Lab 4 - Counter

Prelab

Read Sections 3.2.1 - 3.2.4 of the textbook and answer the following questions:

1. When does the value of Q on a flip-flop change?
2. How can digital designers store multiple bits of data together?
3. Here is a SystemVerilog implementation of a 1-bit register (data flip-flop, aka DFF). How can it be modified to store 4 bits of data?

```
/*
 * One bit register (DFF).  On the rising edge of the `clock`
 * signal, the value of `d` is copied to `q` if `clear_n` is
 * high (inactive).  If `clear_n` is low (active), set the value
 * of `q` to 0.
 */
module RegisterOneBit(
    input logic clock,
    input logic clear_n,
    input logic d,
    output logic q
);

// Update `q` only on the rising edge of `clock` or the falling
// edge of `clear_n`
always_ff @(posedge clock) begin
    if (clear_n == 1'b0) begin
        // Set `q` to 0 when `clear_n` is low (active)
        q <= 1'b0;
    end else begin
        // Set `q` to `d` on `clock` rising edge
        q <= d;
    end
end
```

Procedure

Lab Intro Slides

Overview

Labs 3a and 3b developed a design to drive a single digit on a seven-segment display. Lab 4 will combine the 7-segment decoder block, with the 4 bit adder from previous labs and a 4 bit wide register to create a counter.

Theory of operation:

A user will, using 4 of the slide switches, indicate how much the counter should increment by when an increment is requested. One of the push buttons will be used as a clock input for the system and when pressed it will cause the new selected user value to be added to the existing counter of the counter. In the event, the value of the counter exceeds the 4 bits of the counter, the value should wrap around through 0. The value of the counter is always displayed via a digit of the 7-segment display. There also needs to be a synchronous clear input that will set the count to 0 on the next clock cycle when pressed.

A block diagram of the system, also known as a black-box diagram, is shown below in Figure 4.1 with an interface definition table. Following that, the top level block diagram in Figure 4.2 shows the separate components that make up the system and how they connect with each other. Notice that the interfaces used in both diagrams are well defined with tangible properties. This is important with modular design, as it ensures the subsystems are compatible. It also ensures the system as a whole is modular and can be incorporated into larger designs with appropriate inputs and outputs.

Black Box Diagram

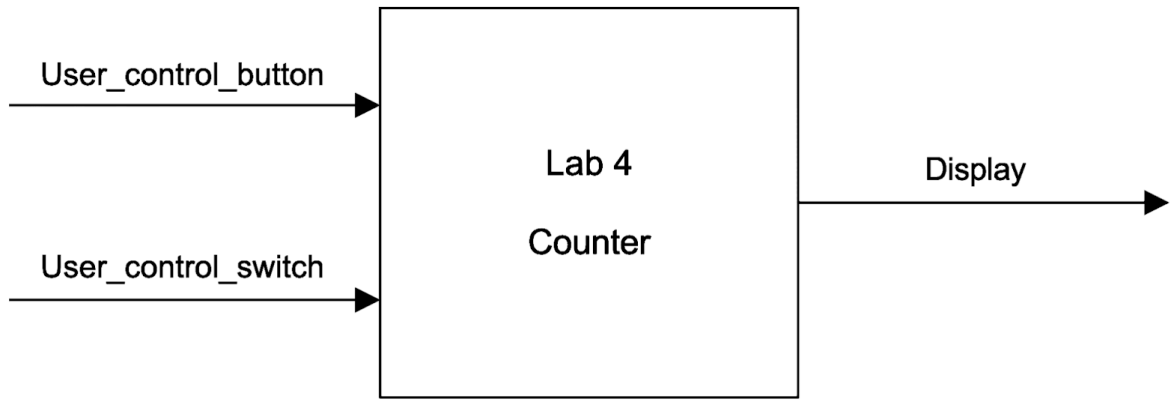
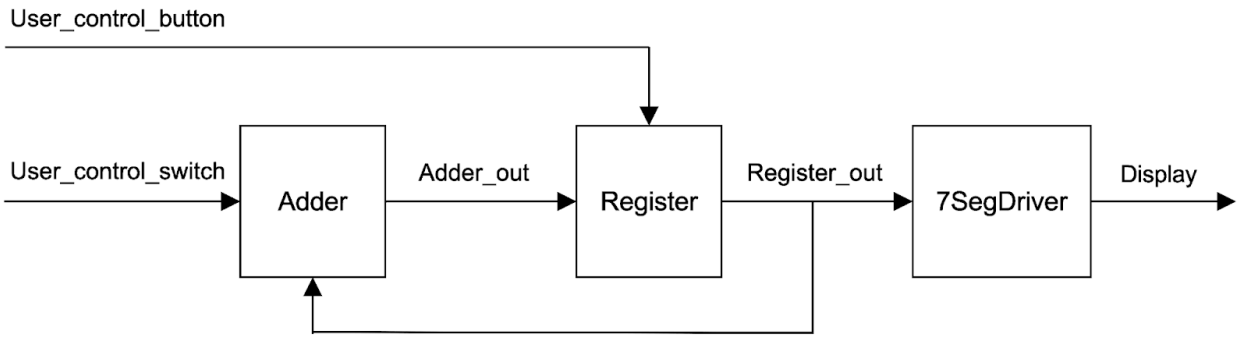


Figure 4.1: Black Box diagram of system

Interface name	Interface Properties
User_control_button (Input)	• 2 push buttons ○ Clear

	○ Clock ● Active LOW
User_control_switch (Input)	● 4 switches ○ 4-bit number ○ 0000 to 1111 ● Active HIGH
Display (Output)	● Single 7-segment display ● Final sum value ● 0 - F

Top Level Block Diagram



Interface name	**Interface Properties**
User_control_button (Input)	● 2 push buttons ○ Clear ○ Clock ● Active LOW
User_control_switch (Input)	● 4 switches ○ 4-bit number ○ 0000 to 1111 ● Active HIGH
Adder_out	● Next value (value to be displayed after the next rising edge of the clock) ● 4-bit number
Register_out	● Current value (value that is being displayed as a system output) ● 4-bit number
Display (Output)	● Single 7-segment display ● Final sum value ● 0 - F

Figure 4.2: Top level block diagram

HARDWARE DIAGRAM (Quartus or other program is used to show the hardware blocks and **wires/busses**):

Notice that the Adder_out and Register_out interfaces on the top level block diagram hold the "next" and "current" values. This is because *the output of a sequential logic circuit is determined by both current and prior input values, while combinational logic relies only on the current input values*. This is the fundamental difference between sequential and combinational logic and why in sequential logic, **timing matters**. In this case, the register is what introduces sequential logic and gives the circuit memory!

You are required to implement this system as several SystemVerilog blocks connected together mimicking the layout from figure 4.2. You are not allowed to make a single System Verilog module that does everything behaviorally.

Project Setup

1. Create a new project in Quartus named `Counter`.
2. Open the `SevenSegmentDecode.sv` file from lab 3b in Quartus.
3. Download the testbench skeleton file CounterTestbench.sv and move it to the lab 4 folder.
4. Add each of the files to the current project by opening them in Quartus and selecting Project→Add Current File to Project. You can double check that these were added successfully by selecting Files in the Project Navigator pane in Quartus.

RegisterFourBit Module

1. Create a new SystemVerilog HDL file `RegisterFourBit.sv`.
2. Declare a module named `RegisterFourBit`.
3. Add the following inputs and outputs to your design:

Name	Direction	Description
`clock`	Input	Load `d` value into register DFFs on rising edge
`clear_n`	Input	Set register value to 0 **when low and `clock` rises**
`d[3..0]`	Input	Value to store in register DFFs on `clock` rising edge
`q[3..0]`	Output	4 bits currently stored in register DFFs

Note: A suffix of "`_n`" (like `clear_n`) is often used to signify that a signal is active low.

Note: The clear in this design is synchronous, meaning that the counter will only be set to 0 on the rising edge of the clock. In contrast, an asynchronous clear (called a reset) would set the counter to 0 when activated, regardless of the clock signal.

4. Adapt your module created in the prelab to meet these specifications.

Counter Module

1. Create a new SystemVerilog HDL file `Counter.sv`.
2. Declare a module named `Counter`.
3. Add 3 inputs and 1 output to the module:

Name	Direction	Description
`clock`	Input	Counter increment signal
`clear_n`	Input	Counter clear signal (active low)
`addBy[3..0]`	Input	Amount to increment by every clock tick (0-15)
`count[3..0]`	Output	Value stored in counter (0-15)

4. Declare a four bit internal value named `count_next` to hold the next value to be stored into the counter. Reference the signals `digitTest` and `segmentsMeasured` in `SevenSegmentTestbench.sv` from lab 3b for a refresher on how to do this.
5. Assign a value to `count_next` using the `+` operator. This will insert an adder into the design. Reference figure 4.2 for which values to add.
6. Create an instance of the `RegisterFourBit` module and connect the signals according to figure 4.2. Reference the `dut` module in `SevenSegmentTestbench.sv` for an example on how to create an instance of a module.
7. Connect the remaining input and output signals according to figure 4.2.

Testbench

Like lab 3b, you will build a testbench from a skeleton file, then simulate the testbench to validate the functionality of your `counter` module. Open the testbench file downloaded at the beginning of the lab and follow the instructions provided.
As you work through the testbench skeleton file, refer back to the following notes. They are not strictly necessary to complete the assignment, but give background information and context that might be helpful.

Note A: Tasks

For those with prior coding experience, it may be tempting to view a SystemVerilog task as a function like in a traditional programming language. This is a good analogy, but has its limitations. Unlike functions in C or C++, no copies of the arguments are made; a task works directly with the original values. This behavior is similar to using pointers or references.

Note B: Advanced `$display` usage

In addition to printing raw text, the `$display` system task can also display values that change over time. This syntax is similar to that of the `printf()` function from C, where a percent sign is followed by a width and format specifier to indicate how to print an argument. For example:

- `%1h`: Prints a number in hexadecimal using 1 character.
- `%7b`: Prints a number in binary using 7 characters.
- `%0t`: Prints a number formatted as time using the minimum number of characters required. Usually used with `$time` to print the current time in simulation.

Read more about the `$display` system task here.

Note C: Simulation cases

It is often impractical to validate every possible simulation case. Usually, it suffices to test a combination of general cases and edge cases that utilize all functionality of a module. In this lab, the counter is tested for `addBy` values of `0x0`, `0x1`, `0x7`, `0x8`, and `0xF` for 5 increments each.

- `0x0` is a special case where no increment occurs.
- `0x1` is a simple case, as well as the most used in counters.
- `0x7` is a mid-range general case.
- `0x8` tests the overflow functionality.
- `0xF` is the maximum value of `addBy` and also tests overflow.
- 5 increments leaves the counter at values that aren't considered "nice."

Design Validation

1. Open ModelSim and create a new project called `Counter`.
2. Add the existing files `Counter.sv`, `CounterTestbench.sv`, and `RegisterFourBit.sv` to the project. Compile the files and fix any compilation errors that may arise.
3. Start simulation and **select the testbench module** `work.CounterTestbench` to simulate.
4. In the transcript, run the command "`add wave *`" to view all signal waveforms.
5. In the waveform diagram, display the `addBy`, `count`, and `expected` signals in hexadecimal by right-clicking the signal name and selecting Radix→Hexadecimal.
6. Execute the command "`run -all`" to simulate the testbench.

7. Monitor the transcript for text output. If no error messages were printed, the module successfully passed validation! Otherwise, use the timestamp and printed signal values to go back and debug issues with the module and/or the testbench.

CounterDisplay Module

Note: The seven segment decoder was validated in lab 3b, and the `countDisplay` module consists of one connection, so we will assume these will work. We skip validation of this module because it is more tedious and adds unnecessary complexity, but it would follow the same principles as the counter testbench.

1. Create a new SystemVerilog HDL file named `CounterDisplay.sv`. Set this as the project's top-level design entity,
2. Declare a module named `CounterDisplay`.
3. Add 3 inputs and 1 output to the schematic:

Name	Direction	Description
`clock`	Input	Button to increment counter
`clear_n`	Input	Button to clear counter
`addBy[3..0]`	Input	4 switches to set increment amount (0-15)
`Seg0[6..0]`	Output	Rightmost seven segment display

4. Add instances of your `Counter` and `SevenSegmentDecode` modules to the module.
5. Connect the `count` output of your counter to the `digit` input of your seven segment decoder. Connect the `clock`, `clear_n`, `addBy`, and `Seg0` signals to module inputs and outputs as appropriate.
6. Save the schematic as `CounterDisplay.sv`.
7. Compile the design.
8. Finally, assign pins to your inputs and outputs in the pin planner.
    a. Assign the `addBy` inputs to switches and the `clock` and `clear_n` inputs to push buttons.
    b. **Please note** that the push buttons have a different I/O standard that the switches and seven segment display. This can be found in the DE10-Lite User Manual.
    c. Instead of assigning the pins of the seven segment display manually, download and import this .qsf file. Download the file to your `ECE272` folder. Then, select Assignments → Import Assignments in Quartus and select the .qsf file. It is important to use the `Seg0[x]` naming scheme for this import to work properly. (0 corresponds to segment A, 1 to segment B, etc.).
9. Save pin assignments and recompile.

Design Verification

Once your Counter module passes the testbench and the display module is created, upload the design to the FPGA and verify that everything works as expected. Remember that the `clear_n` signal is synchronous, so the counter value will only be set to 0 when `clear_n` is low AND the clock has a rising edge.

Verification Deliverables

When checking off your lab, be sure you are set up ahead of requesting the lab assistant review your work. For this checkoff you will be expected to show:

- A SystemVerilog design using a register and an adder to implement a counter.
- `clear_n` signal sets counter to 0 when active and
- Automated testbench in SystemVerilog.
- Valid simulation output.
- **Explanation of simulation results.**
- Valid hardware output.

The full rubric and explanation of the checkoff process can be found on the Lab 4 Checkoff assignment on Canvas.

Extra Credit

- Expand your counter to use all six 7-segment displays on the board. This will require 42 pin assignments for the 7-segment displays alone. You will have to expand your adder and register to handle the larger data. Use this .qsf file as described in the Design Entry section. You must use the naming scheme `Seg0[x]`, `Seg1[x]`, `Seg2[x]`, and so on for the remaining seven segment digits, where `x` is the segment number (0 for segment A, 1 for segment B, …, 6 for segment G). `Seg0` is the rightmost display segment on the FPGA and `Seg5` is the furthest left.
- Add a module between the counter and 7-segment displays to display in decimal on the 7-segment decoders. You must at least process numbers from 0-99.

Postlab

1. What can happen with a sequential circuit like this if it is not initially cleared (think beyond what you observe on the FPGA)?

2. Create a black box diagram of your design from lab 2 or lab 3. Be sure to label the system, inputs, and outputs.

3. Create an interface definition table for your black-box diagram. Provide at least two properties for each interface.