bits and outputs 4 bits. If the input is a legal one-hot code, the lower 3 bits of the output are the corresponding binary number and the 4th bit is set to 0. If the input is not a legal one-hot code, we do not care about the lower 3 bits, and the 4th bit is set to 1. Use Verilog continuous assignment to describe the circuit. [2 pts]

(b) Write a testbench to verify the behavior of the converter. Make sure to test multiple cases where the input is a legal one-hot code, and multiple cases where the input is not a legal one-hot code. [2 pts]

Solution:

Note that this is just one possible example of a correct design/testbench.

Design:

```
module onehot_to_binary(
    input [7:0] in,
    output [3:0] out
);
    // Check if in is a power of 2
    assign out[3] = in && ((in & (in-1)) == 0);

    // Convert one-hot to binary (this value doesn't matter if input is
    ↪  not one-hot)
    assign out[2:0] = $clog2(in);
endmodule
```

Testbench:

```
`timescale 1ns/1ns
module onehot_testbench();
  reg [7:0] onehot;
  wire [3:0] binary;
  onehot_to_binary otb (onehot, binary);

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1, onehot_testbench);

    $monitor("one-hot = %b", onehot, " | binary = %b", binary );
    onehot = 8'b00000000; #1;
    onehot = 8'b00000100; #1;
    onehot = 8'b00100000; #1;
    onehot = 8'b00011000; #1;
    onehot = 8'b00000000; #1;

    $finish();
  end
endmodule
```

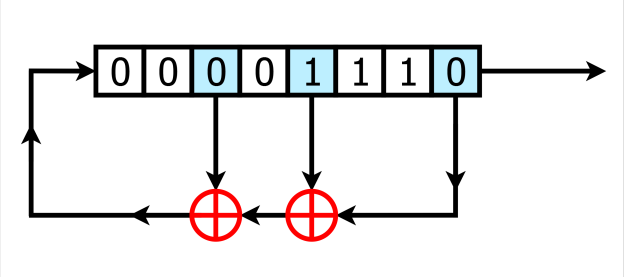
Simulation output:

```
one-hot = 00000000 | binary = 0000
one-hot = 00000100 | binary = 1010
one-hot = 00100000 | binary = 1101
one-hot = 00011000 | binary = 0101
```

Waveform:

Problem 3: Linear-Feedback Shift Register [5 pts]

In this exercise, you will implement an 8-bit linear feedback shift register (LFSR). An LFSR generates pseudo-random numbers using bitwise operations. Applications of LFSRs include digital TV, CDMA cellphones, Ethernet, USB 3.0, and more!



On each rising edge of the clock, you will shift the contents of the register 1 bit to the right. On the left side, you will shift in a single bit equal to the Exclusive Or (XOR) of the bits originally in position 0, 3, and 5. Note that in the diagram, the leftmost bit is bit 7 and the rightmost bit is bit 0.

(a) Write a Verilog module to implement the above listed operation. The module should include a reset signal that initializes the register to the following value: `8'h01`. [2.5 pts]

(b) Write a testbench to verify the behavior of the LFSR. Check that first few outputs from the shift register match what you calculate by hand. [2.5 pts]

Solution:

Design:

```
module lfsr (
    input clk, reset,
    output reg [7:0] out
);
    /* Set initial value
    *   Not required, since we can assume reset has been asserted before
    ↪  we check the output
    */
    initial begin
        out <= 8'h01;
    end

    wire feedback;
    always @(posedge clk) begin
        if (reset)
            out <= 8'h01;
```

```
            else
                out <= {feedback, out[7:1]};
        end
        assign feedback = out[5] ^ out[3] ^ out[0];
    endmodule
```

Testbench:

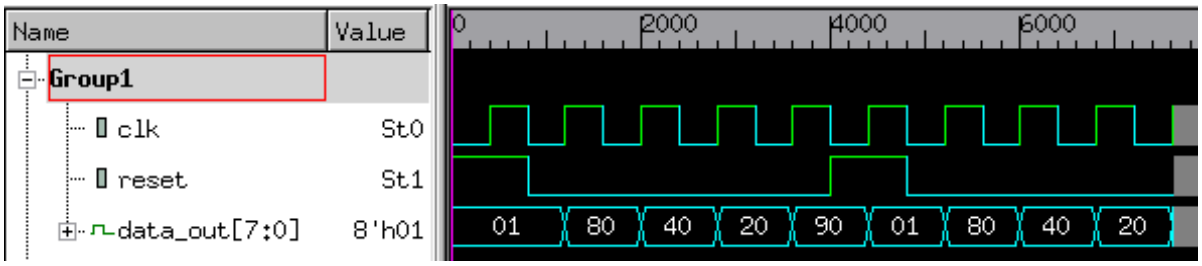
```
    `timescale 1ns/1ns
    module lfsr_testbench;
      reg  clk, reset;
      wire [7:0] data_out;

      //Set the initial state of the clock
      initial clk = 0;
      // Instantiate device under test
      lfsr lfsr(.out(data_out),
                .reset(reset),
                .clk(clk));

      //Every 4 timesteps (1ns/step) flip the clock
      always #(4) clk <= ~clk;
      initial begin
        // Dump waves
        $dumpfile("dump.vcd");
        $dumpvars(1, lfsr_testbench);

        clk = 0;

        reset = 1; #8 reset = 0;
        $monitor("data_out = %b", data_out);
        #32;
        reset = 1; #8 reset = 0;
        #32;

        $finish();
      end
    endmodule
```

Simulation output:

```
    data_out = 00000001
    data_out = 10000000
    data_out = 01000000
    data_out = 00100000
```

```
data_out = 10010000
data_out = 00000001
data_out = 10000000
data_out = 01000000
data_out = 00100000
data_out = 10010000
```

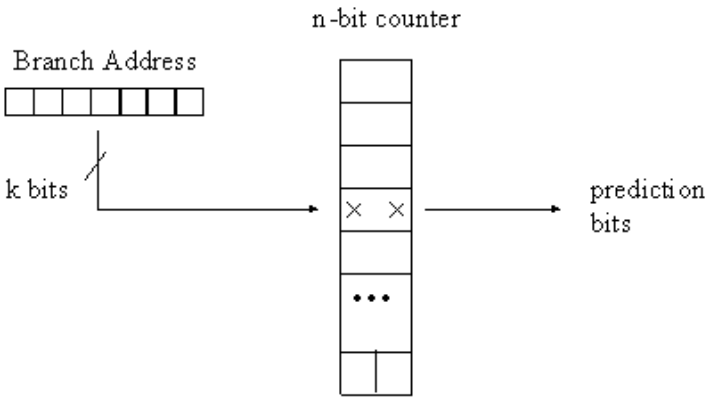
Waveform:



Problem 4: Branch History Table [6 pts]

One type of CPU instruction is a branch instruction. A branch instruction is effectively an if statement that determines whether we will go to a different part of our code (be "taken"), or continue executing code at the current location ("not taken"). As we will learn later in this class, in pipelined CPUs it is useful to predict whether a branch instruction will be taken or not. If we predict correctly that the branch should be taken, then the CPU can go ahead and start fetching instructions from the new location in the code, improving performance.

Each CPU instruction has an address, like a line number in a computer program. To predict whether a given branch instruction will be taken, we can create a buffer called a Branch History Table (BHT):



In this BHT, each row is indexed by the lower bits of the instruction address. Each row stores a 2-bit saturating counter that indicates whether the branch was recently taken. Whenever we find out whether a previously executed branch instruction was taken or not, we update the corresponding row in our table. If the branch was taken, the counter increments; if it was not taken, the counter decrements. Each counter "saturates" at 0 and 3, meaning it cannot decrement or increment beyond these values. We predict whether new branch instructions will be taken or not taken based on the upper bit of the counter in the corresponding row—if it is 1, we predict taken, if it is 0, we predict not taken. With around 4000 entries, even a branch predictor this simple can achieve 80-90% prediction accuracy on real programs.

(a) Write a Verilog module that implements a 2-bit branch history table with a configurable number of entries. Each entry should be set to `2'b01` when a reset signal is asserted. Here is an example of the parameters and ports that should be in your design:

```
module bht #(
    parameter table_size = 4096,
    parameter addr_len = $clog2(table_size)
)(
    input clk, rst,
    input [addr_len-1:0] addr_update,
    input update, taken,

    input [addr_len-1:0] addr_predict,
    output take_branch
);
```

In this example, the table would be updated on each rising edge of `clk` according to `addr_update`, `update`, and `taken`, which indicate the address of a previous instruction, whether it was a branch instruction, and whether it was taken, respectively. The output `take_branch` would be set according to whether a branch instruction at address `addr_predict` should be predicted taken or not taken. Your design should function identically. [3 pts]

(b) Write a testbench to verify the behavior of the BHT. [3 pts]

Solution:

Design:

```
module bht #(
  parameter table_size = 4096,
  parameter addr_len = $clog2(table_size)
)(
  input clk, rst,
  input [addr_len-1:0] addr_update,
  input update, taken,

  input [addr_len-1:0] addr_predict,
  output take_branch
);
  reg [1:0] mem [table_size-1:0];

  /* Note: We assume we are not passed in the lowest 2 bits of the
  *  address, as otherwise 3/4 of table rows are wasted.
  */
  assign take_branch = mem[addr_predict][1];

  integer i;
```

```
    always @(posedge clk) begin
      if (rst) begin  // Reset all rows to 2'b01
        for (i = 0; i < table_size; i = i + 1) begin
          mem[i] = 2'b01;
        end
      end
      else begin      // Update table
        if (update) begin
          if (taken) begin
            if (mem[addr_update] < 2'b11) begin
              mem[addr_update] <= mem[addr_update] + 1;
            end
          end
          else begin
            if (mem[addr_update] > 2'b00) begin
              mem[addr_update] <= mem[addr_update] - 1;
            end
          end
        end
      end
    endmodule
```

Testbench:

```
    `timescale 1ns/1ns
    module bht_testbench ();
      reg clk, rst;
      reg [11:0] addr_update;
      reg update, taken;
      reg [11:0] addr_predict;
      wire take_branch;
      bht #(
        .table_size(4096)
      ) b (
        clk, rst,
        addr_update,
        update, taken,
        addr_predict,
        take_branch
      );

      initial clk = 0;
      always #(4) clk <= ~clk;

      initial begin
```

```
        // Dump waves
        $dumpfile("dump.vcd");
        $dumpvars(1, bht_testbench);

        rst = 1;
        addr_update = 12'd0;
        addr_predict = 12'd0;
        update = 0;
        taken = 0;

        // Test update/taken logic
        #40; // 8 time steps is one clock cycle
        rst = 0;
        // Prediction starts as not taken
        $display("Should be 0: %d", take_branch);
        update = 0;
        taken = 1;

        // Prediction unchanged with update=0
        #40;
        $display("Should be 0: %d", take_branch);
        update = 1;
        taken = 1;

        // With update=1 and taken=1, prediction takes 2 cycles to change to
        ↪  taken
        #4;
        $display("Should be 0: %d", take_branch);
        #8;
        $display("Should be 1: %d", take_branch);
        #8;
        $display("Should be 1: %d", take_branch);

        // Other predictions should still be 0
        addr_predict = 12'd4; #2;
        $display("Should be 0: %d", take_branch);

        // Do some unrelated work
        addr_update = 12'd20;
        #40;

        // With update=1 and taken=0, prediction takes 2 cycles to change back
        ↪  to not taken
        update = 1;
        taken = 0;
        addr_update = 12'd0;
        addr_predict = 12'd0;
```

```
        #16;
        $display("Should be 0: %d", take_branch);

        $finish();
      end
    endmodule
```

Simulation output:

```
Should be 0: 0
Should be 0: 0
Should be 0: 0
Should be 1: 1
Should be 1: 1
Should be 0: 0
Should be 0: 0
```

Waveform: