Computer Architecture Lab 05

Introduction to FPGA, Verilog, and State Machines – Blinking LEDs & Button Input

Introduction

This lab is intended as an exercise in gaining familiarity with synthesis tools available in FPGAs in order to read/write input and output devices available and design state machines accordingly.

Objectives

* Learn to read input from FPGA buttons.
* Design a counter with user configurable delays with variable reset limits.
* Display output using seven segment LED on FPGA.

Sections

|  |  |
| --- | --- |
| Section |  |
| a) Task 1 | **40** |
| You will learn how to take button input to change the state of internal variables. |
|  |
| c) Task 2: | **40** |
| In this section, you will develop modules to write outputs to output peripherals such as Seven Segment LED on the FPGA. |
|  |
| d) Task 3: | **100** |
| In this section, you will develop a counter with user-configurable delay values. |
|  |

Task 1: Preprocessing Button Input

|  |
| --- |
| Why is processing necessary?  An ideal button press is one that pulses as a clean signal when a button is pressed:      Real-time signals do not work this way. When you press a mechanical button, the metal contacts physically **bounce** before settling. This bounce causes the voltage to fluctuate rapidly between 0 and 1 for a few milliseconds:Verilog code for button debouncing  In such a case, logic that does not cater to this might register multiple button presses, and cause undefined behavior. |

We make extensive use of delays and added functionality for debouncing.

1. Develop a module that makes inputs a parameter delay (learn more about parameters as input [here](https://www.chipverify.com/verilog/verilog-parameters)), and outputs a periodic pulse after the recorded delay value. Test your code using a testbench.

|  |  |
| --- | --- |
| Code | Testbench |
|  |  |

1. Develop a debouncing module to process your input, and test its functionality using the following top module:

|  |
| --- |
| `timescale 1ns / 1ps  module topButton (  input clk, btnC,  output reg led  );  initial led = 0;  wire buttonPressed;    // your module  debouncer myDebouncer(  .clk(clk),  .pbin(btnC),  .pbout(buttonPressed)  );  always @(posedge clk) begin  if(buttonPressed) led <= ~led;  end  endmodule |

The button is thus able to vary the state of an LED by turning it on and off. Remember to set up a constraint file to connect relevant top module pins to hardware. You may learn more about writing a constraint file [here.](https://github.com/Digilent/digilent-xdc)

Provide your code here:

|  |
| --- |
|  |

Task 2: State Machine in a Seven Segment Blinking LED

|  |
| --- |
| Why is a state machine necessary?  A seven-segment display on the Basys 3 board actually contains four individual digits, but all of them share the same segment control lines (the pins that light up segments a–g). The only difference is that each digit has its own enable pin (AN0–AN3), which selects which digit is currently active.  Because of this shared wiring, you cannot display four different numbers simultaneously by simply assigning values or all digits would show the same segments.  Instead, we rely on a time-multiplexing technique:   * We rapidly switch (or “cycle”) between digits, lighting up one digit at a time. * Each digit displays its respective value for a very short duration. * This switching happens so fast (hundreds or thousands of times per second) that to the human eye, all digits appear to be glowing continuously.   This is where a finite state machine (FSM) becomes essential. The FSM controls which digit is active and what value is being displayed at that instant.  How the FSM helps:  Each state in the machine corresponds to one of the digits (for example, *state 0 → digit 1*, *state 1 → digit 2*, and so on) and the FSM transitions between these states in a loop, enabling one digit at a time and assigning its segment value. A delay counter (exactly like the one you made before) provides the timing signal that triggers the state changes at a steady rate.  You can learn more about designing FSMs [here](https://www.chipverify.com/verilog/verilog-fsm%23google_vignette)  Your FSM could look something like this: |

Provide FSM module here:

|  |
| --- |
|  |

|  |
| --- |
| Seven Segment LED as memory mapped I/O:  The seven-segment display can be treated like any other output device i.e. it has registers that hold the data to be displayed, and your system writes to those registers using normal store (write) operations.  As such, a memory mapped output module interfaces with design logic as follows:  module segTop(  input clk,  input rst,  input [31:0] writeData,  input writeEnable,  input readEnable, // not to be used (output device)  input [63:0] memAddress, // not to be used (output device)    output reg [31:0] readData = 0, // not to be read (output device)  output [6:0] seg, // 7-segment segments (a-g)  output [3:0] an  );  The LED is able to display only the lower 16 bits of writeData, since each of the four digits needs 4 bits (one hexadecimal digit per display).  This structure mimics how real processors handle display peripherals; you simply “write” to a memory location, and the display hardware takes care of presenting that value visually. |

Compile the delay module and the FSM to make a seven segment top module that is compatible with the following top module:

|  |
| --- |
| `timescale 1ns / 1ps  module topSevenSegmentCheck(  input clk,  output [6:0] seg, // 7-segment segments (a-g)  output [3:0] an  );    reg [31:0] writeData = 32'h00002345;  wire readData;    segTop display (  .clk(clk),  .rst(1'b0),  .writeData(writeData),  .writeEnable(1'b1),  .readEnable(1'b0),  .memAddress(64'b0),  .readData(readData),  .seg(seg),  .an(an)  );  endmodule |

Add all relevant modules you used here:

|  |
| --- |
|  |

Task 3:

Now that you know how to manipulate input and output devices, and generate delays from the bus clock, it is time to design a custom counter with user-configurable delay.

Your task is to design a counter that runs for 20s before resetting to zero. Your inputs are:

* A reset button: This must reset the counter to zero, and the delay value to default (1s).
* A delay increment button: This must increase the delay between a counter increment by 1s.

The maximum value of the counter before it resets should be calculated to ensure that it runs for a maximum of 20s for whatever delay value.

Note that your delay is no longer a constant, and cannot be passed as a parameter input to the delay counter.

Provide all relevant modules not submitted before below:

|  |
| --- |
|  |