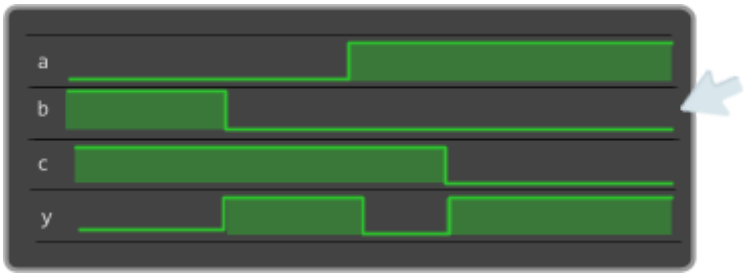
Lab 6: Simulation and Testbenches

ECE 211: Digital Circuits I — Spring 2025

Abstract. In the lab, we will learn how to simulate circuit operation and view timing waveforms in Vivado. The Vivado simulator is a critical part of the design workflow because it allows us to easily check the functional operation of our circuit before uploading a bitstream onto the FPGA board. Additionally, simulation allows us to avoid the often time consuming process of implementing and synthesizing a circuit design. This lab assignment will get you acquainted with the Vivado simulator and Wave view. In the lab assignment, you will modify circuits to remove glitches, create test cases to verify the operation of circuits, and ultimately use these test cases to fix an incorrect circuit design.



Objectives.

- Test the operation of a circuit using simulation
- Implement a testbench for a combination circuit
- Use simulation and testbenches to identify bugs in a circuit implementation

Materials and Equipment.

- Nexys A7-100T FPGA board with USB cable

Deliverables.

- In-person lab demonstration (per team)
- Lab report (per team)

Part 0: Set Up a Vivado Project

To begin the lab, we will set up a new Vivado project for Lab 6 and configure the Nexys A7 board. You can reference Lab 0 for a more detailed description of these steps.

1. Create a Xilinx Vivado project and import the provided files.

2. In this lab, we will primarily use Vivado's simulator rather than the FPGA board. If you would like, you can still connect your FPGA board and open the hardware target at this point in the lab.

Part 1: Simulating with the Vivado Simulator

To begin the lab, we will learn about how to use the Vivado simulator.

Why Simulate?

Simulation is a key capability of SystemVerilog because it allows us to check for errors in our module descriptions *before* we realize them in hardware. Simulation allows us to view not only the inputs and outputs of the circuit, but also intermediate signals. This allows us to locate errors quickly. Further, this technique is necessary for large, complex designs, which often takes minutes to hours to perform synthesis and implementation! Waiting an hour to check your design after modifying a single line of code is ultimately not an efficient way of developing circuits—simulation fills this gap.

Simulation can be applied at several points in the design flow. Often, it is one of the first steps after design entry and one of the last steps after implementation as part of verifying the end functionality and performance of the design. Simulation is an iterative process and is typically repeated until both the design functionality and timing requirements are satisfied.

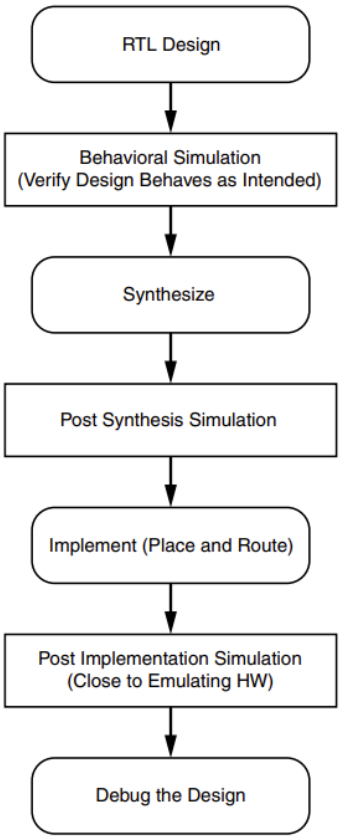


Figure 1. Simulation flow for a typical design, courtesy of the Vivado User Guide

How to Observe/Investigate a Vivado Simulation

There are two ways to monitor a simulation in Vivado: by looking at the circuit's ***waveform***, or by printing messages about different signals via the ***console monitor***. In this lab, we will explore both techniques. Typically, printing messages allows us to quickly verify that the inputs and outputs to our circuit are working properly. If the circuit is not working correctly, investigating the waveform can often pinpoint the error.

How Does the Simulation Work?

Xilinx Vivado is ultimately a computer-aided design (CAD) tool to build digital circuits. Vivado, and other CAD tools, use a technique called ***event-driven simulation*** to capture the behavior of logic circuits. Within a circuit, each signal change is treated as a discrete event at a specific point in time. Logic gates are then modeled based on their response to these events.

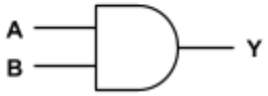
We, the digital circuit designers, are in charge of specifying when we want the input signals to change. This is done using ***delay statements*** (`#10`), which delays the simulation for a set number of timesteps. Delay statements are also used to assign individual gate delays. Without delay statements, Vivado will update a signal or output value instantaneously!

For example, in Figure 2, **a** changes to 1 after 10 ns, and **b** changes to 0 after 20 ns (the two delay statements compound). The AND gate is also given a 5 ns delay. A timing diagram and the waveform output by Vivado are both shown below.

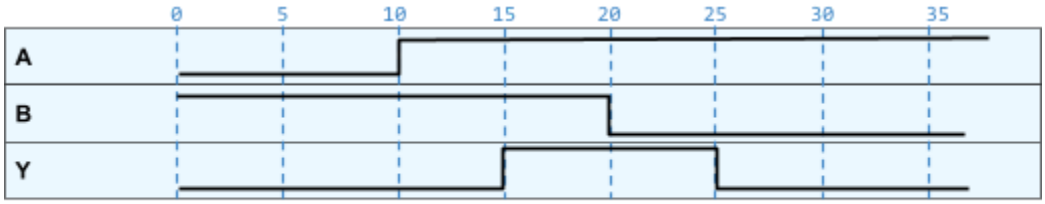
```
logic a, b, y;
// Connect an AND gate with 5 ns delay
assign #5 y = a & b;

// Specify a sequence of procedural statements
initial begin
  a = 0;   // Set values of a and b at t=0
  b = 1;
  #10;     // Delay 10 ns
  a = 1;   // Update a after 10 ns delay (t=10)
  #10;     // Delay 10 ns
  b = 0;   // Update b after 10 ns delay (t=20)
end
```

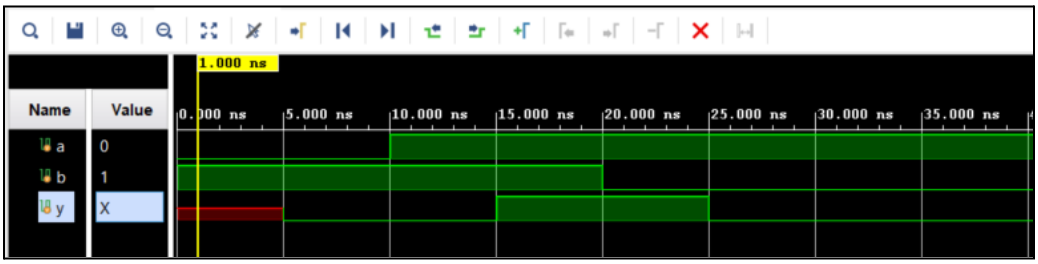
(a) SystemVerilog Code



(b) Circuit Schematic



(c) Timing Diagram



(d) Vivado Waveform

Figure 2. **Simulation of an AND gate with a 5ns delay**

The delay statements that dictate the input changes are placed inside a ***procedural block*** in Figure 2a. This code *does not* execute like our other SystemVerilog statements, which are continuous assignments that model a wire connected to some circuit component. Rather, this code executes once, line-by-line, when the simulation begins. More details about the procedural block are found in Part 2 of this lab manual.

In Part 1 of this lab, you will simulate the provided circuit and get acquainted with the waveform and monitor features of Vivado simulations.

1. Import the provided SystemVerilog modules `lab06_p1_top.sv` and `circuit_p1.sv` as ***simulation sources***. If Vivado does not automatically make `lab06_p1_top.sv` the top-level module, you can do so by right-clicking the file in the Sources pane and selecting "Set as Top."

2. Look at `lab06_p1_top.sv`. This file instantiates a `circuit_p1` module named DUT and connects its input and output ports. This module is the device that we want to simulate, or test. Thus, it is appropriately named DUT for *Device Under Test.* The module name DUV for *Device Under Verification* is also commonly used.

   The file also contains a procedural block that begins with the keywords "initial begin." This block of code is used to send inputs to the circuit that change over time. You will learn more and modify this block of code in Part 2.

3. Look at `circuit_p1.sv`. This implements the circuit shown in Figure 3 and uses a delay statement (`#1`) to assign each gate a 1 ns delay. The implementation is incomplete. Fix the module implementation to fully describe the circuit shown in Figure 3.

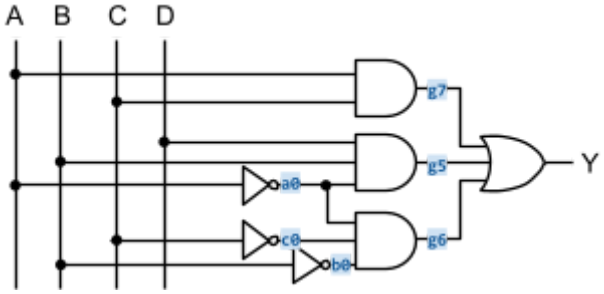


Figure 3. **Circuit implemented by the module `circuit_p1.sv`**

4. Before simulating the circuit, predict what the output waveform will look like. Complete the timing diagram on Q1 of the lab worksheet.

5. Simulate the circuit and view the waveform in Vivado. In the Flow Navigator on the left side of the Vivado window, select "Run Simulation" and then "Run Behavioral Simulation." After a short period, the waveform viewer will appear if there are no syntax errors in the SystemVerilog code.

6. Explore the Wave view:
   a. When you invoke the simulator it opens the Wave window by default, displaying the signals from the top-level module.
   b. Use the Wave window's toolbar to adjust the waveform view. A screenshot of the toolbar is provided below.

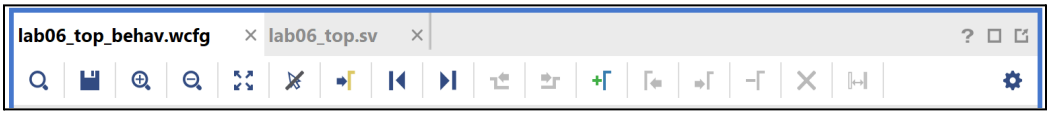


Figure 4. **Wave view toolbar**

c. Use the following buttons to adjust the Wave view: *Zoom In*, *Zoom Out*, *Zoom Fit*, *Go to Cursor*, *Go to Time 0*, *Go to Last Time*.

d. The names for each signal are listed on the left side of the Wave window. Click on the signal **y** to select it. Then, use the buttons *Previous Transition* and *Next Transition* to traverse the singal's changes over time.

e. Select *Zoom Fit* to see the entire waveform in one view. Compare the waveform for **y** to your worksheet. Does your worksheet match the simulated circuit?

f. Manually drag the cursor (vertical yellow bar) left and right. Place the cursor at 12.5 ns. What are the values of a, b, and y at 12.5 ns? Note this on your worksheet.

g. Go to 0 ns using the button *Go to Time 0*. At time zero, the output is red. What does this coloring mean? Why does the signal **y** have this value? Answer this question on your worksheet.

7. Add the circuit's intermediate signals to the wave view:

a. The Scope window (shown in Figure 5), shows the design hierarchy for the simulated circuit. A *scope* is a hierarchical partition of a circuit design. A scope is internal to a particular module instance .

b. Select the **DUT** in the Scope window. All signals contained by that scope will appear in the Objects window.

c. Add the intermediate signals to the waveform. To add a signal to the Wave window, you can right-click the signal name in the Objects pane and select "Add to Wave Window." Alternatively, you can drag and drop the signal into the Wave window.

Note: If the signal shows up in the Wave window but has no waveform, you will need to relaunch/refresh the simulation.

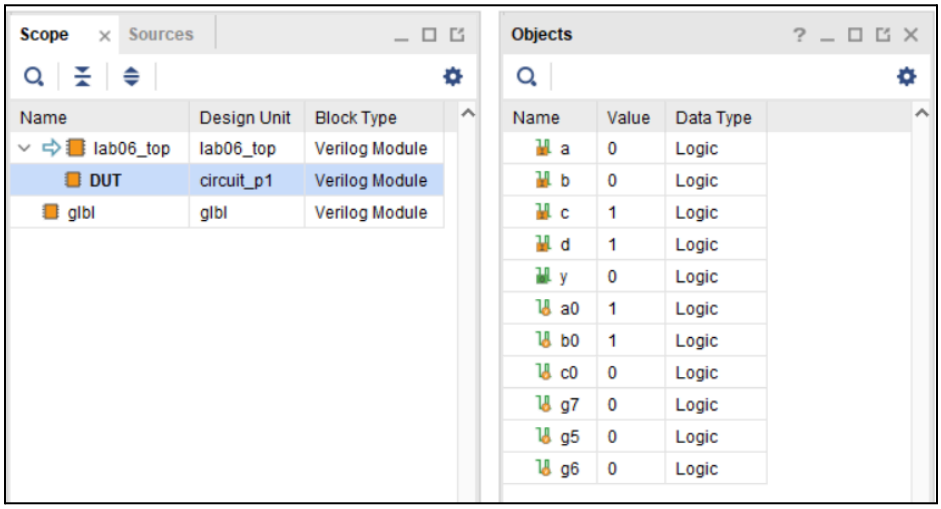


Figure 5. **Scope and Objects windows, showing all signals for the circuit_p1 instance**

8. Modify the Wave style:

a. In the Wave window, explore options to customize the waveform. Right-click a signal name and try the menu options: *Name*, *Rename, Signal Color, New Group,* and *New Divider*.

b. Change some properties of the waveform to make it more readable. Show your modifications to an instructor.

9. Now that you are acquainted with some of the basic properties of viewing a waveform, modify the circuit implementation to eliminate its glitches!

   a. The circuit implemented by `circuit_p1.sv` has a glitch in the output `y`, as shown by your timing diagram. Redesign the circuit to be glitch free by adding additional gates to the implementation.

      Hint: One suggested way to proceed is to start by writing the Boolean equation and creating a K-Map for the current circuit. By analyzing the K-Map and adding consensus terms, you can resolve the glitches in this circuit.

   b. Modify the SystemVerilog implementation to implement your glitch-free circuit and re-run the simulation to ensure that the output `y` is constant.

   c. Draw the schematic of your glitch-free circuit on your lab worksheet.

10. Before moving to the next part, remove the delay statements within the file `circuit_p1.sv`. How does your waveform change? How does SystemVerilog model gate delays when delay statements are excluded? Write your thoughts about these questions on the lab worksheet.

Part 2: Creating Testbenches

In the second part of this lab, you will create your own testbench for your `adder_4b` module from Lab 5 to test its operation. You will then simulate the circuit and use both the Wave view and monitor to observe its outputs.

A ***testbench*** is a SystemVerilog module that instantiates a device under test and provides inputs to it that change over time. Sometimes, both of these tasks are combined in one top-level module, like in Part 1. However, testbenches often contain individual ***stimulus*** generator modules that are then connected to the DUT, as shown in Figure 6. Encapsulating the input generation in its own module can help simplify the top-level module. A stimulus module simply contains a procedure block with a set of test cases.

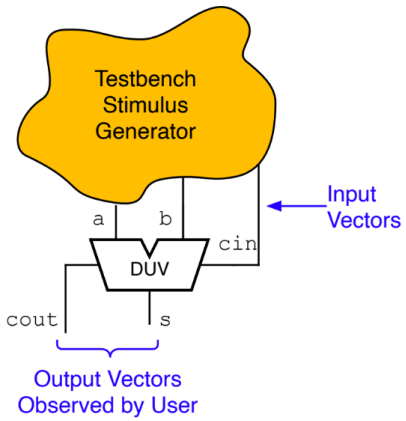


Figure 6. Testbench Structure

The following sections detail some of the SystemVerilog syntax you may need when creating or modifying your testbench:

Literals (Constants)

SystemVerilog supports “integer literals” which are constant whole numbers that have no fractional (decimal part). In this lab, we will use literals in order to create different test cases for our DUT.

A literal can be specified by its number of bits, followed by an apostrophe, the numerical base (**d** for decimal, **h** for hexadecimal, **b** for binary, and **o** for octal), and finally the value. The following code snippet shows some examples of different literal values in SystemVerilog. The number of bits is optional and does not need to be included.

```
logic [3:0] a;  // a is a 4-bit vector
...
a = 4’b0011;    // set a to the value 0011 in binary
a[3] = 1’b0;    // set the MSB of a to 0
a = 4’d3;       // set a to the value 3 in decimal
```

Procedural Blocks

Procedural code is intended to drive simulations—it cannot be realized into actual hardware like the other SystemVerilog code we have written in this course. These blocks of code are exclusively used in testbenches and employ a special operator (#) to delay simulation.

A block of procedural code begins with the `initial` keyword followed by a `begin` and `end` statement that bookend the block’s contents. This syntax is shown in the following code snippet. Each line of code sets an input signal to a particular constant and then halts simulation for a particular amount of time (*e.g.*, 10 ns), before changing the input values. Ideally, by testing a sufficient number of input combinations, the stimulus can ensure that our circuit operates correctly (or can identify when it does not!).

```
initial begin
        $display("[ECE 211] Simulation starting...");

        // Drive stimulus and pause for 10 ns
        a = 4'd3;
        b = 4'd4;
        #10;
        $display("[ECE 211] 0x%0h + 0x%0h = 0x%0h at time=%0t”, a, b, y, $time);
        $display("[ECE 211] Simulation complete...");
        $stop;
end
```

Simulation Statements ($)

We can include additional commands within our procedural block to manage the Vivado simulation.

- **`$stop`**: Vivado simulations will run for 10,000 nanoseconds or until a stop command is reached. In the above procedural block, we use the stop command to halt the simulation after 10 ns. Note that simulation statements are ended by a semicolon (`;`) like all SystemVerilog statements.
- **`$display(...)`**: The display command prints a message to the Tcl Console at the bottom of Vivado. This can be a simple sentence, like done by the first and third display commands in the above example. This can also be used to print the current values of signals, like done by the second command in the above example. In this command, the `%0h` parts of the sentence

are replaced by the current values of signals `a`, `b`, `y`, and a system variable capturing the current time.

- `$monitor(...)`: This command will automatically print the value of a signal whenever that signal changes.

You can read more about display tasks here: https://www.chipverify.com/verilog/verilog-display-tasks.

11. Import the provided SystemVerilog module `lab06_p2_top.sv` and your `adder_6b.sv` module from Lab 5 as simulation sources. You will also need to import any submodules from Lab 5, such as `full_adder.sv`. If Vivado does not automatically make `lab06_p2_top.sv` the top-level module, you can do so by right-clicking the file in the Sources pane and selecting "Set as Top."

    Hint: Close the Simulation window and set `lab06_p2_top.sv` *as top file.*

12. Instantiate your `adder_6b` module.

13. Within a procedural block, create a series of *at least five* tests for your adder module. A single test is a specific set of input signals that you are looking to evaluate (*e.g.,* a=3 and b=4).

    Hint: Figure 2a shows a sample test case for an AND gate. Use a similar approach to create a test case for your 4-bit adder.

14. Simulate your circuit and verify your circuit's operation using the Wave view.
    a. Note that we are creating a behavioral testbench for our adder module—we are *not* modeling gate delays! Rather, we are purely interested in whether the inputs and outputs function correctly. SystemVerilog testbenches are most commonly used for this purpose.
    b. Signals that are vectors will be bundled together in the Wave view. Rather than displaying a waveform for each bit of the vector, Vivado will simply display the value of that vector. You can view the individual waveforms by selecting the dropdown arrow next to the signal's name, as shown in Figure 7.

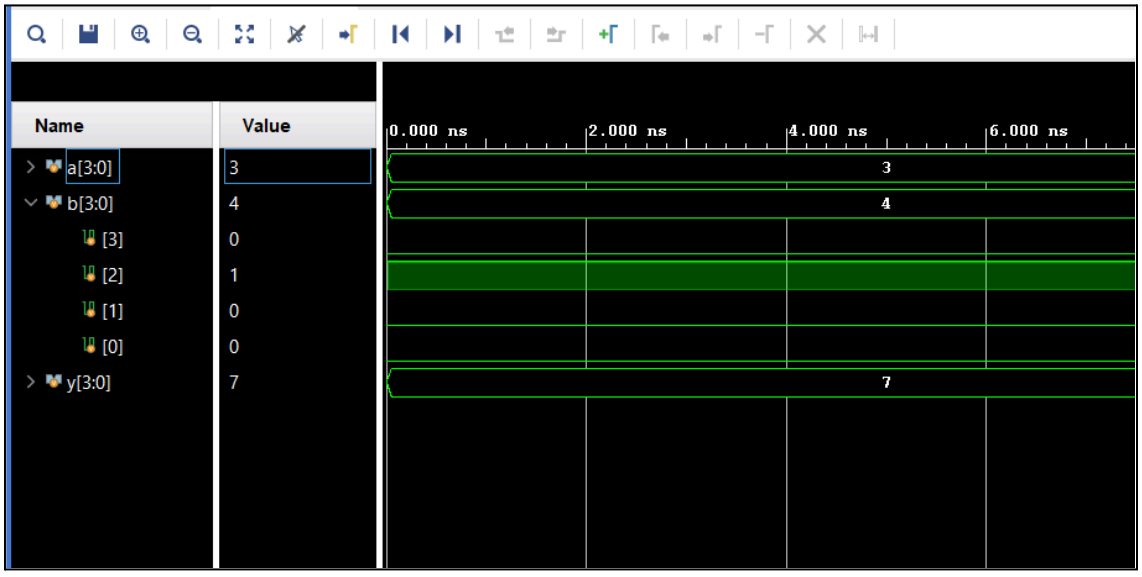


Figure 7. Wave view for vectors with the b waveform expanded

15. Change the numerical value displayed for your vector waveforms to binary. Right-click the signal for a and select “Radix” then “Binary” to change this view on the waveform.

16. Waveforms are helpful, but they can be hard to use to quickly verify a circuit’s behavior. Modify your top file to include display and/or monitor statements to print each test case to the Tcl Console. Re-run your simulation to verify their operation. Show your final waveform and console outputs to an instructor.

Part 3: Using Testbenches to Identify Bugs

In the final part of this lab, you will create and use a testbench to identify and resolve bugs in your circuit. The provided SystemVerilog file `priority_encoder.sv` is an incorrect implementation of an 8-to-3 priority encoder (as discussed in Lecture 11). There are a couple of bugs in the implementation that you must fix in order for the circuit to work properly. As you cannot debug by inspection, you will need a testbench..

17. On the worksheet, write a set of useful test cases for an 8–to-3 priority encoder.

18. Import the provided SystemVerilog module `priority_encoder.sv`.

19. Create a new top-level module under ***simulation sources*** named `lab06_p3_top.sv`. Within this module, instantiate a priority encoder DUT and connect it to a set of inputs and outputs. Within a procedural block, implement your test cases to connect to your priority encoder.

20. Simulate your circuit and evaluate its inputs and outputs. Where is there an error? See if you can use information from the Wave view, such as the encoder’s intermediate signals, to pinpoint the source of the error.

    Hint: There is an X on your output waveform. Work on pinpointing the source of this uninitialized value first.

    Hint: There are two fixes you must make to get the priority encoder to work properly. You do not necessarily need to trace all of the internal signals in the module to identify these fixes!

21. Find and fix any bugs that impact the operation of the circuit.

Part 5: Wrap-Up

Before leaving the lab, make sure to complete the following steps!

1. ***Demonstrate the operation of your circuit to the instructor and submit your lab worksheet.***

2. Turn the Nexys board OFF and disconnect it from the computer. Make sure to save your Vivado project and sign out of your machine!