The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 541 Digital Logic and Computer Design
Prof. Montek Singh
Spring 2020

Lab #6: A Stop Watch
Issued Fri 2/14/20; Due 1:25pm on Fri 2/21/20 (beginning of lab)

You will learn the following in this lab:

- Debouncing switches
- Developing an up-down stop watch
- Writing combination logic using procedural (`always_comb`) statements

---

Part 0: See switch bounce

Let us see the effect of switch bounce. We want to play with one of the slide switches (say, the rightmost slider), and see how many times its output bounces each time you flip the switch. Our strategy is to re-use the design from Part 1 of Lab 5 (`counter8digit`), but instead of having the counter increment on the positive edge of clock, have the counter increment on each positive edge of the switch input. Therefore, each time the switch bounces (i.e., goes down and then up again), the counter will count this as one bounce. *Note:* Only the counter will be fed the slide switch input as a "clock"; the rest of the design, esp. the display, must continue to be fed the real clock.

The following files for this part are available on the website:

- `seebounce.sv`: Observe how the counter (called `numBounces`) counts the positive edge transitions of X, which is the input coming from the rightmost slider switch.
- `seebounce.xdc`: The constraints file for this exercise. The rightmost slider switch is attached to the X input to the top-level module.

Run the tools and program the board. Move the slide switch up and down, and see what happens. It is quite possible that sometimes you will see no bounce; these switches have been designed to minimize bounce! There is analog circuitry on the board that does a fairly good job of debouncing. But, some of you *may* see switch bounce if you slide the switch a bit slowly. (If you don't see any bounce, that's okay; proceed with the next step anyway.)

Part 1: Fix switch bounce

Let us now fix switch bounce. We will insert a new module called `debouncer` between the raw (i.e., bouncy) input and where it is desired to be used. We will use the strategy we developed in class:

> *When the raw input to the debouncer changes its value from the current value of the debouncer output, wait for a certain amount of time (typically, several milliseconds) and verify that the input has been steady throughout this time before changing the debouncer output.*

Use the Verilog source provided on the website for our debouncer. Do the following:

- Some lines of code are hidden. Fill in the blanks. Name this file `debouncer.sv`.

- The parameter *N* determines the duration of debouncing. In particular, the debouncer will check that the input is steady for 2^N consecutive clock cycles before reacting to it. If you set *N* too small (say, *N*=2, i.e. 4 clock cycles), it may not filter out all the bounces. If you set *N* too large (try *N*=26), the debouncing takes too long.
- Use the tester provided (`tester_debounce.sv`) to test your debouncer. The tester sets *N*=4, which is too small a value, but suitable to test the operation of the debouncer via simulation. If your simulation results are similar to the screenshot provided on the website, then your debouncer is working correctly, and you may proceed to the next step.
- Instantiate a debouncer module within `seebounce.sv`, to clean up any bounces on the input *X*. In particular, pass the glitchy *X* input into the debouncer, and use the clean output of the debouncer, say `cleanX`, as the clock to increment `numBounces` (i.e., use `posedge cleanX` instead of using `posedge X`). Calculate the value of *N* that makes the debouncer duration approximately one-hundredth of a second (10 ms). Use this value for the rest of the lab. Now, when you flip the switch on and off repeatedly, you should certainly see no bounces anymore.

Question 1: What value did you calculate for *N* for a debounce duration of 10 ms?

Part 2: Make an 8-digit hex stop watch with up, down, start and stop push buttons

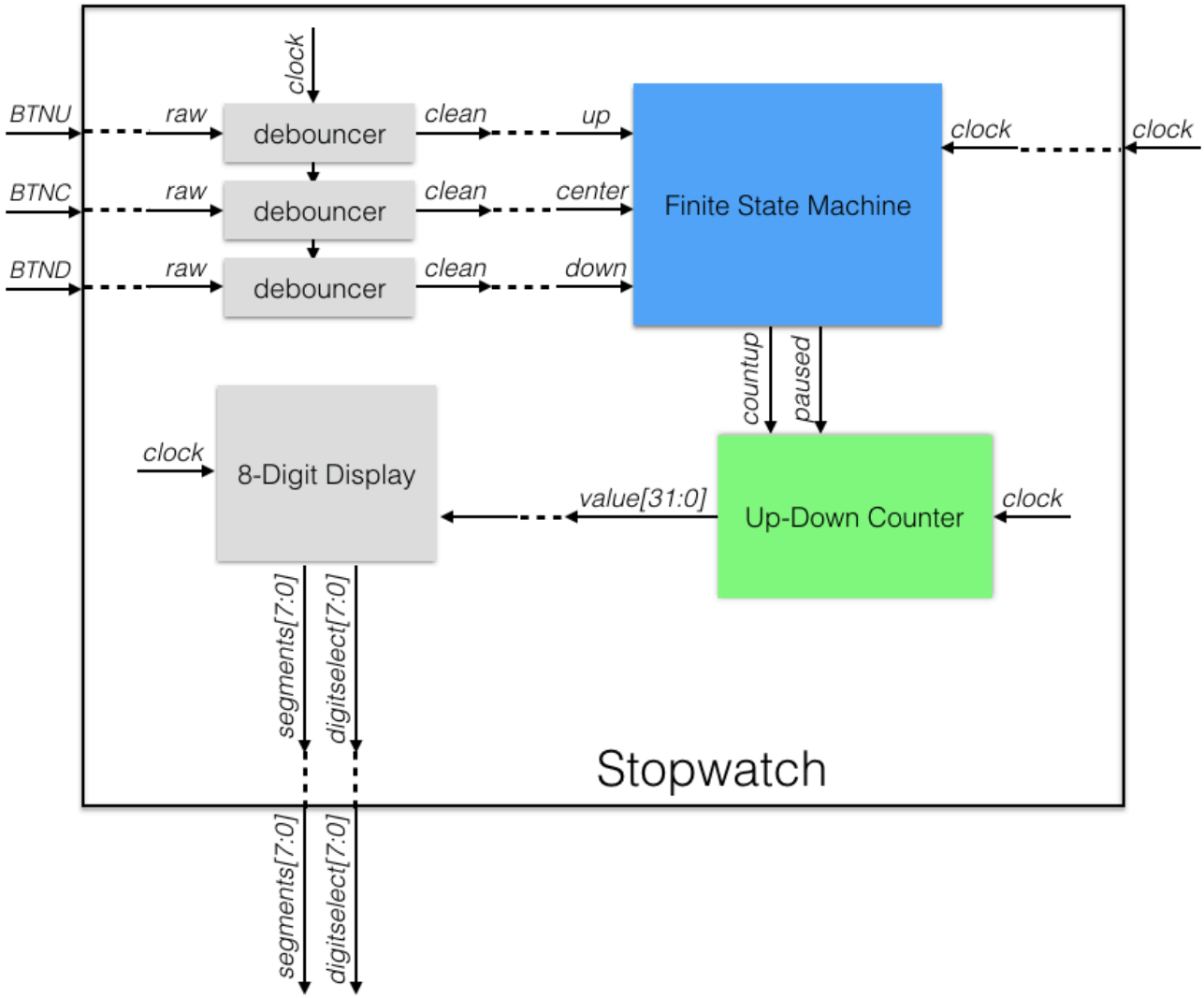
You final task is to take your 8-digit counter design (`counter8digit` from Part 1 of Lab 5), and add the following capabilities: (i) ability to count up or down; (i) ability to stop and start (i.e., pause and resume). Your design should meet the following requirements:

- Use the push-buttons instead of slide switches: *BTNU* for counting up, *BTND* for counting down, *BTNC* (the center one) for pause as well as resume (i.e., toggle between them).
- Use three instances of the `debouncer` module to debounce each of these three push-button inputs.
- The counter starts out at the value 0, and counting up. If the counter is counting up and reaches "FFFFFFFF", it simply rolls over to "00000000" and continues counting. Similarly, if the counter is counting down and reaches "00000000", it wraps around to "FFFFFFFF" and keeps counting. *TIP:* None of this actually needs any special circuitry; this is how a simple counter already wraps around!
- As in Part 1, the least-significant digit of the counter should count at a rate of ***approximately*** *256 times a second*.

In more detail, the desired behavior is as follows:

- The counter starts out at the value 0, and counting up.
- If at any time, the down button *BTND* is pressed, the counter switches its counting mode to counting down. Similarly, if at any time, the up button *BTNU* is pressed, the counter switches to counting up. (Pressing the up button while the counter is already counting up has no effect; similarly for the down button.)
- If the counter is counting (up or down), and the center button *BTNC* is pressed, the counter pauses, i.e., stops counting and holds its current value. Subsequently, the center button must be pressed again for counting to resume. The relevant action (pausing or resuming) should happen upon button press, though it will be necessary to detect a button release before a button press is recognized again.
- When counting is paused, the direction of counting (up or down) is remembered so that upon resumption, counting proceeds in the correct direction. However, during the pause, the direction of counting can be changed by pressing the *BTNU* or *BTND* buttons.
- In summary, the center button toggles between counting and pausing, while the up and down buttons change the direction of the counting.

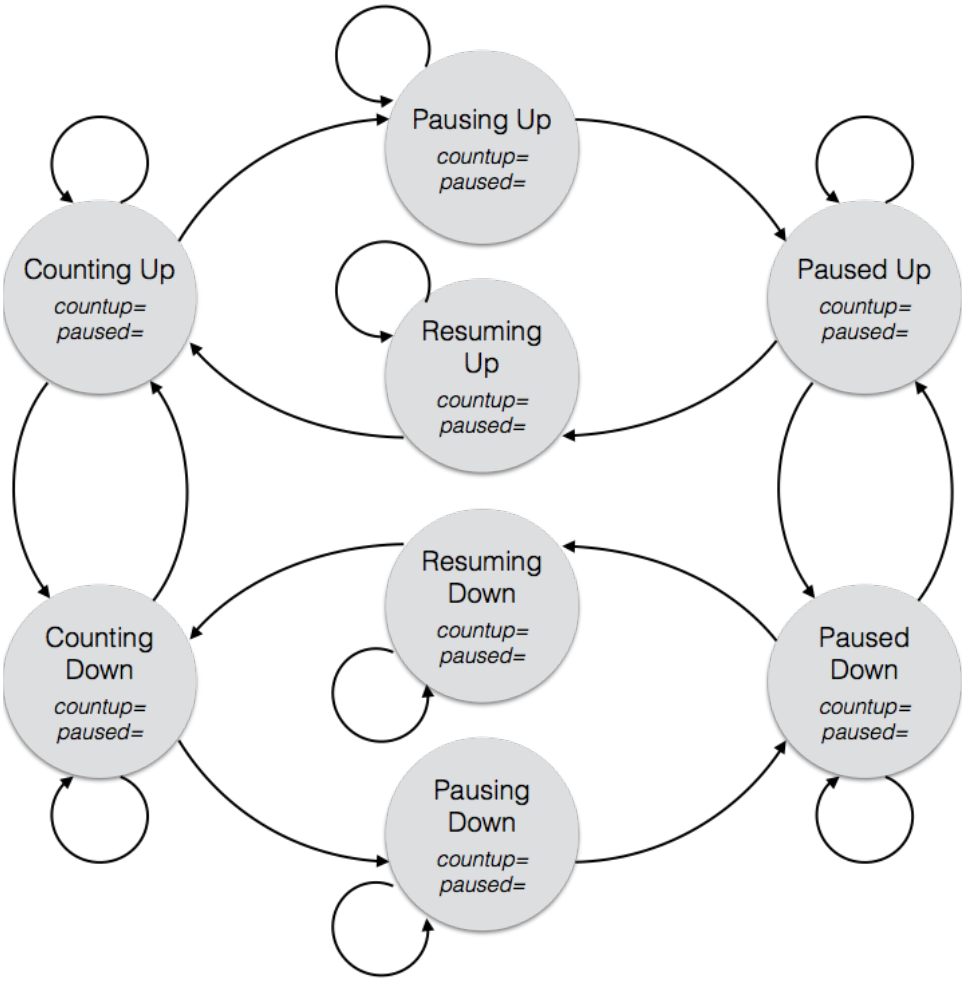
You will need a finite-state machine (FSM) to keep track of the various states of the system. Below is a block diagram that shows the hierarchy of this design. *Be sure that your Verilog description exactly matches the block structure shown in the diagram.*



Be sure that your project contains the following modules in their respective Verilog source files:

- 8-Digit Display: **`display8digit.sv`**
- Debouncer: **`debouncer.sv`**
- Finite State Machine: **`fsm.sv`**
- Up-Down BCD Counter: **`updowncounter.sv`**
- Top-Level Stopwatch (everything else is inside it): **`stopwatch.sv`**

To help you implement the FSM, a state transition diagram is shown below. You will need to fill in the details as discussed in class/lab.



You will need to label the inputs on the arcs and the outputs inside the circles (Moore-style state machine). Follow the usual steps taught in class to convert this state diagram into a Verilog description of an FSM (use the template provided in **`lab6_fsm_template.sv`**).

What to submit:

- **The SystemVerilog sources: debouncer.sv, fsm.sv, updowncounter.sv, and stopwatch.sv**
- **A picture (phone picture is fine) or scanned version of the completed state diagram above.**
- **Your answers to Question 1.**
- **Show a working demo of your stopwatch during the lab session on Fri Feb 21.**

How to submit: **Please submit your work by email by 1:25pm on Fri Feb 21 as follows:**

- **Send email to: `comp541-submit-s20@cs.unc.edu`**
- **Use subject line: Lab 6**
- **Include the Verilog sources and the picture of FSM as specified above**
- **Include your answer to Questions 1 within the email body**
