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

Comp 541 Digital Logic and Computer Design
Fall 2014

Lab #6: A Stop Watch
Issued Wed 9/24/14; Due Wed 10/1/14 (submit by 11:59pm)

You will learn the following in this lab:

- Driving a multi-digit 7-segment display
- Writing combination logic using procedural (always@) statements
- Working with binary-coded decimals (or BCD)
- Debouncing switches
- Developing an up-down stop watch

Part 0: Handling a 4-digit seven-segment display

Create a new project, `Lab6`. Copy your `hexto7seg.v` from Lab 5 Part III to the new project (use "Add Copy of Source"). Do the following:

- Download the Verilog file `display4digit.v` from the website, and add it to the project
  - This file contains a module called `display4digit()`. This is a higher-level module that drives all four digits of the display in rapid succession, each time generating the appropriate segment values using a *single hex encoder* (i.e., single instance of `hexto7seg`).
  - Pay careful attention to the following: How each of the 4 digits of the display are selected in a round-robin fashion; how different hex values are selected for generating the output, again in a round-robin fashion.
- Feed 16 bits (instead of 4 bits) into this display, from the `counter1second` module from Lab 5.

Question 1: Calculate the refresh frequency for each digit of the display (i.e., how many times per second is one particular digit of the 4-digit display generated per second)?

You are encouraged to play with the refresh frequency by changing which bits of the counter are used to perform the round-robin selection. For example, using higher numbered bits of the counter (`counter[23:22]` instead of `counter[18:17]`) will slow the display refresh down enough for you to see the round-robin selection. You may try other slower/faster values also.

Part I: Rewrite display encoder in procedural style

Rewrite the Verilog specification in `display4digit.v` and `hexto7seg.v` so that the following three continuous assignment statements are replaced by the **`always @(*)/case/default`** style discussed in class as an alternative style for describing combinational logic:

- `assign digitselect`
- `assign value4bit`
- `assign segments`

Question 2: Does your rewritten code compile and run and produce exactly the same behavior as in Part 0 (it should!)?

Part II: See switch bounce

Let us now 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 Lab 5 Part III, but instead of having the counter increment at positive edge of clock, have the counter increment at 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 file for this part is available on the website:

- `seebounce.v`: Observe how the counter (called numBounces) counts the positive edge transitions of A, which is the input coming from the rightmost slider switch.

Write an appropriate UCF file to provide the correspondence between the top-level I/O of the design and connections on your board. 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, you will definitely see switch bounce if you slide the switch a bit slowly.

Part III: 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. Note the following:

- Some lines of code are hidden. Fill in the blanks. Name this file `debounce.v`.
- 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. Try setting *N* to 26, and then manually flip the switch of-and-on rapidly. See what happens.
- Calculate the value of *N* that makes the debouncer duration approximately one-hundredth of a second (10 ms).

Question 3: What value did you calculate for *N* for a debounce duration of 10 ms?

Part IV: Make a 4-digit hex stop watch with up, down, start and stop push buttons

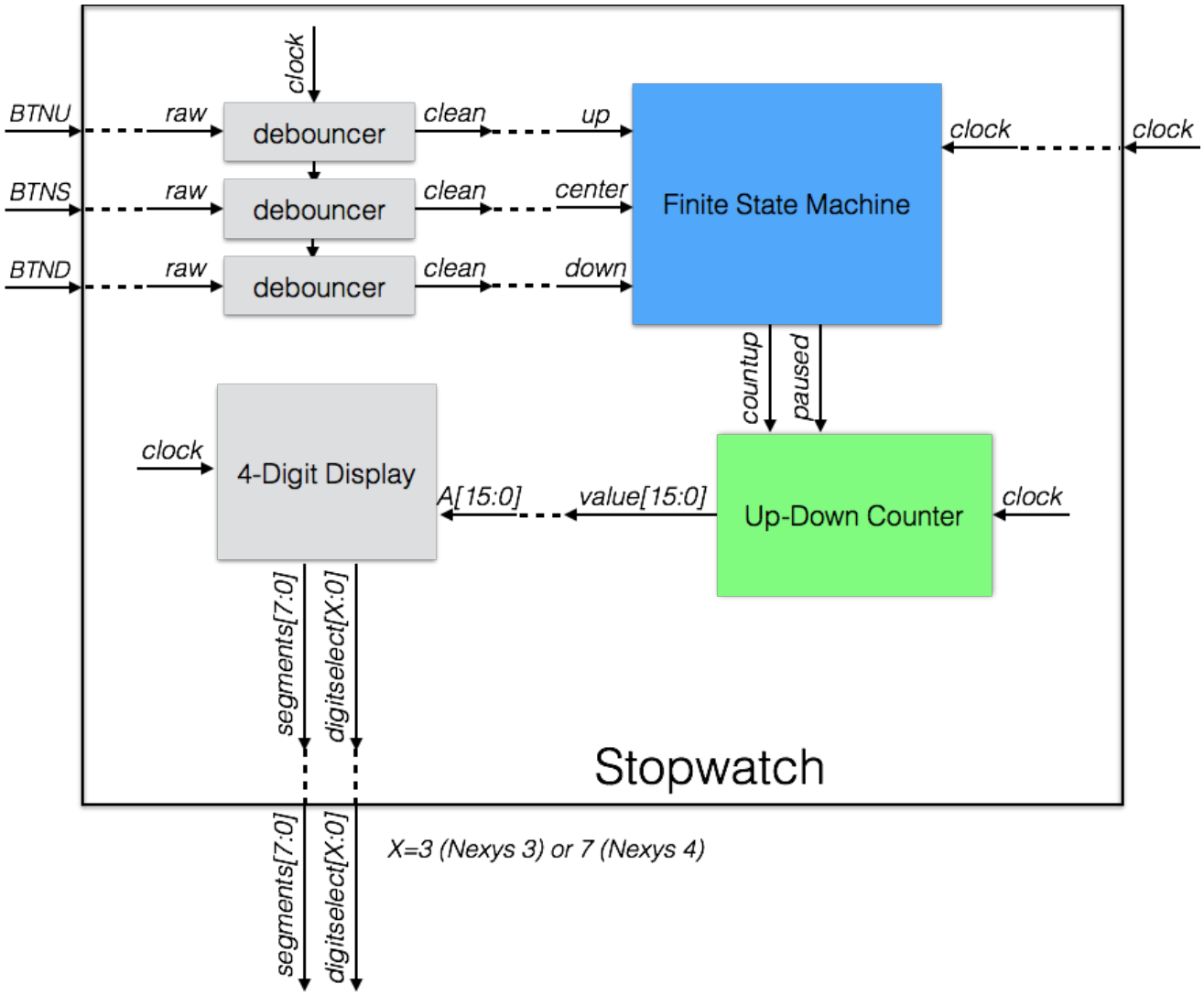
You final task is to take your counter design from Part 1, and add the following capabilities: (i) ability to count up or down; (i) ability to stop and start (i.e., pause and resume). Your design should meet the following requirements:

- Use the push-buttons under the display, instead of slide switches: *BTNU* for counting up, *BTND* for counting down, *BTNS* (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 "0000", and counting up.
- There is no need to reset the counter while it is running. Also, if the counter is counting up and reaches "FFFF", it simply rolls over to "000" and continues counting. Similarly, if the counter is counting down and reaches "0000", it wraps around to "FFFF" and keeps counting. *TIP:* None of this actually needs any special circuitry.
- As in Lab 5, the counter should count at a rate close to one "tick" per second.

In more detail, the desired behavior is as follows:

- The counter starts out at the value "0000", 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 *BTNS* 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.

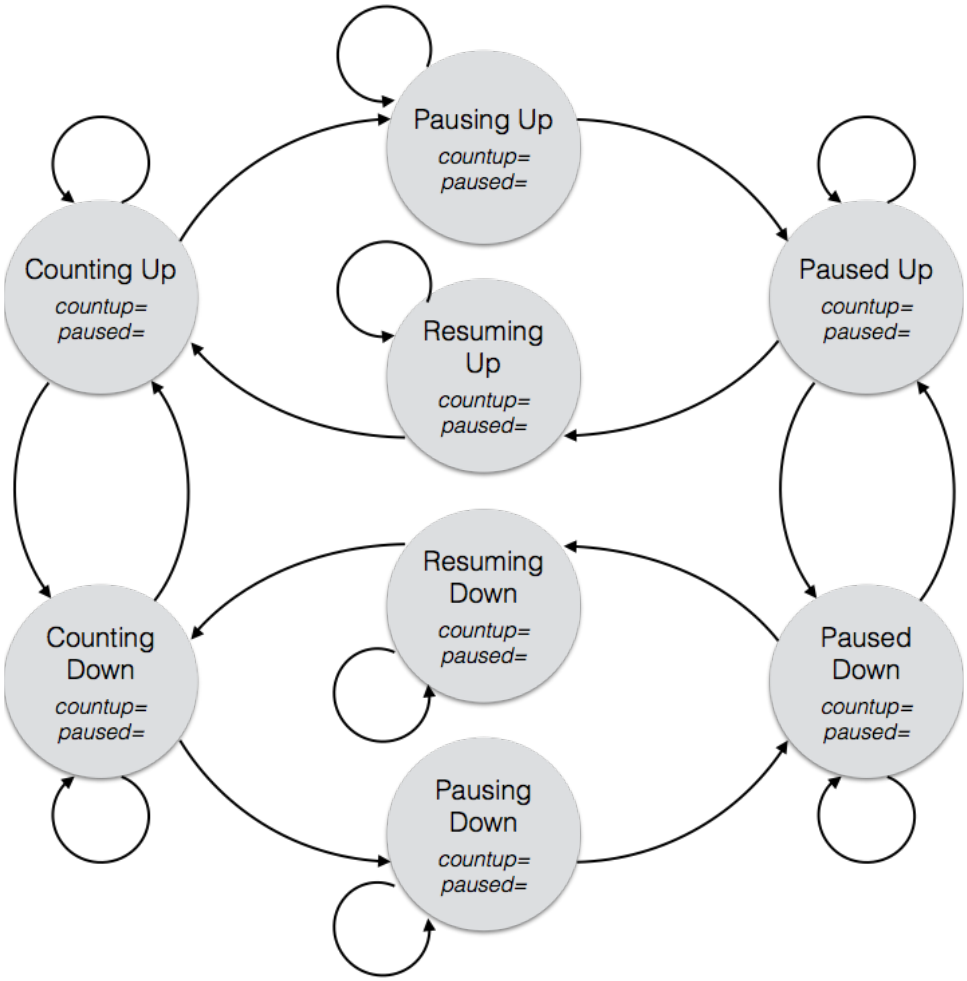
You will need a finite-state machine (FSM) to keep track of the various states of the system. Below is a block diagram that shows the hierarchy of this design. *Be sure that your Verilog description exactly matches the block structure shown in the diagram.*



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

- 4-Digit Display: **`display4digit.v`**
- Debouncer: **`debouncer.v`**
- Finite State Machine: **`fsm.v`**
- Up-Down Counter: **`counter.v`**
- Top-Level Stopwatch (everything else is inside it): **`stopwatch.v`**

To help you implement the FSM, a *partial* 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 **`fsm_3blocktemplate.v`**).

What to submit:

- **The five Verilog sources: display4digit.v, debouncer.v, fsm.v, counter.v, and stopwatch.v**
- **A picture (phone picture is fine) or scanned version of the completed state diagram above.**
- **Your answers to Questions 1-3.**
- **Show a working demo of your design for Part IV during the lab session on Oct 3, 2014.**

How to submit: **Please submit your work by email by 11:59 pm on Wed Oct 1 as follows:**

- **Send email to: `comp541submit-cs@cs.unc.edu`**
- **Use subject line: Lab 6**
- **Include the five Verilog sources and the picture as specified above**
- **Include your answers to Questions 1-3 within the email body**
