**ELC 5313: ADVANCED Computer Architecture**

**Fall 2025**

**Lab 1: Introduction to SystemVerilog**

**Introduction**

SystemVerilog is a hardware description language (HDL). Unlike C or Python, which run on a processor, HDLs describe circuits: modules, wires, and registers that behave like hardware.

* Modules: represent chips or components (adders, counters, flip-flops).
* Inputs/Outputs: define how data enters and leaves a module.
* Simulation: we use testbenches to provide inputs, apply a clock, and observe outputs. This lets us verify a design before building it in hardware.
* Key concepts:
  + Logic replaces wire/reg in SystemVerilog.
  + always\_ff: sequential logic (needs a clock, stores state).
  + always\_comb: combinational logic (instantaneous updates).
  + assign: shorthand for simple combinational connections.
  + Blocking (=) vs Non-blocking (<=): use non-blocking for sequential logic to avoid race conditions.

You will see these concepts in action in the example adder code (see Appendix).

Before beginning this lab:

* Read the ELC 5313 Lab Startup Instructions document.
* Follow the steps to clone the course GitHub repository into your Box folder.
* Make sure you have both directories set up:
  + ELC5313-OG (cloned repo – do not edit)
  + ELC5313-Lab (your working copy – where you will complete labs).

You must complete this setup to access starter files and receive updates throughout the semester.

**Scope**

In this lab, you will learn the basics of SystemVerilog by implementing a simple **modulo-3 counter**. This counter cycles through 0 → 1 → 2 → 0. You will also use a testbench to simulate and verify its behavior. This is a simple introduction lab that you will complete on your own. You will only write the logic for the counter and can otherwise observe how SystemsVerilog works.

This lab introduces:

* Writing SystemVerilog modules.
* Using testbenches for simulation.

**Tasks**

1. **Implement the modulo-3 counter**

Start from this file mod3counter.sv located in ELC5313-Lab:

module mod3counter (

input logic [1:0] num,

output logic [1:0] mod3num

);

always\_comb begin

// TODO: implement modulo-3 logic

end

endmodule

Hints:

* If the input is 0 → output should be 1.
* If 1 → output should be 2.
* If 2 → output should reset to 0.
* For values ≥ 3, default to 0.
* You may use if/else or a case statement.

1. **Implement the testbench**

Use the mod3counter\_tb.sv in ELC5313-Lab:

module mod3counter\_tb;

logic [1:0] num\_in;

logic [1:0] num\_out;

mod3counter uut (

.num(num\_in),

.mod3num(num\_out)

);

initial begin

num\_in = 0; #10;

num\_in = 1; #10;

num\_in = 2; #10;

num\_in = 3; #10;

$finish;

end

endmodule

1. **Run simulation**

* Generate a timing diagram of your counter in Vivado.
* Confirm that the output cycles correctly.
* Feel free to change the num\_in values in the testbench, as long as your program still works.

**Deliverables**

* mod3counter.sv
* mod3counter\_tb.sv
* A screenshot of your waveform simulation.

**Grading Rubric**

* **Modulo-3 Counter implementation** (60 pts)
  + The expectation is that you create a functioning modulo-3 counter. How you implement the logic does not matter.
* **Testbench correctness** (10 pts)
  + Easy points. Use the provided testbench and include it in your submission.
* **Successful simulation & waveform** (20 pts)
  + Your waveform must show that your counter worked as expected for the four values. An example of what your waveform should look like is below.

A screenshot of a computer

AI-generated content may be incorrect.

* **Code clarity & comments** (10 pts)
  + Since you are designing the counter with your preference, you must include comments explaining your approach.

**Due Date**

Your lab is due on 9/17/2025. It should be completed and turned in individually on canvas.

**Appendix: Example Code (from class demo, also located in GitHub)**

adder.sv:

module adder (

input logic clk,

input logic [3:0] a,

input logic [3:0] b,

output logic [3:0] c

);

always\_ff @(posedge clk) begin

c <= a + b; // use nonblocking assignment in sequential logic

end

endmodule

adder\_t.sv:

`timescale 1ns / 1ps

module adder\_t;

// Signals

logic clk;

logic [3:0] a;

logic [3:0] b;

logic [3:0] c;

// Instantiate the adder

adder uut (

.clk(clk),

.a(a),

.b(b),

.c(c)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk;

end

// Test stimulus

initial begin

a = 0;

b = 0;

// Cycle 1

#10 @(posedge clk);

a = 4'b0011;

b = 4'b0101; // 5 in 4 bits

// Cycle 2

#10 @(posedge clk);

a = 4'b0110;

b = 4'b0010;

// Cycle 3

#10 @(posedge clk);

a = 4'b0111;

b = 4'b0001;

// Finish simulation

#10 $finish;

end

// Monitor values

initial begin

$monitor("Time=%0t | a=%0d b=%0d | c=%0d", $time, a, b, c);

end

endmodule