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

Comp 541 Digital Logic and Computer Design
Fall 2014

Lab #4: Sequential Design: Counters

Issued Wed 9/10/14; Due Wed 9/17/14 (11:59pm)

This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided, including screenshots of many of the steps. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission instructions are at the end.

You will learn the following:

- Specifying sequential circuits in Verilog
- Designing different types of counters
- Including synchronous reset capabilities
- Including start/stop functionality in your counters
- Verilog simulation, and test fixtures with clocks

Part 0: Reading

Since we have not yet covered sequential logic in the lectures, here is a list of the relevant sections of the textbook that you must read carefully before proceeding with this lab assignment:

Sections 3.2.3 – 3.2.6: Flip-Flops and Registers

Sections 4.4.1 – 4.4.3: Verilog for Flip-Flops and Registers

Section 5.4.1: Counters

Part I: A Mod-4 Counter

Let us begin by designing a modulo-4 counter, i.e., one that counts in the following sequence: 0, 1, 2, 3, 0, 1, 2, 3, 0, … . The counter module will need the following: a 2-bit register to store the current value, a clock input to pace the counting, and a *Reset* input to synchronously reset the counter to 0. All changes to the counter's value—whether counting up or resetting—take place at the upward clock transition (i.e., *positive edge* of clock).

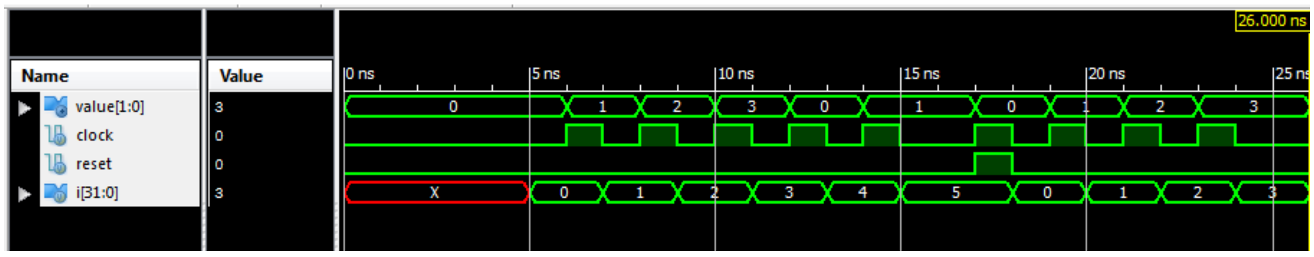
Use the following Verilog specification for the counter:

```
module CounterMod4(clock, reset, value);
    input clock;
    input reset;
    output reg[1:0] value = 0;

      always @(posedge clock) begin
        value <= reset ? 0 : (value + 1);
      end

endmodule
```

Note the following:

- The signal "value" is declared as a "reg" type, not a "wire" type. A "reg" type typically allocates a register, i.e., one flip-flop per bit. (There is a special case that will be discussed in later labs where the "reg" type does not allocate a register.) In this example, a 2-bit register is allocated for "value". Thus, "value" is a sequential type of logic: it has memory and can remember its value between changes.
- The "always" statement is a new type of Verilog behavioral construct. In this example, it states that whenever there is a positive edge of clock [`always @(posedge clock)`], the "value" is updated to either "value+1" (if counting) or to "0" (if resetting).

To test, use the Verilog test fixture provided on the website (CounterMod4_test.v). This test fixture does the following: waits 5 ns; starts the clock (positive edge at 6 ns, period of 2 ns); simulates the counter for 5 clock cycles; then asserts the reset signal to reset the counter back to 0; then runs the counter for another 3 clock cycles. *Be sure to go through the test fixture line-by-line and understand what each line does!* If you format all the waveforms to display in decimal, you should see exactly the following:



Part II: A Mod-7 Counter

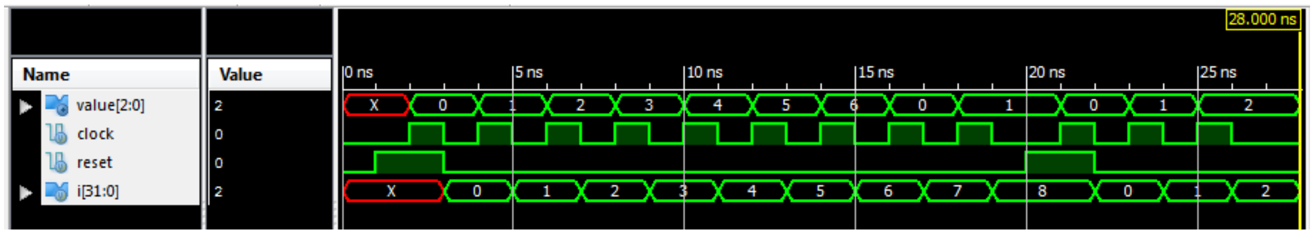
Your task is to design a mod-7 counter (with a synchronous reset), and test it via simulation. (Use the mod-4 counter from Part I as a starting point, and make appropriate modifications.) In particular, the mod-7 counter will be different in two respects:

- It needs a 3-bit register for *value,* instead of the 2-bit register used in Part I.
- It counts in the sequence 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, … . Note: This sequence length is not a power of 2, so you cannot rely on the counter wrapping around to 0 on its own after reaching 6!

Use the Verilog code given below to implement this counter, and appropriately fill in the details that have been blacked out.

```
module CounterMod7(
    input clock,
    input reset,
    output reg [2:0] value  // Observe how this line is different from Part I
    );

      always @(posedge clock) begin
        value <= reset ? 0 : [redacted] ?[redacted]: [redacted]);
      end

endmodule
```

Simulate the new counter using the Verilog test fixture provided on the website (CounterMod7_test.v). *Be sure you understand every line of the test fixture file!* Set the display format to *Unsigned Decimal* for all the outputs. Your simulation output should look exactly like this:



Question 1: Why is the *value* waveform shown as "X" for the first two nanoseconds of this simulation? Why does it become "0" at 2 ns? In contrast, for Part I, the *value* waveform starts out as "0"; why?

Part III: A Mod-7 Counter with a Stop signal

Copy your mod-7 counter from Part II to a new file named *CounterMod7Reset.v*. Your task is to modify the mod-7 counter to incorporate a *Stop* signal, which inhibits the counter from counting up at the next (one or many) upward clock transitions. In particular, if *Stop* equals “1”, then at the next positive edge of the clock, the counter’s *value* does not change. This will be true as long as *Stop* is high, providing you a means to “freeze” the counter for as long as you want. Subsequently, when *Stop* is changed back to “0”, the counter starts counting again, from where it left off.

NOTES:

- If *Stop* and *reset* are both “1”, the counter should reset. That is, *reset* has higher priority than *Stop.*
- The assignment to *value* should still be done using a single statement `(value <= …)`, although you are allowed to use nested conditionals. You can also split the statement up onto multiple lines for clarity, but it should remain a single statement.

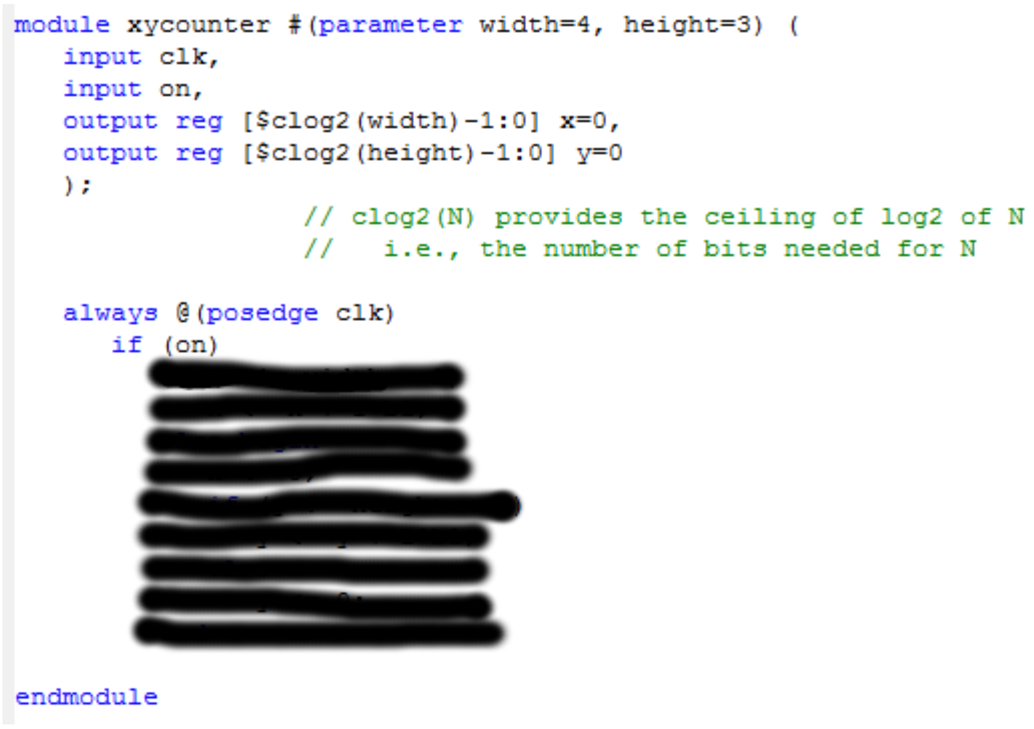
Use the Verilog test fixture provided on the website (CounterMod7Reset_test.v).

Part IV: Designing an *xy-counter.*

Design a two-dimensional counter (i.e., *xy-counter*). This counter steps through a 2D matrix, one row at a time. The matrix range is [*0..width-1*, *0..height-1*].

- The counter starts at (*x, y*) = (0, 0) and increments *x* to go from (0, 0) to (*width-1,* 0).
- Then it wraps around to the beginning of the next line, (0, 1).
- Similarly, the counter wraps around from the end of the bottom line, (*width-1, height-1*) back to the top, (0, 0).
- The counter also has an input called *on* that tells the counter whether counting is enabled or disabled. Thus, if *on* == 0, the counter does not increment on the next positive edge of clock.
- Name this module `xycounter`, and name the source file `xycounter.v`.

A Verilog template for `xycounter` is provided below:



```
module xycounter #(parameter width=4, height=3) (
   input clk,
   input on,
   output reg [$clog2(width)-1:0] x=0,
   output reg [$clog2(height)-1:0] y=0
   );
                  // clog2(N) provides the ceiling of log2 of N
                  //   i.e., the number of bits needed for N

   always @(posedge clk)
      if (on)
        [redacted]

endmodule
```

A Verilog test fixture is provided on the website (`xycounter_test.v`). Fill in the details, simulate using the text fixture, and verify that the counter behaves as expected. You may try a couple of different sets of values of *width* and *height* (and accordingly, the number of bits needed for *x* and *y*). But, you need only submit the results for the test fixture provided.

What to submit:

- **A screenshot of the simulator window clearly showing the final simulation result for PART II (use filename waveforms2.png, or other appropriate extension). Do not submit Verilog file.**
- **Your answer to Question 1 in PART II (write it in the body of the email).**
- **Your code for the counter module in PART III (CounterMod7Reset.v).**
- **A screenshot of the simulator window clearly showing the final simulation result for PART III (use filename waveforms3.png).**
- **A screenshot of the simulator window clearly showing the final simulation result for PART IV (use filename waveforms4.png).**

How to submit: **Please submit your work by email as follows:**

- **Send email to: `comp541submit-cs@cs.unc.edu`**
- **Use subject line: Lab 4**
- **Include the four attachments and the text answer as described above**