ECEN 248: Introduction to Digital Design
Department of Electrical and Computer Engineering
Texas A&M University

Laboratory Exercise #7
Introduction to Sequential Logic

Lab exercise created and tested by
Abbas Fairouz, He Zhou, Joshua Mashburn, and Sunil P. Khatri

1 Introduction

The purpose of lab this week is to introduce sequential logic circuits. To start off, we will cover storage elements such as latches and flip-flops. You will have the opportunity to describe these components at the gate-level using Verilog and then simulated them within Vivado. Next, synchronous sequential circuits will be discussed. To make the concepts more clear, you will combine flip-flops with combinational logic discussed in previous labs to simulate the operation of synchronous logic. Furthermore, you will introduce simulation *delays* into your combinational logic and observe the effects it has on clock timing.

2 Background

The following subsection will expound on the theory necessary for completion of this lab assignment. The pre-lab will test your knowledge of these concepts.

2.1 Latches and Flip-flops

What differentiates sequential circuits from the combinational circuits you have designed thus far is the memory component. Simply put, the output of a sequential circuit is dependent on not only the current inputs but also the 'state' of the circuit (i.e. values stored in memory components). The 'state' of a sequential circuit can viewed as a historical record of past inputs. Figure 1 illustrates this concept.

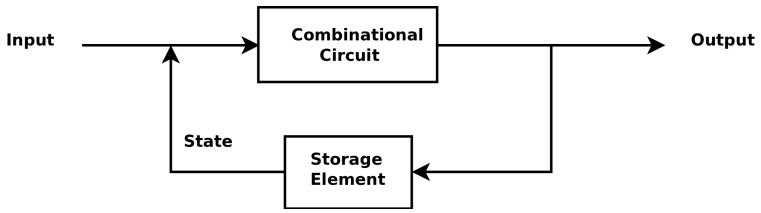

Figure 1: Simplified Diagram of a Sequential Circuit

Memory components come in many flavors but we will only study two types in this lab, namely latches and flip-flops. Latches can be combined to create flip-flops so we will begin by discussing latches. The simplest latch is the Set-Reset (SR) latch. The function table and gate-level schematic of an SR latch with an enable, *En*, are depicted in Table 1 and Figure 2, respectively. As you can see, the Set and Reset operations are expected to be mutually exclusive, meaning they cannot happen at the same time. However, when neither Set nor Reset is active, the value of Q (and consequently $\overline{Q}$) remains the same. Hence, the value of Q is latched or stored. From the SR-latch, we can construct a D-latch, also known as a transparent

latch, as seen in Figure 3. The function table for the D-latch can be found in Table 2. The behavior of the D-latch is simple. When *En* is high, the input, D, is forwarded to the output, Q (i.e. the input is transparent); however when En is low, the output simply holds the last value of Q.

Table 1: SR-latch Function Table

En	S	R	Q	$\overline{Q}$
0	X	X	hold	
1	0	0	hold	
1	0	1	0	1
1	1	0	1	0
1	1	1	undefined	

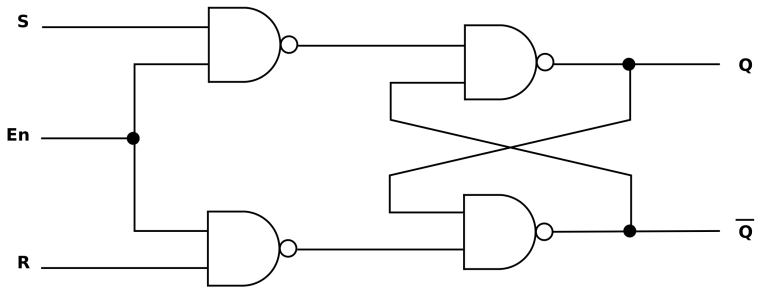

Figure 2: SR-latch Gate-level Schematic

Table 2: D-latch Function Table

En	D	Q	$\overline{Q}$
0	X	hold	
1	0	0	1
1	1	1	0

A D-latch is considered a *level-sensitive* memory device because the output is controlled by the level (i.e. high or low) of the En signal. Conversely, we can create an *edge-sensitive* device such that the output changes when the control signal transitions from low to high or vice versa. This sort of device is known as a flip-flop, and the control signal is usually referred to as a Clock (for reasons which will become clear later in the lab) or Clk for short. Consider the circuit in Figure 4 in which a level-sensitive D-latch and an edge-sensitive D flip-flop are stimulated with the exact same input. Figure 5 depicts the simulation waveform

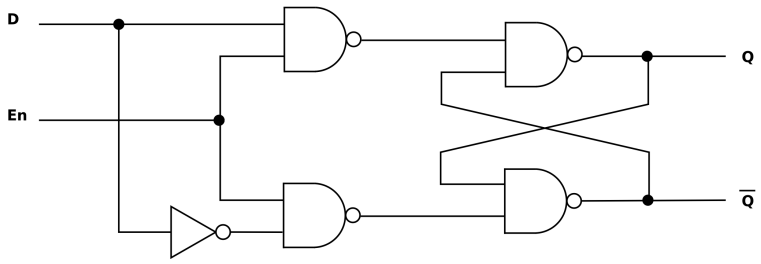

Figure 3: D-latch Gate-level Schematic

for the D-latch/D flip-flop circuit. Notice that the output of each memory component is undefined in the simulation until Clk goes high at 10 ns (10,000 ps). While Clk is high, the output of the D-latch (*Q_latch*) matches the input net (D) whereas the output of the D flip-flop (*Q_flip_flop*) holds the value that was on the D signal at the exact moment the rising edge of Clk came in. At 15 ns, D drops low as does *Q_latch*; however, *Q_flip_flop* does not respond to D dropping low until the next rising edge event from Clk. Notice the pulse on D at 22 ns. Neither the latch nor the flip-flop respond because the Clk signal is low and not transitioning (i.e. no rising edge).

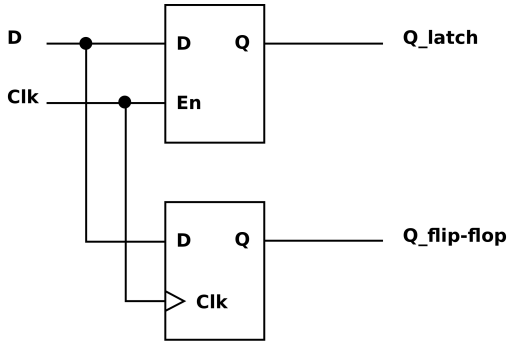

Figure 4: Latch/Flip-flop Circuit

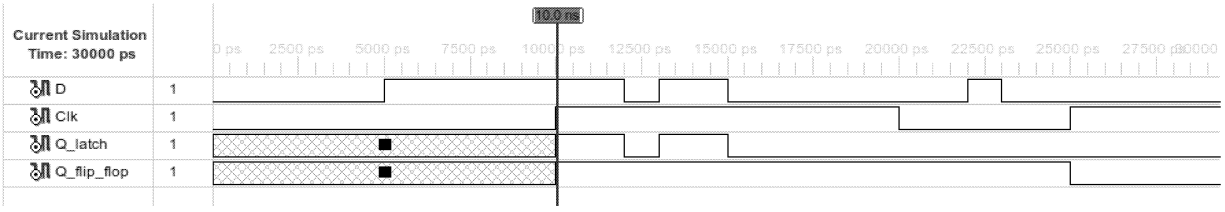

Figure 5: Simulation Waveform of Latch/Flip-flop Circuit

To construct a D flip-flop, we can actually combine two D-latches with two inverters in what is often referred to as a *master-servant* configuration illustrated in Figure 6. Remember that a rising-edge sensitive D flip-flop essentially samples the input, D, at the rising-edge of Clk. Let us convince ourselves that the circuit in Figure 6 does exactly that. When Clk is low, the master latch is enabled while the servant latch is disabled. Thus, the output of the master latch, Q_m, equals the input, D, whereas the output of the servant latch, Q_s, is equal to whatever value was last store in that latch. For this example, let us assume D = '1', and the value latched into the servant latch is 'X'. At some point, the Clk will transition from low to high. This event is referred to as a rising-edge. When this happens, the master latch will now become disabled, which means it will hold the value of D immediately before the transition (i.e. '1' will be latched into the master latch). Conversely, the servant latch will become enabled and therefore transparent (i.e. $Q_s = Q_m$ = '1'). When the clock returns to a low state, the servant latch will simply hold the previous value of Q_m which is '1'. In the experiments below, you will have the opportunity to simulate this behavior to help you further understand how the master-servant D flip-flop works.

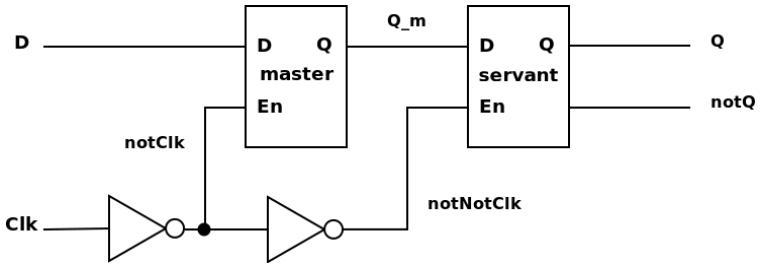

Figure 6: Master-Servant D flip-flop

2.2 Synchronous Sequential Circuits

Sequential circuits can be broadly classified into two categories, asynchronous and synchronous. Simply put, asynchronous logic utilizes latches, while synchronous logic utilizes flip-flops. In this lab, we will focus on synchronous logic because asynchronous logic requires a far more advanced understanding of signal timing and propagation. Synchronous logic, as opposed to asynchronous logic, requires an external synchronizing control signal appropriately named the 'clock' signal. Often abbreviated, Clk, the clock signal acts like the heart beat of the synchronous circuit, controlling the exact moment when data transitions from one flip-flop to the next.

As an example, consider the circuit in Figure 7, which is a 2-bit *synchronous* adder. This circuit is constructed from three 2-bit wide flip-flops[1] , one 1-bit wide flip-flop, and the familiar ripple carry adder. One thing to note in the synchronous circuit diagram is that the Clk signal is not actually shown. The $>$ symbol found within the flip-flops implies a clock signal is distributed to those components. The clock

[1] An n-bit flip-flop (a.k.a n-bit register) is simply n 1-bit wide flips-flops arranged in a parallel fashion.

signal that drives the synchronous adder is shown below the circuit diagram. The period of oscillation, T, is determined by the propagation delay of the 2-bit ripple carry adder. The rate of oscillation, f, is computed as $1/T$. As a digital circuit designer, your goal will often be to reduce the propagation delay through the combinatorial logic within a synchronous circuit in order to drive the circuit with a higher clock rate.

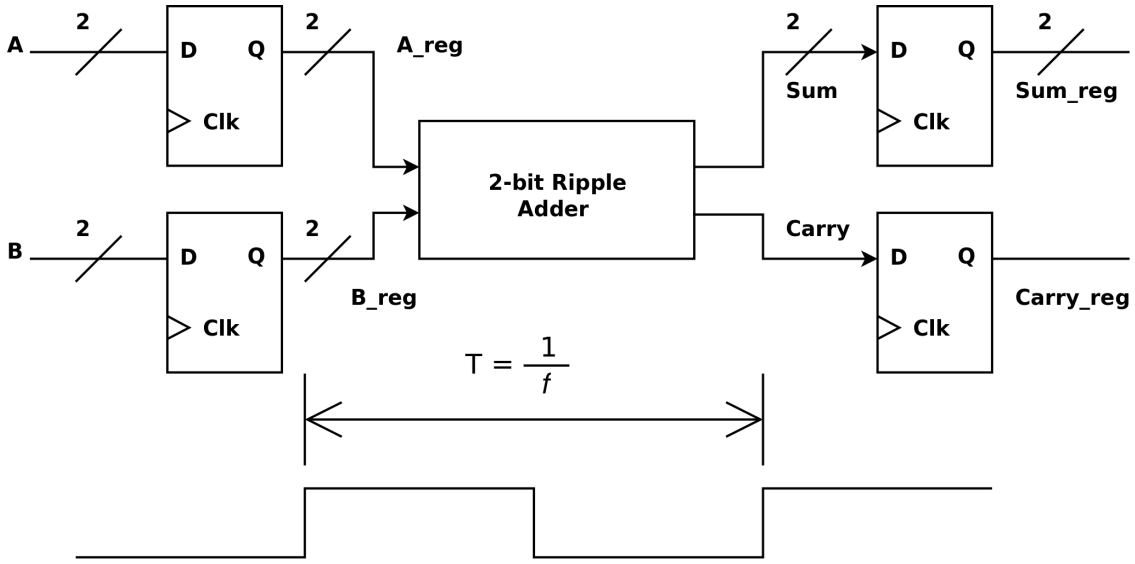

Figure 7: Example Synchronous Logic Circuit

2.3 Assignment in Verilog

Verilog supports two types of assignments within **always** blocks:

- Blocking assignment: evaluation and assignment are **immediate**, so statement order matters.

```
1     always@(posedge Clk)
      begin
3             x = 1;
              y = 0;
5             y = x;          // x=1, y=1
              z = x & y;      // z=1
7         end
```

- Non-blocking assignment: all assignments are deferred until all right-hand sides have been evaluated. Think of the assignments as not being written until the "end" of the block is reached.

```
always@(posedge Clk)
begin
        x = 1;
        y = 0;
        y <= x;          // x=1, but defer assignment, y=0 right now
        z <= x & y;      // x & y = 0, but defer assignment
    end                  // y=1 and z=0
```

After reading through the code examples above, it can be found that final results of z are different. In conclusion, we only use blocking assignment for combinational always blocks and use non-blocking assignment for sequential always blocks.

3 Lab Procedures

The following lab exercises serve to reinforce the material discussed in the background section. By following the procedures, you will write Verilog code to implement a 2-bit synchronous adder.

3.1 Experiment Part 1

We will begin by using structural Verilog to describe different memory components, as well as to simulate them in Vivado. To do so, we must introduce simulation **delays** into our Verilog code. Furthermore, we will learn how to use behavioral Verilog to describe a D-latch and D flip-flop so that we can create synthesizable synchronous logic.

Note: Be sure to capture screenshots of all waveforms discussed below. Record the simulation console output as well.

1. Simulate the operation of the SR-latch in Figure 2 using Verilog. The steps below will guide you through the process.

   (a) Create a new Vivado project and call it 'lab7'.

   (b) Using the code below as a template, describe the SR-latch at the gate-level using *structural* Verilog. Pay attention to the delays that have been added into our code. When we place a '#2' between the module and instance name, the output of this instance will be delayed by 2 time units. Because we set the timescale to 1ns, its delay in the simulation will be 2ns. Remember that every NAND gate is delayed by 2ns.

```
`timescale 1ns / 1ps // delay unit is 1ns
`default_nettype none

module sr_latch(Q, notQ, En, S, R);
    /* port list */
```

```
    output wire Q, notQ;
7   input  wire En, S, R;

9   /* intermediate nets */
    // nandSEN is the output of NAND(S, EN)
11  // nandREN is the output of NAND(R, EN)
    wire nandSEN, nandREN;
13
    // delay of gate nand0 is 2ns
15  nand #2 nand0(Q, nandSEN, notQ);
    // fill in the rest with your own code
17
endmodule
```

(c) Copy the test bench file 'sr_latch_tb.v' from the course directory into your lab7 directory. Use it to test your SR-latch in Vivado.

(d) Once your SR-latch passes all of the tests, add the internal signals nandSEN and nandREN to the waveform. To do this:

i. Look in the **Scopes** panel and click **uut** in the list.

ii. Look in the **Objects** panel. You should now see nandSEN and nandREN in the list of wires.

iii. Click and drag nandSEN and nandREN into your waveform window, in the list of signal names.

iv. Along the top toolbar, find the blue **Restart** button. Click this, then click the **Run All** button immediately to its right. The waveform should now have populated for nandSEN and nandREN.

(e) After you've rerun the test bench with the internal signals, take your screenshot. Examine the waveform to gain an understanding of how the SR-latch is functioning. Compare the results to the function table (Table 1).

(f) Change all the delays in your code from '#2' to '#4'. Run the test bench again and take a screenshot. Explain the results of the simulation.

2. Simulate the D-latch in Figure 3 using Verilog.

(a) Use structural Verilog to describe the D-latch. Please make sure your module name is "d_latch", or the provided test bench will not recognize your module.

(b) Add 2ns delays to all of the NAND gates and the inverter.

(c) Copy the 'd_latch_tb.v' test bench file into your lab7 directory. Use it to test your D-latch. Take a screenshot and include the console output. Check the simulation waveform and compare the results to Table 2.

3. With the D-latch module you just created, construct a D flip-flop as shown in Figure 6.

(a) The template code is given below. Add a 2ns delay to the inverter. Do not put any additional delay on the D-latches because you already built delays into that module.

```
`timescale 1ns / 1ps // delay unit is 1ns
`default_nettype none

module d_flip_flop(Q, notQ, Clk, D);
    // declare all ports
    output wire Q, notQ; // outputs of slave latch
    input  wire Clk, D;

    /* intermediate nets */
    wire notClk, notNotClk;
    wire Qm; // output of master latch
    // notQm is used in instantiation
    // but left unconnected
    wire notQm;

    // instantiate the NOT gates, add 2ns delay

    // instantiate the D-latches
endmodule
```

(b) Copy the test bench file 'd_flip_flop_tb.v' into your lab7 directory. Use it to test your D flip-flop.

(c) Add internal signals Qm, notClk to the waveform and rerun following the same process given for the SR-latch. Take a screenshot, including the console message.

4. When developing a synthesizable Verilog module, we can only use behavioral modeling to describe memory components. As mentioned in the previous lab, **reg** is the only legal type on the left-hand side of = or <= signs in an always block. The following code shows how to create a D-latch with an always block.

```
`timescale 1ns / 1ps    // delay unit is 1ns
`default_nettype none

module d_latch_behavioral(
    output reg Q,      // driven with a behavioral statement
    output wire notQ, // driven with a dataflow statement
    input wire D, En  // wires can drive regs
);

    /*describe behavior of D-latch*/
    always@(En or D)
        if(En)     // if En != 1'b0
            Q = D;
        //else Q = Q; // this is implied, so it's unnecessary
    assign notQ = ~Q; // regs can drive wires
endmodule
```

Although the code above is syntactically correct, it is not appropriate for an FPGA because the FPGA is designed to implement *synchronous* logic, and latches are used create *asynchronous* logic. Even if your code works in simulation, it may still not work correctly on the FPGA. The details will not be explained in this class, but we strongly recommend that you only use flip-flops in your sequential circuits as shown below.

```
`timescale 1ns / 1ps // delay unit is 1ns
`default_nettype none

module d_flip_flop_behavioral(
    output reg Q,
    output wire notQ,
    input wire D,
    input wire Clk // clock signal
);

    /* describe behavior of D flip-flop */
    // posedge means positive (rising) edge
    always@(posedge Clk) //trigger when Clk goes HIGH
        Q <= D; // non-blocking assignment statement
    assign notQ = ~Q;
endmodule
```

Type the code above into two source files named 'd_latch_behavioral.v' and 'd_flip_flop_behavioral.v', respectively, and simulate each of them with the corresponding test bench files. After seeing "All tests passed", take a screenshot including the console message.

3.2 Experiment Part 2

For this experiment, we will describe the 2-bit synchronous adder drawn in Figure 7 and simulate it in Verilog. As with the previous experiment, we will begin with gate-level descriptions in Verilog so that we can demonstrate the effects of combinational circuit delay on a synchronous circuit. Then for comparison, we will show how the same circuit can be described using only behavioral constructs.

1. Use the full-adder Verilog module you created in lab 5 to construct a 2-bit ripple-carry adder.

   (a) Copy the 'full_adder.v' file from your lab5 directory to your lab7 directory and add delays to the **assign** statements in order to simulate gate delays. The code snippet below shows how you can add delay to the three input XOR gate found within the full-adder. Assume that 3-input AND, OR, and XOR gates have a delay of 6ns, while 2-input AND, OR, and XOR gates have a delay of 4ns. NOT, NAND, and NOR gates can be assumed to have a delay of 2ns.

   ```
   assign #6 S = A ^ B ^ Cin; //the hat (^) is for XOR
   ```

   (b) Create a new Verilog source file called 'adder_2bit.v' with the following module interface:

```
module adder_2bit(Carry, Sum, A, B);
  //Carry is a 1-bit output
  //Sum is a 2-bit output
  //A, B are 2-bit inputs
```

(c) To test the 2-bit adder circuit, we will use the file 'adder_2bit_tb.v' as a template. Copy this file from the course directory into your lab7 directory and **complete the test bench code based on the hints found within the comments (should be 16 test cases)**.

(d) Add the test bench to your Vivado project and simulate it. Correct any errors with the test bench or the ripple carry adder. Have the TA ensure that what you have done is correct so far.

(e) Use the simulation waveform to determine the worst case propagation delay through the ripple-carry adder.
Hint: The signal with the longest propagation path is $Carry$. You may use the time markers in the simulation window to aide in this measurement.
Hint 2: Look for a point in the waveform where the inputs change from a pair of operands that shouldn't set Carry to a pair of operands that should set Carry. See how long it takes for Carry to change from 0 to 1.

2. Now that we have our ripple-carry adder, we can complete the logic shown in Figure 7. To simplify the experiment, we will use Verilog code to complete the synchronous aspect of the design and assume the flip-flops (i.e. the storage portion of the design) are ideal and do not contribute to our timing constraints. This is not always a safe assumption so be aware!

   (a) The code below should get you started. Name the source file 'adder_synchronous.v'. Take note of how we are able to create n-bit flip-flops (a.k.a. n-bit registers) very easily in Verilog.

```
`timescale 1ns / 1ps // specify 1ns for each delay unit
`default_nettype none

/* This module could possibly be the first synchronous circuit you've *
 * ever written. How exciting?!?
 * You will probably remember this moment for the rest of your life... */

module adder_synchronous(Carry_reg, Sum_reg, Clk, A, B);

    /* Output ports are regs!!! why? well they need to be able to
       hold state */
    output reg Carry_reg;
    output reg [1:0] Sum_reg;

    /* Inputs are still wires */
    input wire Clk;
    input wire [1:0] A, B;

    /*intermediate nets*/
```

```
20      reg [1:0] A_reg, B_reg; // will use these as 2-bit registers
        wire Carry; // need this to connect to the registers described below
22      wire [1:0] Sum;

24      /*instantiate your 2-bit adder here...*/

26
        /* here is where things get interesting...*/
28      /* this behavioral block describes two 2-bit registers*/
        /* remember registers are nothing more than grouped flip-flops */
30      always@(posedge Clk) // trigger on positive edge of the clock
          begin // will need this because we will put two statements in here
32          A_reg <= A; // we use non-blocking assigments here because we want
            B_reg <= B; // these two statements to happen concurrently
34        end

36      /* well that was easy... */
        /* let's describe the registers for the result */
38      always@(posedge Clk)
          begin
40          Carry_reg <= Carry; // woah! is this right?!? ooh yeah wires can drive regs
            Sum_reg   <= Sum;   // these two statements look the same even though the
42        end                   // bit width is different

44      /* we could have actually grouped these all into one always block; however
          it is nice to clearly divide up the inputs and outputs of the circuit */
46
    endmodule //yeet
```

(b) Copy the test bench file, 'adder_synchronous_tb.v', from the course directory into your lab7 directory. Add the test bench to your Vivado project and simulate it to ensure your synchronous adder circuit passes all of the tests.

3. For the last part of this experiment, we would like to drive our synchronous adder circuit at a higher clock rate to gain an understanding of how the propagation delays through combinational logic effects the speed at which synchronous circuits are able to operate.

   (a) Once your synchronous circuit is working properly, open the test bench and change the **'define** CLOCK_PERIOD from 40 to 18. This will force the clock signal generated within the test bench to run at a higher rate. Re-simulate the test bench and take note of the results. If some tests failed, use the console output along with the waveform viewer to determine exactly why those particular tests failed.

   (b) Now increase the **'define** CLOCK_PERIOD by one and re-simulate. Repeat the process until all tests pass.

   (c) Compare the final value of the **'define** CLOCK_PERIOD with the propagation delay of your

ripple adder that you measured earlier. You will need these results to answer a question at the end of the lab assignment.

4 Lab Deliverables

4.1 Pre-lab Deliverables

Please include the following items in your pre-lab write-up:

1. Create a new Verilog source file named "adder_2bit.v" with the module named "adder_2bit". You can use your full adder module created in lab6.

2. If the circuit in Figure 7 utilizes the 2-bit carry ripple adder designed in Lab 3, what would be the maximum clock rate f given that each gate delay is 4ns? You can refer to your post-lab deliverable answer.

3. Compare all the three memory components discussed in the background part in the same table. Explain their differences and improvements.

4.2 Post-lab Deliverables

Include the source code with comments for all modules you created and the test bench code you finished. Include all screenshots you take during the lab. Ensure they show the complete waveform and the console message.

1. For 1(f) in part 1, explain the results of the simulation.

2. For 3 in part 1, check the waveform with the internal signals. Are the latches behaving as expected? Why or why not?

3. For 1.4 in part 1, compare the waveforms you captured from the behavioral Verilog to those from the structural Verilog. Are they different? If so, how?

4. For 1(e) in part 2, what is the worst-case propagation delay through the adder?

5. Based on the clock period you measured for your synchronous adder, what would be the theoretical maximum clock rate (frequency)? What would be the effect of increasing the width of the adder on the clock rate? How might you improve the clock rate of the design?

5 Important Student Feedback

The last part of the lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.
Note: If you have any other comments regarding the lab that you wish to bring to your instructor's attention, please feel free to include them as well.

1. What did you like most about the lab assignment and why? What did you like least about it and why?
2. Were there any sections of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?
3. What suggestions do you have to improve the overall lab assignment?