Laboratory Exercise #8
Introduction to Sequential Logic

ECEN 248: Introduction to Digital Design

Department of Electrical and Computer Engineering
Texas A&M University

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 ISE. 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.

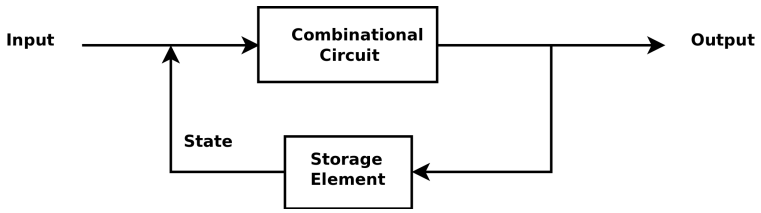


Figure 1: Simplified Diagram of a Sequential Circuit

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	

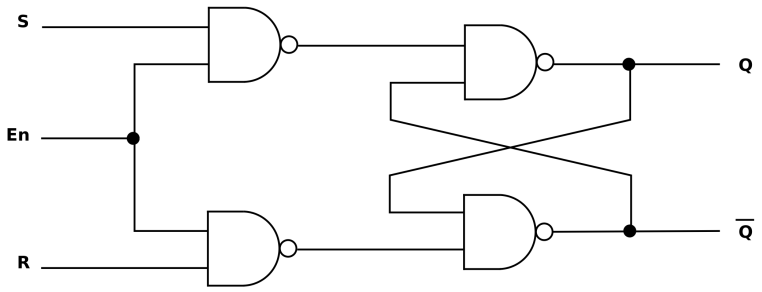


Figure 2: SR-latch Gate-level Schematic

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

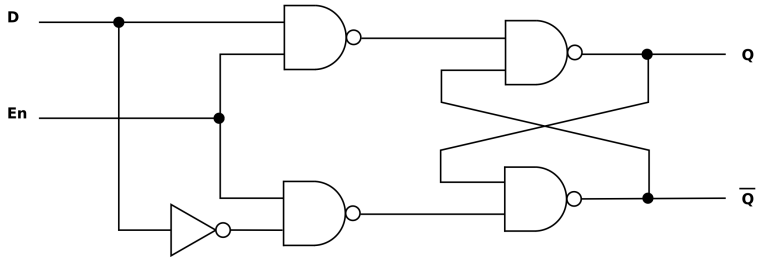


Figure 3: D-latch Gate-level Schematic

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).

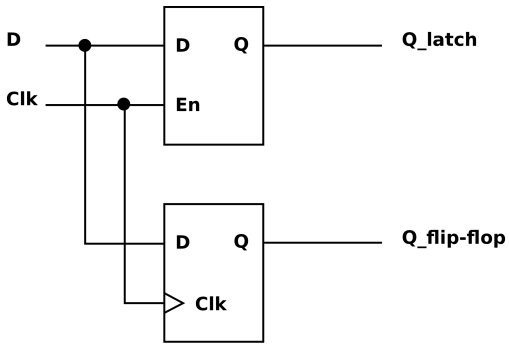


Figure 4: Latch/Flip-flop Circuit

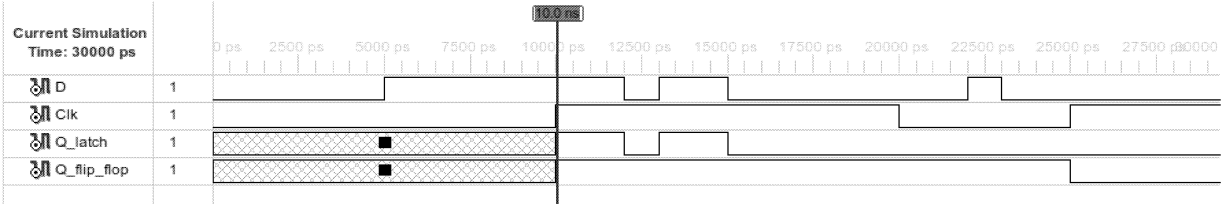


Figure 5: Simulation Waveform of Latch/Flip-flop Circuit

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-slave* 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 slave latch is disabled. Thus, the output of the master latch, Q_m, equals the input, D, whereas the output of the slave 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 slave 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 slave latch will become enabled and therefore transparent (i.e. $Q_s = Q_m$ = '1'). When the clock returns to a low state, the slave 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-slave D flip-flop works.

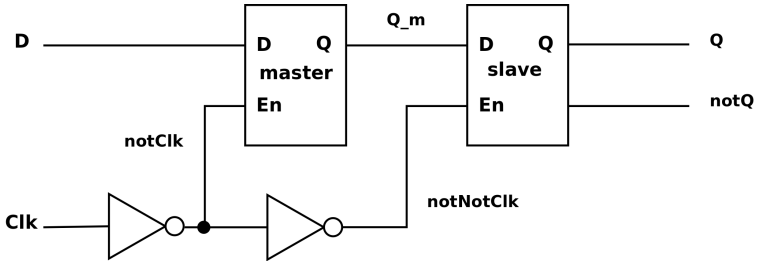


Figure 6: Master-Slave D flip-flop

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.

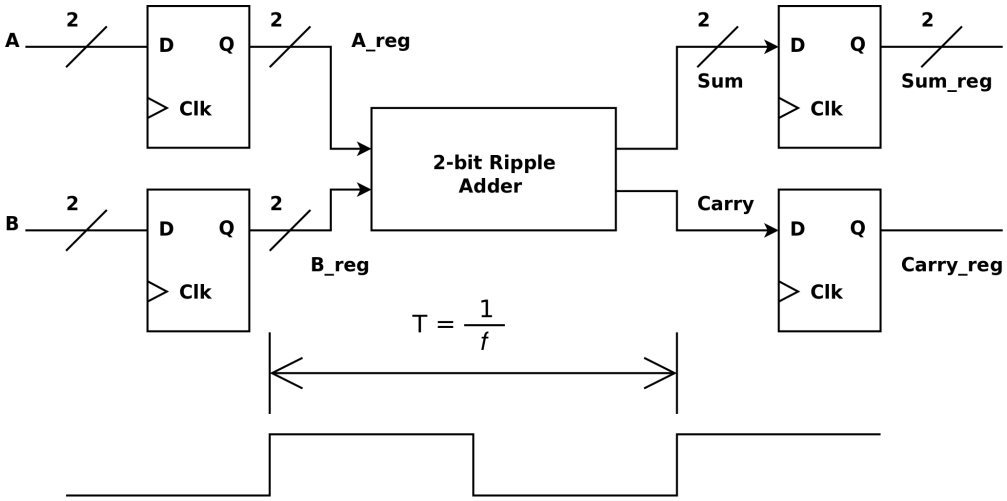
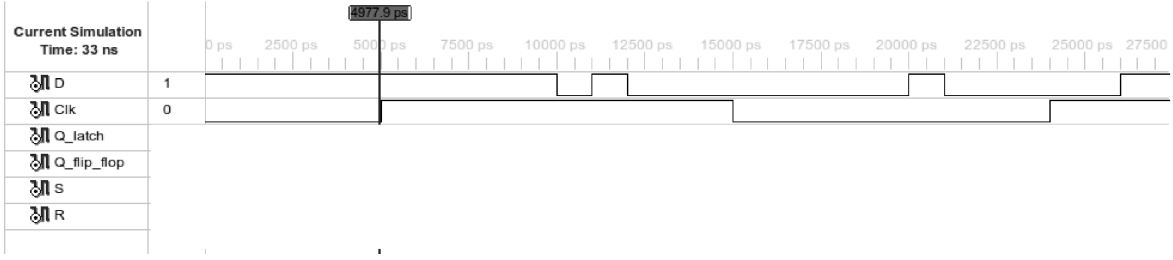


Figure 7: Example Synchronous Logic Circuit

3 Pre-lab

The pre-lab questions below will test you on your knowledge of the material presented above. Please answer the following questions completely:

1. Complete the waveform below given the circuit in Figure 4. For the S and R, assume the D-latch is constructed as shown in Figure 3. You may submit a hand drawn waveform.



2. If the circuit in Figure 7 utilized the 2-bit ripple-carry adder designed in Lab 3, what would be the maximum value of f given that each gate-delay unit is 4 ns? Assume that the flip-flops are ideal and do not add any additional timing constraints to the circuit.

4 Lab Procedure

The following lab exercises serve to reinforce the material discussed in the background section above and in the course lecture.

4.1 Experiment 1

For the first experiment, you will become acquainted with the storage elements discussed in the background section of this lab manual. We will begin by using structural Verilog and the built-in gate-level primitives to describe these storage elements, as well as simulate them in ISE. To do so, we must introduce simulation *delays* into our Verilog code. Furthermore, we will make use of behavior Verilog to describe some of the same storage components so that in future experiments, 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 ISE project and call it "lab8".

   (b) Using the code below as a template, describe the SR-latch at the gate-level using *structural* Verilog. Take note of the delays that have been introduced into the code. When we place a "#2" between the module and instance name in Verilog, the output of the module instance will be delays by 2 time units. Because we set the timescale to 1ns, the simulation delay will be 2ns.

```
1  `timescale 1ns / 1ps //specify 1ns for each delay unit
   `default_nettype none
3
   /* Use structural Verilog and the built-in gate-level
5   * primitives to construct the SR latch described in lab*/

7  module sr_latch(Q, notQ, En, S, R);

9      /*all ports should be wires*/
       output wire Q, notQ;
11     input  wire En, S, R;

13     /*intermediate nets*/
       wire nandSEN, nandREN;
```

```
15
      /*woah!!! what is this #2 thing?!?*/
17    /*it's a delay (simulation only!!!)*/
      nand #2 nand0(Q, nandSEN, notQ); //2ns gate delay
19    //finish things up here...

21 endmodule
```

(c) Copy the "sr_latch_tb.v" file from the course directory into your lab8 directory and use it to test your SR-latch in ISE.

(d) Once your latch passes all of the tests, add the SR-latch internal signals to the waveform and examine the waveform to gain an understanding of how the SR-latch is functioning. Compare the results to the function table (Table 1).

(e) Now, change the 2 unit delays in your code to 4 units and run the test bench again. Explain the results of the simulation.

2. Simulate the D-latch using Verilog in ISE.

(a) Figure 3 illustrates the gate-level construction of a D-latch. Using the module interface below and the code above as a starting point, describe the D-latch in *structural* Verilog. Assume NOT gates also have a delay of 2ns each.

```
module d_latch(Q, notQ, En, D);
```

(b) Copy over the "d_latch_tb.v" test bench file and simulate the operation of your D-latch. Verify that the operation matches that described in 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 for the D flip-flop is supplied below. Within your code, instantiate only two D-latch modules and two inverter gates. Assume the inverter gates have a delay of 2ns each. Do not put any additional delay on the D-latches because the delays are built into the module already.

```
1 `timescale 1ns / 1ps //specify 1ns for each delay unit
  `default_nettype none
3
  /* Use structural Verilog, the built-in gate-level
5  * primitives, and our D-latch to construct the D flip-flop described in lab*/

7 module d_flip_flop(Q, notQ, Clk, D);

9     /*all ports should be wires*/
      output wire Q, notQ; //outputs of slave latch
11    input  wire Clk, D;
```

```
13      /*internal nets*/
        wire notClk, notNotClk;
15      wire Q_m; //output of master latch
        wire notQ_m; //notQ_m will be wired to the d_latch but then left unconnected from there
17
        /*structural level wiring*/
19      //instantiate and wire up the not gates here...

21      //instntiate and wire up the d latches based on schematic in lab

23  endmodule
```

(b) Simulate your D flip-flop using the "d_flip_flop_tb.v" file in the course directory. Add the internal nets within your D flip-flop to the waveform and examine the waveform after restarting the simulation. Do the latches behave as expected? Why or why not?

4. Behavior Verilog can be used to describe the components we have simulated so far. In fact, when developing a synthesizable Verilog module (i.e. something we intend to put on the FPGA), we should only use behavior Verilog to describe storage components. This is due to the fact that delays are part of the *non-synthesizable* Verilog subset, and latches and flip-flops require the delays to function properly. Luckily, describing storage components in behavior Verilog is very easy. The following example show how to create a D-latch with an **always** statement. Please read over the code including the comments which have been provided.

```
1  `timescale 1ns / 1ps //specify 1ns for each delay unit
   `default_nettype none
3
   /*Verilog behavioral model of a D-latch                    *
5   *BTW, if we synthesized this thing for our fancy          *
    *FPGA, the synthesizer would yell at us. We will talk     *
7   *about why later on in the lab                            */

9  module d_latch_behavioral(
       output reg Q, //Q is driven with a behavioral statement!
11     output wire notQ, //driven with a dataflow statement!
       input wire D, En //wires can drive regs
13 );

15     /*describe behavior of D-latch*/
       always@(En or D)
17         if(En) //if En != 1'b0
               Q = D; //transparent!
19         else //this part is actually not needed!
               Q = Q; //if left out, it will be implied
21
       /*so far we have described the output Q but have made no mention *
23      *of what notQ is so here goes...  this should look familiar     */
       assign notQ = ~Q;//regs can drive wires!
```

```
25
endmodule
```

Although the code above is syntactically correct, it is not appropriate for an FPGA due to the fact that the FPGA was designed to implement *synchronous* logic and latches are used create *asynchronous* logic. One way of thinking about it is that the code above falls into a gray area between synthesizable and non-synthesizable logic. The synthesizer will not error out but will produce warnings because we are using latches. Unless you have a deep understanding of what it is you are doing, your logic will probably not function correctly on the FPGA even if it works in simulation. Instead, you should use only flip-flops in your sequential designs and do so as shown below.

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

4 /*Verilog behavioral model of a flip-flop without an enable.*
  *Our FPGA prefers this sort of storage element over the    *
6 *D-latch.                                                  */

8 module d_flip_flop_behavioral(
      output reg Q, //yeah y'all know what this stuff does
10    output wire notQ,
      input wire D,
12    input wire Clk //Clock is our trigger signal. Stuff happens
                     //during transistions of Clk
14 );

16    /*describe behavior of D flip-flop*/
      always@(posedge Clk) //trigger edge is the positive (rising) edge
18        Q <= D; //woah!!! what is this <= thing?!?  it's a non-blocking
                  //assignment statement. Don't worry about it right now!
20
      /*so far we have described the output Q but have made no mention *
22     *of what notQ is so here goes...  this should look familiar     */
      assign notQ = ~Q;//regs can drive wires!
24
endmodule
```

(a) 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 appropriate test bench to ensure proper functionality.

(b) Compare the waveforms you captured from the behavioral Verilog to those captured from the structural Verilog. Are they different? If so, how?

4.2 Experiment 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 in your lab5 directory into your lab8 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);
   ```

   (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 lab8 directory and complete the test bench code based on the hints found within the comments.

   (d) Add the test bench to your ISE 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.

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 idea 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.

   ```
   1 `timescale 1ns / 1ps //specify 1ns for each delay unit
     `default_nettype none
   3
     /*This module could possibly be the first synchronous circuit you've *
   5  *ever written. How exciting?!?
      *You will probably remember this moment for the rest of your life...*/
   ```

```
 7
module adder_synchronous(Carry_reg, Sum_reg, Clk, A, B);
 9
    /*Output ports are regs!!! why? well they need to be able to
11    hold state*/
    output reg Carry_reg;
13  output reg [1:0] Sum_reg;

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

19  /*intermediate nets*/
    reg [1:0] A_reg, B_reg; //will use these as 2-bit registers
21  wire Carry; //need this to connect to the registers described below
    wire [1:0] Sum;
23
    /*instantiate your 2-bit adder here...*/
25


27  /*okay here is where thing get interesting...*/
    /*this behavioral block describes two 2-bit registers*/
29  /*remember registers are nothing more than grouped flip-flops */
    always@(posedge Clk) //the trigger condition is the postive edge of clock!
31    begin //will need this because we will put two statements in here
        A_reg <= A; //we use non-block assigments here because we want
33      B_reg <= B; //these two statements to happen concurrently
      end
35
    /*well that was easy...*/
37  /*let's describe the registers for the result*/
    always@(posedge Clk)
39    begin
        Carry_reg <= Carry; //woah! is this right?!? ooh yea  wires can drive regs
41      Sum_reg   <= Sum;   //these two statements look the same even though the
      end                   //bit width is different
43
    /*we could have actually grouped these all into one always block; however
45    it is nice to clearly divide up the inputs and outputs of the circuit*/

47 endmodule //that's all folks
```

(b) Copy the test bench file, "adder_synchronous_tb.v", from the course directory into your lab8 directory. Add the test bench to your ISE project and simulate it to ensure your synchronous adder circuit passes all of the tests.

(c) Once your synchronous circuit is working properly, open the test bench file and locate the behavioral code that computes the *KnownAnswer* value. It turns out, this succinct description found within the test bench could replace the code we just wrote. Both Verilog descriptions would

synthesize to the same logic. You will be asked to compare these two descriptions at the end of the lab assignment.

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.

5 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

1. Include the source code with comments for **all** modules you simulated. You do **not** have to include test bench code that was provided; however, you must supply the test bench code that you wrote! Code without comments will not be accepted!

2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.

3. Answer all questions throughout the lab manual.

4. Compare the behavioral description of the synchronous adder found in the test bench code with the combination of structural and dataflow Verilog you used in the lab assignment. What are the advantages and disadvantages of each? Which do you prefer and why?

5. Based on the clock period you measured for your synchronous adder, what would be the theoretical maximum clock rate? 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?

6 Important Student Feedback

The last part of 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 section 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?