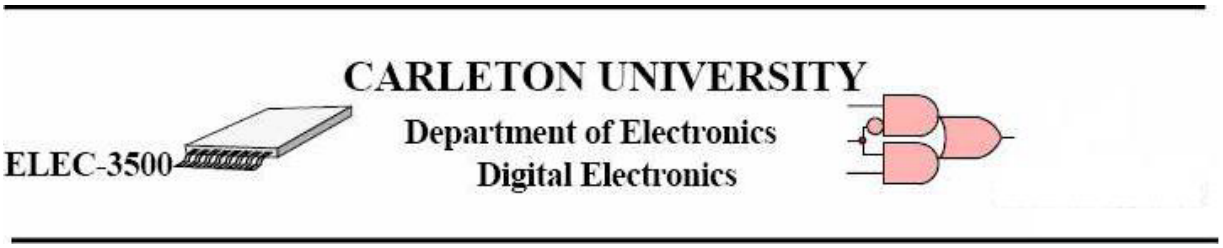

Tug of War: Up/Down Counter

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 Up Down Counter :

Our design is a counter which either holds, counts up, or counts down based on two control buttons. 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)

We will use the middle push-button for reset (denoted rst), the left push-button (pbl) to count down, and the right pushbutton(pbr) to count up.

FPGA Pin name:	In our design, used for:
input clk;	clock
input pbl;	countdown control - the counter counts down when pressed
input rst;	reset - clears the counter to the defined value
input pbr;	count up control - the counter counts up when pressed
output [6:0] leds_out ;	counter 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.

```
always @(posedge clk or posedge rst)
        if(rst) counter <= 127;                 //default notation is in decimal
        else counter <= next_count;
```

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

```
always @(counter or cntdwn or cntup)
begin
        next_count = counter;
        if(cntdwn&~cntup) next_count = counter - 1;
        if(~cntdwn&cntup) next_count = counter + 1;
end
```

But, there is a slight complication in that we can't use the count-down or count-up signal directly from the push-buttons. The reasoning will be covered more in the lectures. We need to feed them through flip-flops first.

```
always @(posedge clk or posedge rst)
        if(rst) cntdwn <= 0;
        else cntdwn <= cntdwn_from_pushbutton;
        always @(posedge clk or posedge rst)
        if(rst) cntup <= 0;
        else cntup <= cntup_from_pushbutton;
```

We need to call the main pins in and out of our design the same as they are on the FPGA board, but in our design they use different names, and so we must perform the mappings.

```
assign cntdwn_from_pushbutton = pbl;        //map them to the external names
assign cntup_from_pushbutton = pbr;         //map them to the external names
assign leds_out = counter;                  //note that this will map all 7 bits
```

The full version of the code can be downloaded from the course directory as ***updncounter.v***. To ensure designs are not creatively acquired from one-another, the counter should ***reset to a value assigned by your TA***.

Q1) Modify the reset condition of the code to reset the counter to a value as assigned by your TA.

Q2) Modify the code to load the counter with an error tag (ie. 1001001) when BOTH button are pushed.

Q3) Draw a logic diagram which includes all inputs, outputs, internal signals, registers, and 'clouds' which indicate sections of combinational logic.

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 that it is VERY IMPORTANT not to change input values at the same time the clock changes. It makes it confusing for both the simulator and the human to determine which values the design is working with.

```
module updncounter_tb;
        reg pbl;                                    //inputs to your circuit are declared as registers
        reg rst;
        reg pbr;
        reg clk;
        wire [6:0] leds_out;                        //outputs from your circuit are declared as wires

        always #20 clk <= ~clk;                     //toggles the clock every 20 time units

initial begin                                       //All initial statements start from the same time, t=0.
        clk=0; rst=0; pbl=0; pbr=0;                 //initialize all inputs to something

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

        repeat(10) @(posedge clk);                  //we should see 10 cycles of the counter's reset value
        #5; pbr =1;

        // Now, the counter should start merrily counting up, wait until it reaches 69 and switch directions
        wait(leds_out==69);
        $display("%t - TESTBENCH: The counter has reached 69", $time);
        @(posedge clk);
        #5; pbl = 1; pbr= 0;
        $display("%t - TESTBENCH: Switching Directions", $time);
        wait(leds_out==0);
        $display("%t - TESTBENCH: The counter has reached 0 - Finishing.", $time);
        $finish;
end

// every time the clock falls, print out the value of the leds
always @(posedge clk) $display("%t - CLKSAMPLE: Leds sampled to be %d", $time, leds_out);

// 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)
updncounter updncounter_instance(.clk(clk), .rst(rst), .pbl(pbl), .pbr(pbr), .leds_out(leds_out));

endmodule
```

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

Q4) The above testbench has a parallel process in it which spits out a status indication when the reset line changes. These are commonly referred to as 'monitors'. Add code to the testbench which monitors the pbl and pbr signals also.

Q5) Simulate your rtl code, as modified, and prove that it works by printing out the log file and a waveform view.

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 by printing out the log file and a waveform view. What change did you have to make to your test-bench to get this to work?

Q7) **(post-demo)**: Near 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?

Q8) How many of the LUTs are used by your circuit, and how many are still available. (Hint: In the Xilinx tool this is part of the Mapping process.)

1.4 Requirements

DEMONSTRATE your working UP/DN counter to the TA by the end of the lab. **REMEMBER** it must reset to your assigned value.

Submit with your lab report:

a) Cover page with the dated signature of the TA who witnessed the demo.
b) The modified test-bench and rtl codes.
c) Answers to all questions which should including:
   - a waveform and log file of the top-level RTL simulation covering reset and the first few counts up.
   - a waveform and log file of the final gete-level simulation covering reset and the first few counts up.
d) The gate-level schematic for your circuit (note: this is available from within Xilinx).