# Tug of War: Scorer

The purpose of this lab is to get your hands dirty with verilog and the FPGA. More formally, these are the steps in the process and their associated tools: You can do the entire lab just with Xilinx ISE. It incorporates the associated tools need for each process mentioned below.

a) Verilog Design Entry (Register Transfer Level (RTL) Coding) – Modelsim
b) Writing a Testbench to stimulate the design – Modelsim
c) Testing the RTL code- Modelsim or Modelsim from within ISE
d) Converting the RTL code to a gate level netlist then a downloadable .bit file (Xilinx ISE)
e) Testing the gate level netlist - Modelsim or Modelsim through ISE
f) Downloading the .bit file to the FPGA and testing it - Impact
g) Documenting what you did, what your design looks like, how many resources it takes, etc.

## 1 Scorer :

Our design is the scorer module for the TOW game. The scorer accepts 3 inputs and adjusts the score of the game accordingly. On the FPGA, there are 3 push buttons, and 7 lights (or leds). There is also a clk input, fed by a function generator.
A Xilinx ISE tutorial can be accessed from the lab website. The tutorial gives step by step instructions on the following.

a) Starting Xilinx Project Navigator
b) Creating a new project
c) Using ISE
d) iMPACT Device Configuration

Open and set up the tool according to the instructions given. Make sure all the parameters are set to appropriate values.

### 1.1 Verilog Design (Register Transfer Level (RTL) Coding)

In this design we use the winrnd to signify the someone has pushed a button, right to signify that right button was pushed if the value is 1 and that the left button was pushed if the value is 0, and leds_on the signify that the lights were on when the buton was pressed

| FPGA Pin name: | In our design, used for: |
|---|---|
| input clk; | clock |
| input rst; | reset - clears the score to the default value |
| input winrnd; | win round – someone has won the round the score must be updated |
| input right; | right – Right pressed for logic High, Left pressed for logic low |
| input leds_on; | leds on – the LEDS were on when the button was pressed |
| output [6:0] score ; | Score output |

Below, and completely out of order, are all the code snippets which implement the up/dn counter. You MUST be able to understand them.

The synchronous part of this counter contains only the registers, synchronous state assignment.

```
always @(posedge clk or posedge rst)
        if(rst) state <= `N;
        else state <= nxtstate;
```

We use a separate section of combinational logic to compute what value the score should take on in the next clock cycle, based on the inputs.

```
always @(state or mr or leds_on or winrnd)
begin
nxtstate = state;
    if(winrnd) begin
            if(leds_on)      // Proper pushes (uses favour the loser options)
                case(state)
                `N:     if(mr) nxtstate = `R1; else nxtstate=`L1;
                …
                `R3:    if(mr) nxtstate = `WR; else nxtstate=`R1;
                …
                default: nxtstate = `ERROR;
                endcase
            else            // the leds were off, player jumped the light
                case(state)
                …
                default: nxtstate = `ERROR;
                endcase
    end
  end
```

The last section of the module determines the output depending on the current state of the FSM.

```
always @(state)
        case(state)
            `N:     score = 7'b0001000;
            …
            `WR:  score = 7'b0000111;
            default: score = 7'b1010101;
        endcase
```

The full version of the code can be downloaded from the course directory as score_template.v

**Q1)** Complete the code for the next state assignment being sure to include the favor the loser case.

**Q2)** Complete the code for the output logic.

**Q3)** Review the code and complete any unfinished sections.

## 1.2 Writing a Testbench to stimulate the design; ISE, Modelsim, or your favorite editor

The design entry stage is now complete and we need to verify that everything was coded correctly. This requires writing a test-bench which stimulates the design, and monitors its responses. The testbench is normally more important, and harder to write, than the design itself. Keep in mind that it does not need to be synthesizable and so the whole range of the verilog language is available. Study the code below and make sure you under-

stand it. It is the simplest kind of testbench. Note the two golden rules of testbenches!

1) Never change an input on a clock edge!
2) Only look at outputs when they are valid!

```
module updncounter_tb;
        reg leds_on, winrnd, rst, right, clk;          //inputs to your circuit are declared as registers
        wire [6:0] score;                              //outputs from your circuit are declared as wires
        always #10 clk <= ~clk;                        //toggles the clock every 10 time units
reg [6:0] score_reg;                                   // a SAMPLED version of the device output
        always @(posedge clk or posedge rst)
                if(rst) score_reg <= 0;
                else    score_reg <= score;

initial begin                                          //All initial statements start from the same time,
        clk = 0; rst = 0;  right = 0;leds_on = 0;winrnd = 0;        //initialize all inputs to something

        @(posedge clk);                                //wait for the first clock edge
        #1; rst=1;                                     //wait till after the clock settles, turn on the reset
        @(posedge clk);                                //wait for another clock edge
        #1; rst=0;                                     //turn off the reset

        @(posedge clk); #1;                            //we should see 10 cycles of the counter's reset value
        // Check the result from the last test
        if(score_reg == 7'b0001000) $display("%t - SUCCESS. Score was %b", $time, score_reg);
        else                $display("%t - ERROR. Score was %b", $time, score_reg);
        // setup inputs for next test
        $display("%t - Setting up for Left jumping the light.", $time);
        right = 0;          leds_on = 0;      winrnd = 1;
        @(posedge clk); #1;//set the values back to zero to simulate the button being pressed not held
        right = 0;          leds_on = 0;      winrnd = 1;
        //wait to see if the score changed accordingly
        wait(score_reg == 7'b0000100); $display("%t - SUCCESS. Score was %b", $time, score_reg);

        $finish;
end
// every time the clock falls, print out the value of the score
always @(posedge clk) $display("%t - CLKSAMPLE: Score sampled to be %d", $time, score_reg);

// set up statements to inform you when inputs change
always @(rst) $display("%t - DATAMONITOR: rst signal changed to %b", $time, rst);

// and finally instantiate the device under test (DUT)
scorer scorerinst (.clk(clk), .rst(rst), .leds_on(leds_on), .right(right), .winrnd(winrnd), .score(score));

endmodule
```

The above testbench can be downloaded from the course directory as ***scorer_tb.v.***

**Q4)** The above testbench has a parallel process in it which spits out a statues indication when the reset line changes. These are commonly referred to as ' monitors'. Add code to the testbench which monitors the winrnd, leds_on, and right signals also.

**Q5)** Complete the test bench to simulate every possible state transition, use your pre-lab as a reference for the possible transitions. If you are still unsure where to start try these transitions; left pushing frist from L2, right pushing first from L2, left pushing from R3, right pushing first from R3, right pushing first from R3.

## 1.3 Gate Level Design and Simulation

RTL code only simulates the behavioural function of the hardware you are designing. To account for things such as delays and real device performance, a gate level simulation can be performed to verify timing making use of the models of the gates used in the target device.

To run a gate level simulation:

a) Select the test bench in the Simulation pane of the Design panel.
b) From the drop down menu select Post-Route.
c) Expand the ModelSim Simulator branch in the Processes pane.
d) Double click Simulate Post-Place & Route Model.

Note that ISE will automatically synthesize the implementation files as well as perform other functions. You can see this in the Implementation pane by the green check marks indicating successful completion of the function. The gate level design will then be simulated in ModelSim. Note that this takes somewhat longer than the behavioural simulation.

**Q6)** Simulate the final gate level code, and prove that it works WARNING FREE by printing out the log file and a waveform view.

## 1.4 Requirements

**DEMONSTRATE** your working Scorer to the TA by the end of the lab, and answer the following questions.

a) In the test bench we used the sampled score not the score itself, why was it done this way opposed to using the score itself?
b) Did you have make any changes to your test-bench to get the gate level simulation to work?
c) At the end of the test-bench we instantiate the design. There are two syntaxes for instantiating an object. In the testbench above, each port is explicitly stated in a .port(connection) format. What is the other option? What are some advantages/disadvantages of each?

If you have demonstrated your working code and answered the questions above you **do not** have a lab report to submit. However if you do not finish within this lab you must email your working code and the answers to the questions to the your TA and you will not receive the checkout marks for this lab.