ASIC Design Laboratory
Lab 4 : Simple Sequential Logic Design and Verification
Lab Manual

Fall 2020

The purpose of this lab exercises is to help you become familiar with designing, implementing, and verifying sequential logic modules.

It is important to note, that given the fundamentally parallel and interactive nature of hardware designs, debugging designs described with HDL code requires a method that strictly identifies and leverages guaranteed cause-effect relationships with in the design's description. Other lazy or speculative debugging methods will generally result in vast amounts of wasted time, effort, and frustration and can easily increase debugging times by a factor of 10x.

In this lab, you will perform the following tasks:

- Create and Test Verilog code for two Synchronizer designs using QuestaSim®
- Create and Test Verilog code for a flexible counter design
- Work with and implement test benches that use tasks to cleanly manage the sequential verification work

1 Lab Setup

In a UNIX terminal window, issue the following commands, to setup your Lab 4 workspace:

mkdir –p ~/ece337/Lab4
cd ~/ece337/Lab4
dirset
setup4

The setup4 command is an alias to a script file that will check your Lab 4 directory structure and give you file needed for starting the lab. If you have trouble with this step please ask for assistance from your TA.

IMPORTANT: Make sure to add this new workspace into your 337 Repository, like you did in Lab1.

This way, you will always have the original copy in storage.

2 Input Synchronizer Implementation

2.1 Verilog Syntax for Describing Flip-Flops

Flip-Flops are the basic synchronous storage cell for CMOS designs and form the basis for all 'registers'. The most common is the D-Flip-Flop with Set and Reset signals (DFFSR) that allow the design to be cleared/reset/initialized to a known operating state. The syntax for describing a Flip-Flop is rather simple and straight-forward but is different than the combinational logic that you have been working with primarily so far. The most common Flip-Flip used in designs and the primary one used for any designs in this course is a rising-edge sensitive Flip-Flop with an active low reset, which has the following syntax:

```
always_ff @ (posedge clk, negedge n_rst)
begin [: <block tag name>]
  if(1'b0 == n_rst) begin
    <Flip-Flop Signal Name> <= <reset value>;
  end
  else begin
    <Flip-Flop Signal Name> <= <Flip-Flop input signal>;
  end
end
```

This syntax implements a Flip-Flop due to the nature of how the sensitivity list is used and the "always_ff" tells the compiler that you intend for it to be flip-flop so it should give error messages if the code is not correct for a flip-flop. One should always use the "always_comb" block instead of the "always" block for combination logic for similar reasons as the "always_ff" block for flip-flops/registers.

2.2 Reset to Logic Low Synchronizer

Design (code and verify) the synchronizer module from lab 1's post-lab with the following specifications:

Required Module Name: sync_low

Required Filename: sync_low.sv

Required Ports:

Port name	Direction	Description
clk	input	The system clock. (Maximum Operating Frequency: 1GHz)
n_rst	input	This is an asynchronous, active-low system reset. When this line is asserted (logic '0'), all registers/flip-flops in the device must reset to their initial value.
async_in	input	This is the original signal which is not synchronized to the supplied clock signal.
sync_out	output	This is the form of the input that is now synchronized with the supplied clock signal.

Figure 1 illustrates expected behavior of this reset-to-low synchronizer.

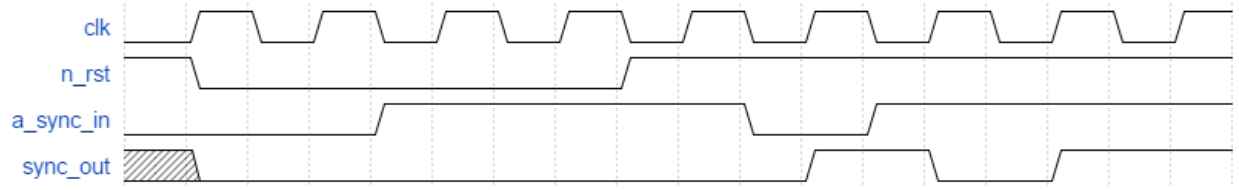

Figure 1: Timing waveform for 2-stage synchronizer for an active high input

2.3 Reset to Logic High Synchronizer

Design (code and verify) a simple modified version of the synchronizer module with the following specifications:

Required Module Name: sync_high

Required Filename: sync_high.sv

Required Ports:

Port name	Direction	Description
clk	input	The system clock. (Maximum Operating Frequency: 1Ghz)
n_rst	input	This is an asynchronous, active-low system reset. When this line is asserted (logic '0'), all registers/flip-flops in the device must reset to their initial value.
async_in	input	This is the asynchronous input port (the original signal which is not synchronized to the supplied clock signal).
sync_out	output	This is the synchronous output port (the form of the input that is now synchronized with the supplied clock signal).

Figure 2 illustrates expected behavior of this reset-to-high synchronizer.

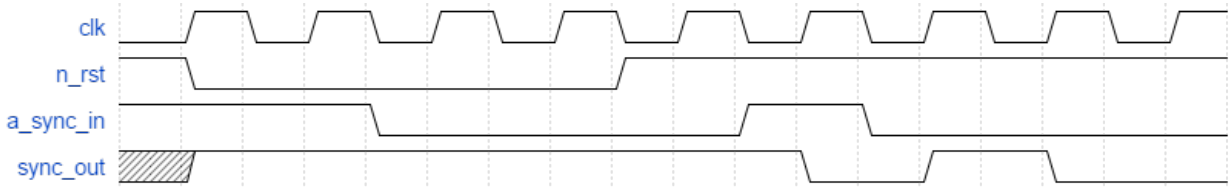

Figure 2: Timing waveform for 2-stage synchronizer for an active low input

2.4 Synchronizer Design Testing

Unlike you prior designs, which were purely combinational, the synchronizers are sequential designs which mean that input timing combinations now become important instead of just value combinations. However, it is possible to effectively exhaustively test a design as simple as a synchronizer, one just has to focus on the classes of input timings that are important for the design. In this case there are three major timing classes with respect to the clock and a data input value: setup time violations, hold time violations, and nominal timings. Within each of these input timing classes, the behavior of a synchronizer will be the same for all different timing combinations, so one only needs to create a single scenario for each class in order to effectively exhaustively test those three classes of design behavior. To aid you in your testing you have been provided with a starter test bench for the synchronizer which has test cases for all three timing classes for the clock signal and one data input value, as well as reset conditions. You will need to extend it to cover the three timing cases for the other data input value. The starter test bench is structured with Verilog tasks for handling repeated testing work and Section 3 discusses both the syntax and recommend approaches to using Verilog tasks for efficient test bench design.

Also, pay attention to the behavior of the stored/output values of the flip-flops involved during the timing violation cases and compare that to what you know about the role of a synchronizer in a system and the discussion of metastability in prior digital design classes. Since the purpose of a synchronizer is to filter out metastability from an input directly or indirectly via timing violations source simulations are not really useful outside of verifying nominal operations. This is because source simulations do not model timing information for any aspect of the system which is a critical part of the non-ideal behavior for synchronizers.

Additionally, since source simulations do not involve actual gate or flip-flop models they do not even approximate the analog load characteristics of the flip-flops that actually cause the synchronizer to work. Therefore, source simulations of the provided test bench and any proper synchronizer test benches will have erroneous behavior during, and should fail, any non-ideal test cases. For these reasons, the only way to test a synchronizer in non-ideal test cases effectively is to use mapped simulations where the flip-flop loads and behaviors are at least approximated.

To submit your working synchronizer designs and exhaustive test benches for grading use the following command:

submit Lab4sync

IMPORTANT: Only submit once you have both synchronizers and their respective test benches working correctly as they are graded as a group.

3 Using Tasks in Test Benches to Manage Shared and Repeated Verification Work

Commonly when designing test benches you will find that there is repeated work among test cases that vary in both what and how things are stimulated and checked. And while test vector based loops are great for handling fully repeatable work, they do not allow one to efficiently manage work that is repeated but surrounded by other work that varies. Verilog has a specific tool for handling this called 'tasks' and while they seem like how 'functions' are used in programming languages you are already familiar with, they are more similar to preprocessor directives/macros in operation and power. Verilog also has a syntax for defining 'functions' but they are like programming functions and are both strictly scoped can only be used effectively be used to generate a values, while 'tasks' can be used to do work, like apply values to signals, both within and outside of the task's namespace. In fact the various message commands ($info, $error, $display, etc.) are actually implemented as 'tasks' not functions in Verilog.

The generally recommend syntax for defining a Verilog'task' is as follows.

```
1 task
2   [Any input signals/values using syntax similar to module port declarations]
3   [Any output signals using syntax similar to module port declarations]
4   [Any local variables or constants]
5 begin
6   < Code to execute >
7 end
8 endtask
```

When designing your test benches you will need to think about what work are you going to do either repeatedly or similarly between your various test cases and then create dedicated tasks for each meaningful chunk of work. For example when you are dealing with testing synchronous designs, like your synchronizers, you will need to apply values to design inputs based on specific timings or rules. So rather than copy an pasting code that does this kind of work, it should be made into a task that is then used whereever/whenever that work needs to be done in your test bench. An example of this kind of task is the Design Under Test (DUT) reset task in the starter test bench, which is included in Source Code 1. This specific task does not have any 'arguments' or input signals declared because resetting a DUT should always involve the same test bench signal, the signal connected to the reset port.

```
task reset_dut;
begin
    // Activate the reset
    tb_n_rst = 1'b0;

    // Maintain the reset for more than one cycle
    @(posedge tb_clk);
    @(posedge tb_clk);

    // Wait until safely away from rising edge of the clock before releasing
    @(negedge tb_clk);
    tb_n_rst = 1'b1;

    // Leave out of reset for a couple cycles before allowing other stimulus
    // Wait for negative clock edges,
    // since inputs to DUT should normally be applied away from rising clock edges
    @(negedge tb_clk);
    @(negedge tb_clk);
end
endtask
```

Source Code 1: The starter sync_low test bench's task for consistently reseting the DUT during testing

Additionally, you will generally be checking DUT outputs in similar ways among the various normal operation test cases in your test bench and during error handling test cases. So at a minimum you should always create a task for consistently performing the output checks during normal operational test cases and other tasks for handling the checking of output during your various error handling test cases. The task used for checking the sync_low's output during normal operation test cases in the starter test bench is included Source Code 2. With this task there are two inputs, the value to check for and a string to use for tailoring the message posted as a result of the check. However, since the task will only be used to check the DUT's output, it's test bench signal is directly used rather than a having an input to the task for the signal to check. Additionally, this task uses the test bench's current test case string navigation signal (tb_test_case) in the message generation. Similarly, the task for checking the sync_low's output during the various timing and metastability handling test cases is included in Source Code 3. However, since the values to check for (that it's either a logic '1' or a logic '0') are always the same for each usage, only the check specific text for the message is declared as an input.

```
task check_output;
    input logic  expected_value;
    input string check_tag;
begin
    if(expected_value == tb_sync_out) begin // Check passed
      $info("Correct synchronizer output %s during %s test case", check_tag,
    tb_test_case);
    end
    else begin // Check failed
      $error("Incorrect synchronizer output %s during %s test case", check_tag,
    tb_test_case);
    end
end
endtask
```

Source Code 2: The starter sync_low test bench's task for consistently checking the DUT output during normal operational test cases

```
task check_output_meta;
    input string check_tag;
begin
    // Only need to check that it's not a metastable value since decays are random
    if((1'b1 == tb_sync_out) || (1'b0 == tb_sync_out)) begin // Check passed
      $info("Correct synchronizer output %s during %s test case", check_tag,
    tb_test_case);
    end
    else begin // Check failed
      $error("Incorrect synchronizer output %s during %s test case", check_tag,
    tb_test_case);
    end
end
endtask
```

Source Code 3: The starter sync_low test bench's task for consistently checking the DUT output during error handling test cases

4 Flexible Counter Design

Another common building block used in hardware systems is the counter with controlled rollover. Counters are used to keep track of events that have occurred and how much time has elapsed (in terms of clock cycles or other events that have occurred). In each of the later design labs you will be needing to use a form of a counter for one of these purposes. So to improve reusability of code and minimize wasted time, you will design a flexible and scalable counter during this lab based on the knowledge and experience you have gained from the prior sections and tasks of this lab. The function of a counter is to increment an internal count value every cycle that a desired event occurs until a specified 'rollover' value has been reached at which point it will set the internal value to 1 upon the next desired event occurrence. The output should form a sequence like: 0, 1, ..., RO-1, RO, 1, 2, ... It activates the value of a rollover flag output port whenever the desired rollover value is reached and clears the value upon the next desired event occurrence. Additionally since the counters are very commonly used in control logic for designs it is imperative to keep its output logic (especially the rollover flag) as simple and fast as possible. Otherwise this delay could end up slowing down a design unnecessarily.

In order to have very fast rollover flag output logic (and be able to earn full credit for this design) you must design it so that the actual port is simply connected to output of a flip-flop and that all flag decisions are done as next state logic for that flip-flop. This optimization for an output is referred to as 'registering the output' and is common for modules that need synchronized outputs, highly stable outputs, or low-delay (relative to the clock) outputs.

4.1 Design Specifications

Required Module Name: flex_counter

Required Filename: flex_counter.sv

Required Ports:

Port name	Direction	Description
clk	input	The system clock. (Maximum Operating Frequency: 400MHz)
n_rst	input	This is an asynchronous, active-low system reset. When this line is asserted (logic '0'), all registers/flip-flops in the device must reset to their initial value.
clear	input	This is the active-high signal that forces the counter to synchronously clear its current count value back to 0.
count_enable	input	This is the active-high enable signal that allows the counter to increment its internal value
rollover_val[#:0]	input	This is the N-bit value that is checked against for determining when to rollover. The actual port declaration should use the NUM_CNT_BITS parameter value to determine the value of '#'.
count_out[#:0]	output	This is the current N-bit count value stored in the counter. The actual port declaration should use the NUM_CNT_BITS parameter value to determine the value of '#'.
rollover_flag	output	This is the active-high rollover flag and must be asserted when the counter is at the rollover value and cleared with the next increment.

In general, it is best to focus on design functionality first and then optimize timing/performance after you have a functionally correct design. With this in mind, you should first start out by implementing and validating a design that satisfies the core functionality requirements before updating the design to use the timing optimization of 'registering' the rollover_flag output.

Figure 3 illustrates the expected behavior for a 4-bit version of this flexible counter design. The orange filled count value results from the asynchronous reset, the yellow values result from the clear pin, the blue values result from counting due to the active count enable, and the white values result from maintaining the current count value.

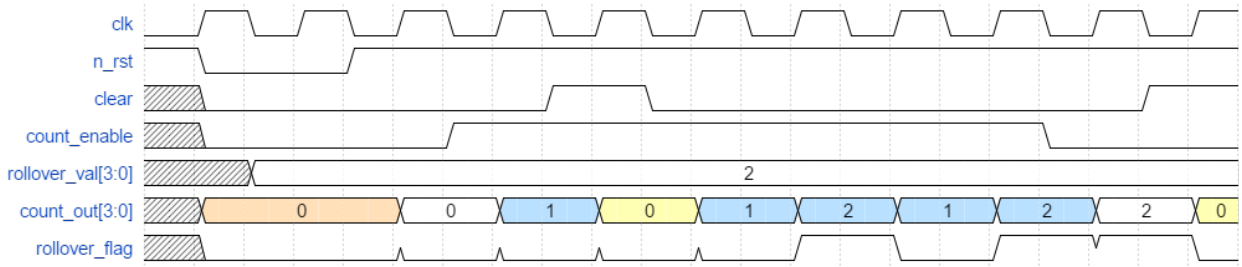

Figure 3: Example 4-bit Counter Timing Waveform Diagram

4.2 Flexible Counter Grading

You will need to create a RTL diagram for the counter design and have it signed off for correctness by a TA. It is expected that you will have this step done before beginning to implement your design and as such, course staff will be instructed to refuse to help with design debugging if you do not have a valid RTL diagram for this design. You will need to verify your design using a test bench that fully covers the default parameter value form of your design, as well as another test bench for testing a scaled version of your design (using a wrapper file like was used for the 8-bit and 16-bit adder testing). Furthermore, your design will not be able to earn full credit if it does not have a 'registered' rollover_flag output. Section 4.3 outlines the minimum requirements for your flex counter test benches. Once you have completed these steps, submit your design's code and default parameter value form test bench for grading, use the following command:

submit Lab4fc

Make sure to test your design using different values for NUM_CNT_BITS. Also make sure that the tb_flex_counter.sv test bench submitted only tests your flex_counter design with default values. Otherwise the grading of your test bench may not work properly. Your scaled design should be tested with a separate test bench design/file.

4.3 Test Bench and Minimum Verification Requirements

4.3.1 Required Minimum Test Cases

- Power-on-Reset
- Rollover for a rollover value that is not a power of two
- Continuous counting
- Discontinuous counting
- Clearing while counting to check clear vs. count enable priority

4.3.2 Required Verification Tasks used in Test Bench

- DUT Reset Task
- Normal Operation DUT Output Check Task (both count_out and rollover_flag must be checked)
- Normal Clear Task (pulse the clear for 1 cycle)

5 Closing Remarks

- Turn in your check-off sheet at the beginning of Lab Lab 5 .