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

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

Lab #3A: Sequential Design: Counters

Issued Fri 1/24/2020; Due Wed 1/29/2020 (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 description 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 SystemVerilog
- Designing different types of counters
- Including synchronous reset capabilities
- Including start/stop functionality in your counters
- SystemVerilog simulation, and test fixtures with clocks
- A few new SystemVerilog constructs

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

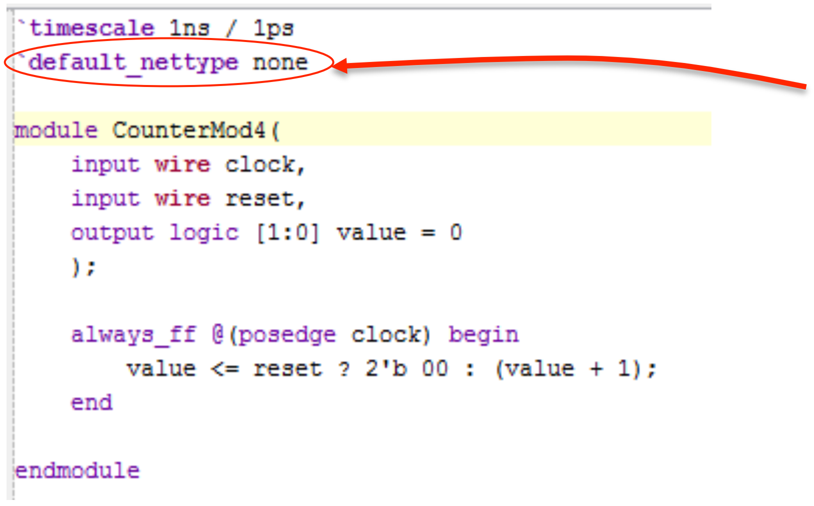
Sections 4.4.1 – 4.4.3: Verilog for Flip-Flops and Registers

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 SystemVerilog specification for the counter:



```
`timescale 1ns / 1ps
`default_nettype none

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

    always_ff @(posedge clock) begin
        value <= reset ? 2'b 00 : (value + 1);
    end

endmodule
```

This line tells the compiler *not* to assume that undeclared names are of the default type `wire`. By suppressing the default type, we are forcing undeclared names to trigger compiler errors. This helps catch undeclared names, typos, etc. *Very useful!* *Use it in every design from now on!*

This specification uses constructs that is not present in standard Verilog, but are part of SystemVerilog: `logic` and `always_ff`. Therefore, when you create a new design source, *be sure to choose SystemVerilog as its file type.* (If you choose Verilog by mistake, you can change the file type by right-clicking on the file under *Design Sources* and choosing *Source Node Properties.* Then, in the pane below (which you might have to pop out), under the *General* tab, change the type to SystemVerilog.) Name the file `CounterMod4.sv`.

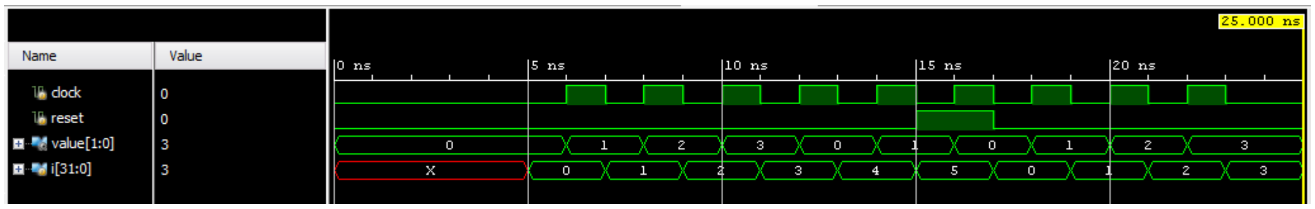
Note the following:

- The first two lines define some defaults. The time units for all delays (e.g., see the tester) are nanoseconds, with a picosecond resolution. And the default data type for any wires etc. not declared will be "none". Without this `default_nettype` declaration, Verilog treats all undeclared identifiers as wires, which can cause trouble with debugging. By explicitly declaring the types of all undeclared identifiers as "none", most such instances will cause compiler errors, forcing us to properly declare them before using them. *Use this directive in every design from now on!*
- The inputs are declared as "`wire`" type (since they are just wires coming in from outside the module), but the output here is of the "`logic`" type, not "`wire`" type. The `logic` type is unique to SystemVerilog, and resolves to different types of structural implementation depending on the complete description. In particular, a `logic` type variable will resolve to a `wire` if its description indicates a combinational function. On the other hand, a `logic` type variable will result in flip-flops being instantiated if its description indicates that state or memory is needed. Thus, the logic type is used to indicate that a variable *might* (but not necessarily) need flip-flops in its implementation.
- The `logic` type used to be called the "`reg`" type in Verilog, but that often got confused with the term *register*. In SystemVerilog, we often simply use the `logic` type in most instances, letting the compiler infer combinational or sequential logic from the rest of the description. But, if you are certain that you are trying to describe a piece of combinational logic, it is better to declare its output as

of the `wire` type, which will force the compiler to indicate an error is the implementation is non-combinational.

- The "`always_ff`" statement is a new type of SystemVerilog behavioral construct. It is used to specify how the value of a *flip-flop* (i.e., register) is to be updated. In this example, it states that whenever there is a positive edge of clock [`always_ff @(posedge clock)`], the "value" is updated to either "value+1" (if counting) or to "0" (if resetting). Thus, since "value" is updated inside an "`always_ff`" construct, it means "value" will be implemented as sequential logic using flipflops, and not as combinational logic, because the "value" is only updated at discrete clock ticks and held steady in-between.
- The assignment operator here specifies what is called a non-blocking assignment, represented by the symbol "<=". This is not to be confused with *less-than-or-equal!* Instead, regard it as a left arrow.
- Since *value* is declared to be 2-bit, when incrementing the value 3, it wraps around to 0.

To test, use the Verilog test fixture provided on the website (`CounterMod4_test.sv`). 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:

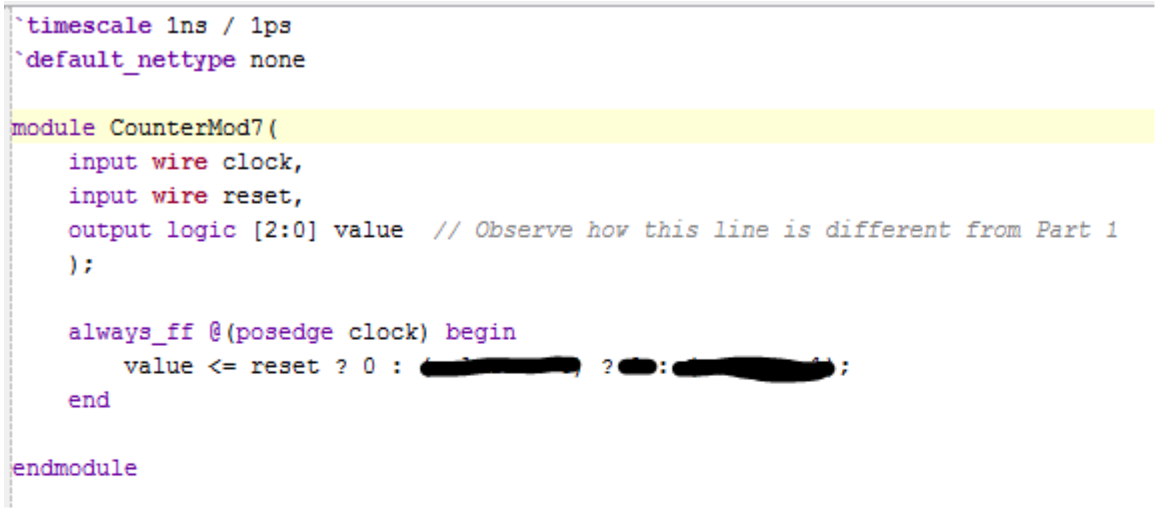


Part II: A Mod-7 Counter

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, copy it to file `CounterMod7.sv`, and make appropriate modifications. In particular, the mod-7 counter will be different in two respects:

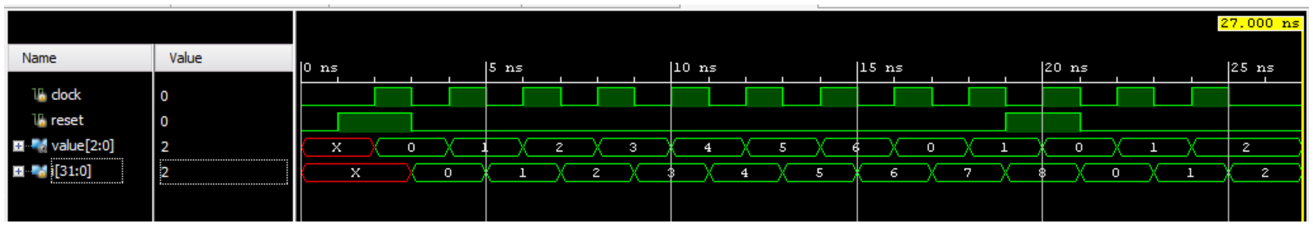
- It needs a 3-bit register for *value,* instead of the 2-bit register used in Part I.
- It counts in the sequence 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, … . Note: This sequence length is not a power of 2, so you cannot rely on the counter wrapping around to 0 on its own after reaching 6!

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



```
`timescale 1ns / 1ps
`default_nettype none

module CounterMod7(
    input wire clock,
    input wire reset,
    output logic [2:0] value  // Observe how this line is different from Part 1
    );

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

endmodule
```

Simulate the new counter using the SystemVerilog test fixture provided on the website (`CounterMod7_test.sv`). *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?

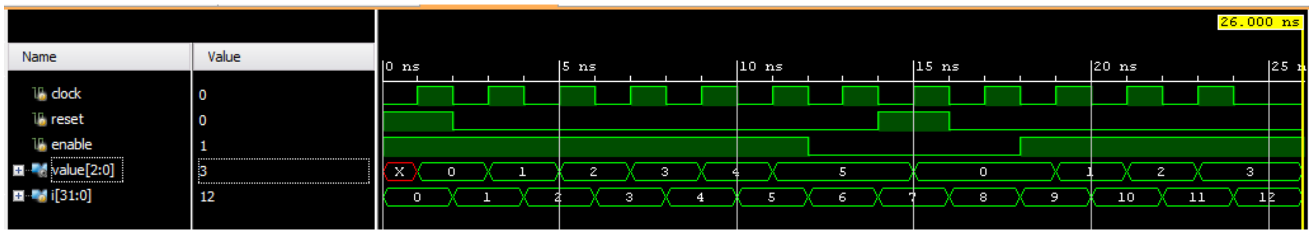
Part III: A Mod-7 Counter with an "enable" signal

Copy your mod-7 counter definition from Part II to a new file named `CounterMod7Enable.sv`. Your task is to modify the mod-7 counter to incorporate an *enable* signal, which inhibits the counter from counting up at the next (one or many) upward clock transitions. In particular, if *enable* equals "0", then at the next positive edge of the clock, the counter's *value* does not change. This will be true as long as *enable* is low, providing you a means to "freeze" the counter for as long as you want. Subsequently, when *enable* is changed back to "1", the counter starts counting again, from where it left off.

NOTES:

- If *enable* is "0" and *reset* is "1", the counter should reset. That is, *reset* has higher priority than disabling.
- 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 onto multiple lines for clarity, but it should remain a single statement.

Use the Verilog test fixture provided on the website (`CounterMod7Enable_test.sv`). Once again, look over the test fixture carefully, and be sure you understand every line. Your simulation output should look exactly like this:

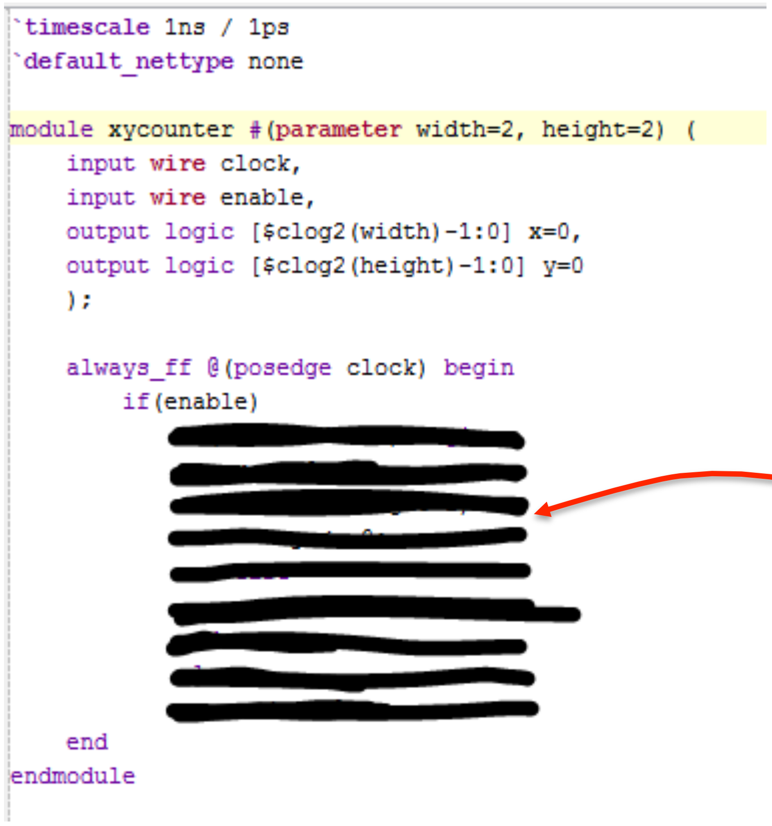


Part IV: Designing an *xy-counter*.

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 *enable* that tells the counter whether counting is enabled or disabled. Thus, if *enable* is 0, the counter does not increment on the next positive edge of clock.
- Name this module `xycounter`, and name the source file `xycounter.sv`.

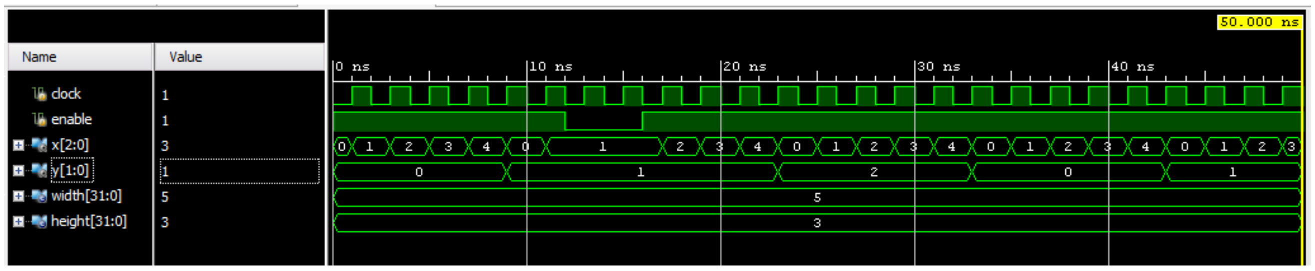
A Verilog template for `xycounter` is provided below:



```
`timescale 1ns / 1ps
`default_nettype none

module xycounter #(parameter width=2, height=2) (
    input wire clock,
    input wire enable,
    output logic [$clog2(width)-1:0] x=0,
    output logic [$clog2(height)-1:0] y=0
    );

    always_ff @(posedge clock) begin
        if(enable)
            [redacted]

    end
endmodule
```

The number of lines in the body could be very different than suggested here, esp. depending on whether or not you use *begin-end* pairs for clarity where they are not really necessary, and whether you use *if-else* blocks for assignment or conditional expressions (?:).

A Verilog test fixture is provided on the website (`xycounter_test.sv`). Fill in the details above, simulate using the text fixture, and verify that the counter behaves exactly as shown below. You may try a couple of different sets of values of *width* and *height*. But, you need only submit the results for the test fixture as provided.



What to submit:

- **A screenshot of the simulator window clearly showing the <u>final simulation result for PART II</u> (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 Verilog for the counter module in PART III (CounterMod7Enable.sv), and a screenshot of the simulator window clearly showing the <u>final simulation result for PART III</u> (use filename waveforms3.png).**
- **Your Verilog for the xy-counter module in PART IV (xycounter.sv), and a screenshot of the simulator window clearly showing the <u>final simulation result for PART IV</u> (use filename waveforms4.png).**

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

- **Send email to: `comp541-submit-s20@cs.unc.edu`**
- **Use subject line: Lab 3A**
- **Include the five attachments and the text answer as described above**

CONTINUE ON TO LAB 3B (on course website)