Lab 8: Sequential Counter

ECE 211: Digital Circuits I — Spring 2025

Abstract. In this lab, you will build a circuit that uses sequential logic to count upwards from zero to nine. From this foundation, we can build circuits that can count increasingly larger ranges (*e.g.,* from 0 to 99 or from 0 to 999). Further, by configuring the FPGA clock period to 1 second, we can use this sequential counter circuit as a timer! The resultant counter would increment every second, recording the number of seconds that have elapsed. This lab will introduce you to sequential logic, clock signals, and the **`always_ff`** block in SystemVerilog. During this lab, you will test and verify your counter circuit in simulation and via synthesis on the Nexys A7 FPGA.

Objectives.

- Gain experience designing sequential circuits (binary counters) in SystemVerilog
- Test the operation of individual circuit components using testbenches
- Test the operation of the circuit using the input and output components on the Nexys A7

Materials and Equipment.

- Nexys A7-100T FPGA board with USB cable

Deliverables.

- In-person lab demonstration (per team)
- Lab report (per team)

Part 0: Set Up a Vivado Project

To begin the lab, we will set up a new Vivado project for Lab 8 and configure the Nexys A7 board. You can reference Lab 0 for a more detailed description of these steps.

1. Create a Xilinx Vivado project and import the provided constraints file and SystemVerilog modules. This lab will use both simulation and design sources. The file **`lab08_top.sv`** should be uploaded as a design source. The file **`lab08_tb.sv`** should be uploaded as a simulation source. You will use this file as a testbench to write test cases for each module.

2. In this lab, you will primarily use Vivado's simulator to verify your design at each part, and then use the FPGA board at the end to demonstrate the entire circuit. If you would like, you can connect your FPGA board and open the hardware target at this point in the lab.

Part 1: Design and Implement a 4-Bit Sequential Counter

Sequential circuits allow us to build devices that operate based not only on the current inputs (*e.g.,* switches) but also data stored in memory. ***D flip-flops (DFFs)*** are the most common storage element on an FPGA. Figure 1 shows a SystemVerilog implementation of a positive-edge driven D flip-flop that stores 1-bit of information for a single clock cycle. The SystemVerilog implementation uses the **`always_ff`** block and a non-blocking assignment (**`<=`**). Together, this syntax indicates that the procedural statement **`q<=d`**; should be activated whenever there is a rising edge of the **`clk`** signal. During other time units, the signal **`q`** will hold ("remember") its current state. Other logic can be combined within the **`always_ff`** block. For example, Figure 2 shows a D flip-flop with a reset operation. This reset input is needed when testing our circuit in simulation.

```
module DFF(  input logic d, clk,
             output logic q);

   always_ff @(posedge clk)
   begin
     q <= d;
   end
endmodule
```

Figure 1. D Flip-Flop in SystemVerilog

```
module DFF_R( input logic d, clk, rst,
              output logic q );

   always_ff @(posedge clk)
   begin
     if(rst) begin
        q <= 1'b0;
     end
     else begin
        q <= d;
     end
   end
endmodule
```

Figure 2. D Flip-Flop with Reset in SystemVerilog

In this part of the lab assignment, you will design and implement a 4-bit counter module. The behavior of the counter module will *count* from 0 to 9, and then return to 0. To implement this functionality, your module should store a 4-bit binary value and increment this value on each successive clock cycle. Specifically, the output of your module, **`q`**, should cycle through the states **`4'b0000`**, **`4'b0001`**, **`4'b0010`**, **`4'b0011`** … **`4'b1001`**, and then return to **`4'b0000`**. Counters are used in many different applications in digital circuits. For example, they can be used to sequence combinational circuit elements through a repetitive set of operations. The SystemVerilog code that controls the Nexys A7's seven-segment display actually uses a counter module!

3. Design and implement a 4-bit counter module.
   a. Add a new source file to your project for `counter_4b.sv`
   b. Your module should take a scalar reset signal and a scalar clock signal
   c. Your module should produce a 4-bit vector output
   d. Design and implement the module to meet the description above. Use non-blocking assignments and the addition operator in your implementation.
   e. Your implementation should include a synchronous reset that will set the counter to **`4'b0000`** on the rising edge of the clock signal if reset is HIGH.

4. In the next part of the lab, you will test your 4-bit counter module using a testbench.

Part 2: Test your Counter Module in Simulation

In order to simulate sequential circuits, we need to include code in our testbench to generate a ***clock signal***. Additionally, we will need to use the reset input to our 4-bit counter to initially set our output **q** to zero. Without reset, the initial output of our D flip-flop will be "X", meaning that all future values derived from this output will also be "X." Figure 3 shows a simple stimulus-only testbench for a 4-bit counter module named **`counter_4b`**. Each element of this testbench is described below.

```
module example_tb();
   logic clk, rst;
   logic [3:0] q;

   // Instantiate the DUT
   counter_4b DUT(.clk(clk), .rst(rst), .q(q));

   // Sequence input stimulus (specifically rst)
   initial begin
       rst = 1'b1;    // reset counter to zero
       #11;           // wait till first clock edge + 1 ns
       rst = 1'b0;    // disable reset
       #100;          // let simulation run for 10 clock cycles
       $stop;
   end

   // Generate clock (repeats!)
   always begin
      clk = 1'b0;  #5;
      clk = 1'b1;  #5;
   end
endmodule
```

Figure 3. Stimulus Testbench for a Sequential Circuit

The first section of the testbench instantiates the device under test (DUT), just as we have done when writing other testbenches in this course.

The second section, which begins with the keywords **`initial begin`**, generates the initial inputs to the DUT. Starting at simulation time 0, the first statement sets **`rst=1`**. The next line delays simulation until after the next rising. These two lines will cause the output of the D flip-flop to be reset to zero. Without this line of code, the output of the D flip-flop (**`q`**) would be uninitialized and have the value "X" on the resulting timing diagram. Further, all future values of **`q`** would also have the value "X" since the calculation of **`q`** also depends on the current value of **`q`**. After resetting the counter module, the testbench will disable reset (setting **`rst=0`**). The fourth line delays the simulation for ten clock periods, allowing the counter to run through its full set of states.

The third section, which begins with the keywords **`always begin`**, generates the clock signal for the testbench. This always statement is followed by an assignment that sets **`clk`** to zero and then delays for 5 ns. The following line of code sets **`clk`** to one and then delays for 5 ns. In other words, the clock signal is LOW for 5 ns and HIGH for 5 ns, making the entire clock period 10 ns in length (*What is the frequency of this clock signal?)*. This section uses the keyword **`always begin`** instead of **`initial begin`** because the former will start at the beginning of the simulation (timestep 0) and repeat over-and-over until the simulation ends. Essentially, this **`always begin`** block repeats, allowing us to generate a continuous clock signal without having to manually assign the clock cycle every 5 ns.

5. Test your counter module using a testbench.
   a. In `lab08_tb.sv`, instantiate a counter module
   b. Using the sample testbench above, generate a clock signal for your counter module
   c. Initialize the counter module using the reset signal in the first clock period
   d. Configure your simulation so that the counter module will cycle through all ten states and return to the value zero.

6. Simulate your circuit and verify your circuit's operation using the Wave view or by printing messages to the Tcl Console

7. Verify that your counter increments from 0-9 before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly and is triggered on the rising edge of the clock pulse.

Record: Screenshot the waveform for your 4-bit counter module to include in your Lab Report.

Part 3: Extend your Counter Module to Count from 0 to 999

Your 4-bit counter module counts from 0-9 in sync with the clock pulse. In this part of the lab, you will extend your module to count from 0-999, where each digit of the output will be represented by a 4-bit binary number. For example, the number 123 will be represented as **`4'b0001`**, **`4'b0010`**, and **`4'b0011`**. When this number is incremented on the next clock tick, it will update to the number 124 with the representation: **`4'b0001`**, **`4'b0010`**, and **`4'b0100`**. Here, the least-significant digit is simply an instantiation of your 4-bit counter from Part 2, as its value is incremented by one on each clock tick.

You can design this module in two ways: *1)* By modifying your original 4-bit counter module, or *2)* By using your 4-bit counter module as a submodule to construct this circuit. You may choose either option for design. Explain and justify your choice in your lab report.

8. Design and implement a 3-digit counter module that can count from 0-999.
   a. Add a new source file to your project for `counter.sv`
   b. Your module should take a scalar reset signal and a scalar clock signal
   c. Your module should produce three 4-bit vector outputs, corresponding to each of the three digits
   d. Design and implement the module to meet the description above. Include a reset operation as done in `counter_4b`

9. Test your 3-digit counter using a testbench.
   a. In `lab08_tb.sv`, instantiate your counter module
   b. Simulate your circuit for enough clock cycles to ensure that your implementation works correctly.
   c. Verify your circuit's operation using the Wave view or by printing messages to the Tcl Console

10. Verify your test cases and record your testbench code and results before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly and increments on the rising edge of the clock pulse.

Record: Screenshot the waveform for your counter module to include in your Lab Report.

Part 4: Connect your Counter to the Nexys A7 Board

In this part of this lab assignment, you will connect your counter module to the Nexys A7 seven-segment display to count the number of elapsed seconds. Because your counter module increments on the rising edge of the clock pulse, you will want to configure your clock to trigger every second, enabling you to count elapsed time. This design can be achieved using a clock divider and the provided clock signal on the Nexys A7 FPGA.

Sequential digital circuits require a stable clock signal at a specified frequency, *f*. Clocks are generated using an oscillator circuit that produces a signal at a desired frequency. Most digital systems use a clock generated by a ***crystal oscillator***. This circuit is built around a small quartz crystal that vibrates at a resonant frequency dependent on the physical dimensions of the crystal. When connected with appropriate electronics, the crystal can be used to generate a clock signal at that frequency. Because crystal oscillators are highly stable and accurate, they are used to generate clocks for most digital systems, ranging from watches and clocks, to smartphones and computer systems.

The Nexys A7 uses an Epson SG8002JF module to generate a 100 MHz clock signal hardwired to the FPGA. However, it is sometimes desirable to run circuits at different frequencies. One way to accomplish this is to use a specialized counter circuit known as a ***clock divider***. This circuit will create a slow clock, essentially changing the output clock signal at a slower rate than the input clock signal. In this lab, you will use a parameterized clock divider module, called **`clkdiv`**, that is provided in the lab starter code. This module takes the 100 MHz clock as input and outputs a clock signal at the frequency specified by **`DIVFREQ`**.

```
module clkdiv(input logic clk, reset,
              output logic sclk);
   parameter DIVFREQ = 100;  // desired frequency in Hz (change as needed)
   parameter DIVBITS = 26;   // enough bits to divide 100MHz down to 1 Hz
   parameter CLKFREQ = 100_000_000;
   parameter DIVAMT = (CLKFREQ / DIVFREQ) / 2;

   logic [DIVBITS-1:0] q;

   always_ff @(posedge clk)
   begin
      if (reset) begin
          q <= 0;
          sclk <= 0;
      end
      else if (q == DIVAMT-1) begin
          q <= 0;
          sclk <= ~sclk;
      end
      else begin
          q <= q + 1;
      end
   end
endmodule
```

Figure 4. Clock Divider Module

For example, the following code would instantiate the clock divider to produce a 100 Hz clock signal by setting **`DIVFREQ`** to 100. Because **`DIVFREQ`** is specified as a ***parameter***, it can be set externally to the module, as shown below. SystemVerilog will synthesize and optimize a circuit for the exact value of the parameter specified. Ultimately, the parameter allows us to write more reusable code.

```
clkdiv #(.DIVFREQ(100)) D0(.clk(clk100MHz), .reset(1'b0), .sclk(clk));
```

11. In the top-level module (`lab08_top`), instantiate a 3-digit counter module and a clock divider circuit. Configure the clock divider circuit to generate a clock pulse with a period of 1 second. Then, connect this clock to the 3-digit counter module

12. Connect the output of your counter module to the seven-segment display. The seven-segment display is programmed automatically via the `counter_display` module. You will need to connect the relevant ports of this module, as shown in the following code snippet:

```
    counter_display DISP(
        .d0( ),      /* Connect your least-significant digit here (0-9) */
        .d1( ),      /* Connect your middle digit here   */
        .d2( ),      /* Connect your most-significant digit here */
        .clk(clk), .seg_n(SEG), .an(AN));
```

13. Verify that the circuit schematic matches your proposed design. You can view the circuit schematic under "RTL Analysis/Open Elaborated Design/Schematic" in the Flow Navigator.

14. Generate a bitstream and program the FPGA. Test that your circuit functions correctly and demonstrate it to an instructor.

Stop and Check: Observe your counter module increment upwards from 0. Ensure that the final increments each second, and that the numbers increment logically.

Record: Screenshot a schematic for each module for inclusion in your lab report.

Part 5: Use a Switch to Start/Stop Counting

In this part of this lab assignment, you are challenged to further modify your counter module to take a switch input (*e.g.,* **`SW[0]`**) that pauses the counter when the switch is LOW. In other words, when the switch is HIGH, the circuit will increment on the next clock pulse. If the switch is LOW, the circuit will not increment and hold its current value, **`q`**.

This modification will allow us to start and stop the timer circuit based on a switch input. You might consider using an if-statement to implement this functionality. In addition to the start/stop switch, you could also connect the reset input of your flip-flop to a switch on the Nexys A7 board. With these input switches, your counter becomes a timer that can start, stop, and reset!

15. Modify your 3-digit counter module to take in an additional input signal. Configure the module to meet the above specification. Specifically, the counter is enabled and increments when the signal is 1.

16. Modify the top-level module (`lab08_top`) to connect this enable signal to a switch of your choosing on the FPGA board.

    a. You may also consider connecting a switch to the reset input of your counter module. This additional switch will allow you to start, stop, and reset your timer!

17. Verify that the circuit schematic matches your proposed design. You can view the circuit schematic under “RTL Analysis/Open Elaborated Design/Schematic” in the Flow Navigator.

18. Generate a bitstream and program the FPGA. Test that your circuit functions correctly and demonstrate it to an instructor.

Stop and Check: Observe your counter module increment upwards from 0, and pause when the designated switch is flipped.

Record: Screenshot a schematic for each module for inclusion in your lab report.

Part 6: Wrap-Up

Before leaving the lab, make sure to complete the following steps!

19. ***Demonstrate the operation of your circuit to the instructor.*** A portion of your lab grade will come from the lab report in addition to this demonstration.

20. Turn the Nexys board OFF and disconnect it from the computer. Make sure to save your Vivado project and sign out of your machine!

Lab Report

Lab Report. In collaboration with your lab partner, submit one lab report on Moodle that describes the lab activity and your team's circuit design. Your written report is worth 60% of your grade for each lab assignment. Your lab report should include the following sections:

- **Title Page:** Your lab report should begin with a title block that includes the course number, the title of the lab assignment, the names of each team member, and the date of the lab assignment. The title page should also include a "Statement of Collaboration" (detailed below) and an estimate of the total amount of time spent on the lab assignment. The inclusion of these components are graded, but the content of each statement is not used in determining your grade.

- **Statement of Collaboration:** Describe the contributions of each team member to the lab assignment, including writing of the lab report. What is specified in this section will not affect your grade, however it is expected that you periodically rotate roles within your lab group if you do divide work, including writing the report.

- **Introduction:** Include a brief paragraph summarizing the objectives of the lab and what your team achieved. This section should be written in your own words; it is not acceptable to copy text for this from the lab handout.

- **Design:** Include a concise yet descriptive explanation of the circuit design, following all steps in the process—from the problem specification and initial truth table, to equations and circuit schematics. Tables and equations should be typeset and clearly labeled. Technical information should be accompanied by some text narrating each step or aspect of the design.

- **Testing & Results:** Describe how you tested your circuit, including rationale for why you chose specific test cases if the circuit is not tested exhaustively. If you tested your circuit at multiple points (*i.e.,* using a testbench and then on the Nexys board), this section should include all test cases at each point. Test cases can be included in the form of tables or simulation waveforms. Results should be neatly formatted and labeled, indicating what step is being tested. This section should also include any recorded measurements, tests, diagrams, or images specified by the lab document.

- **Discussion/Conclusion:** Your discussion/conclusion section should comment on any observations about the technique you employed to get your results. Briefly describe any difficulties that you encountered and how you resolved them. Then, summarize what you concluded from your work. (It may also be helpful to think negatively, *i.e.* what your results did not show. Remember that a negative result is often as valuable as a positive result!)

  A number of technical questions may be posed in the assignment sheet. If questions are given, include your answers to them in this section.

Your lab report should be typed and submitted as a .pdf document. Only one member per team needs to submit the lab report.