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

Laboratory Exercise #9
An Introduction to High-Speed Addition

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

1 Introduction

In the previous lab assignments, we saw how the propagation delay through combinational logic has a direct impact on the speed at which we drive our synchronous circuits. We also witnessed the impractical delays associated with ripple-carry addition for larger bit widths. Based on these observations, one can surmise that the ripple-carry adder is not appropriate for high-speed arithmetic units, such as those found in mobile devices and gaming units. As digital circuit designers, it is important that we understand other techniques that exist for binary addition. In this lab assignment, we will introduce a well-known fast-adder circuit for two-operand addition, namely *carry-lookahead* addition. We will design the necessary components using a mixture of dataflow and structural Verilog and then simulate them in Xilinx Vivado. The hands-on experience you will gain working with fast-adder circuits in this lab will server to reinforce the concepts discussed in lecture.

2 Background

Background information necessary for completion of this lab assignment will be presented in the next few subsection. Please note that this lab utilizes the Δ symbol for delay. So for example, gate-delay is represented as Δ_g, while the delay through a ripple-carry adder is represented as $\Delta_{ripple-carry}$. The pre-lab assignment that follows will test your understanding of the background material.

2.1 4-bit Carry-lookahead Addition

Ripple-carry addition suffers from an impractical propagation delay caused by the sequential generation of arithmetic carries. In other words, c_{i+1} is dependent on c_i, which is further dependent on c_{i-1}, etc. The effect of this carry chain is a propagation delay that has a linear dependency on n, the bit width of the adder. Therefore, methods that compute the arithmetic carries in parallel have potential performance benefits over ripple-carry addition.

As the name implies, *carry-lookahead* is one such technique for high-speed addition that computes arithmetic carries in a parallel fashion. To understand how exactly a *carry-lookahead* adder works, consider the addition of two numbers, X and Y, such that x_i is the i^{th} binary digit of X, and y_i is the i^{th} binary digit of Y. The $(i+1)^{th}$ arithmetic carry is c_{i+1} and is computed as follows:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \tag{1}$$

$$= x_i y_i + (x_i + y_i) c_i \tag{2}$$

The effect of simply factoring out c_i from the last two terms in expression (1) is shown in expression (2). Now observe that c_{i+1} is logic '1' if either of the two conditions exists:

1. $x_i y_i$ is logic '1'
2. $x_i + y_i$ is logic '1' and there is a previous carry (i.e. $c_i = 1$)

Therefore, $x_i y_i$ is referred to as *generate* function because when '1', a carry is generated, while $x_i + y_i$ is referred to as the *propagate* function because when '1', it will propagate a carry. In mathematical terms, we see that

$$g_i = x_i y_i \tag{3}$$
$$p_i = x_i + y_i \tag{4}$$
$$c_{i+1} = g_i + p_i c_i \tag{5}$$

Clearly, expressions (3) and (4) do not depend on the carry in the previous bit position and thus, can be generated in parallel. It turns out, we can write expression (5) for the first four carries in such a way that they, too, do not depend on one another, but rather only depend on the input carry, c_0, and the g_i's and p_i's. Examine the expressions below to convince yourself of this.

$$c_1 = g_0 + p_0 c_0 \tag{6}$$
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0 \tag{7}$$
$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \tag{8}$$
$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \tag{9}$$

Although the expression for c_i becomes increasingly complex, the theoretical gate-delay for each of the above expressions, given the g_i's, p_i's, and c_0, is $\Delta_g = 2$. However, the increased complexity is reflected in the number of inputs to each gate (i.e. the gate fan-in) and the number of gates required. Figure 1 illustrates this point with the gate-level schematic for each of the sub-modules within a 4-bit *carry-lookahead* adder. One thing to note is that:

$$p_i = x_i \oplus y_i \tag{10}$$
$$s_i = p_i \oplus c_i \tag{11}$$

In other words, **expression (10) is being used in place of expression (4). It turns out that expression (5) works correctly in either case, but using expression (10) allows the Sum to be computed with expression (11).** Before moving on, let us try to understand how data flows through the 4-bit carry-lookahead adder. To do so, we enumerate through the steps below:

1. Data arrives at the Generate/Propagate Unit, and the g_i's and p_i's are computed in one gate-delay (i.e. $\Delta_g = 1$).

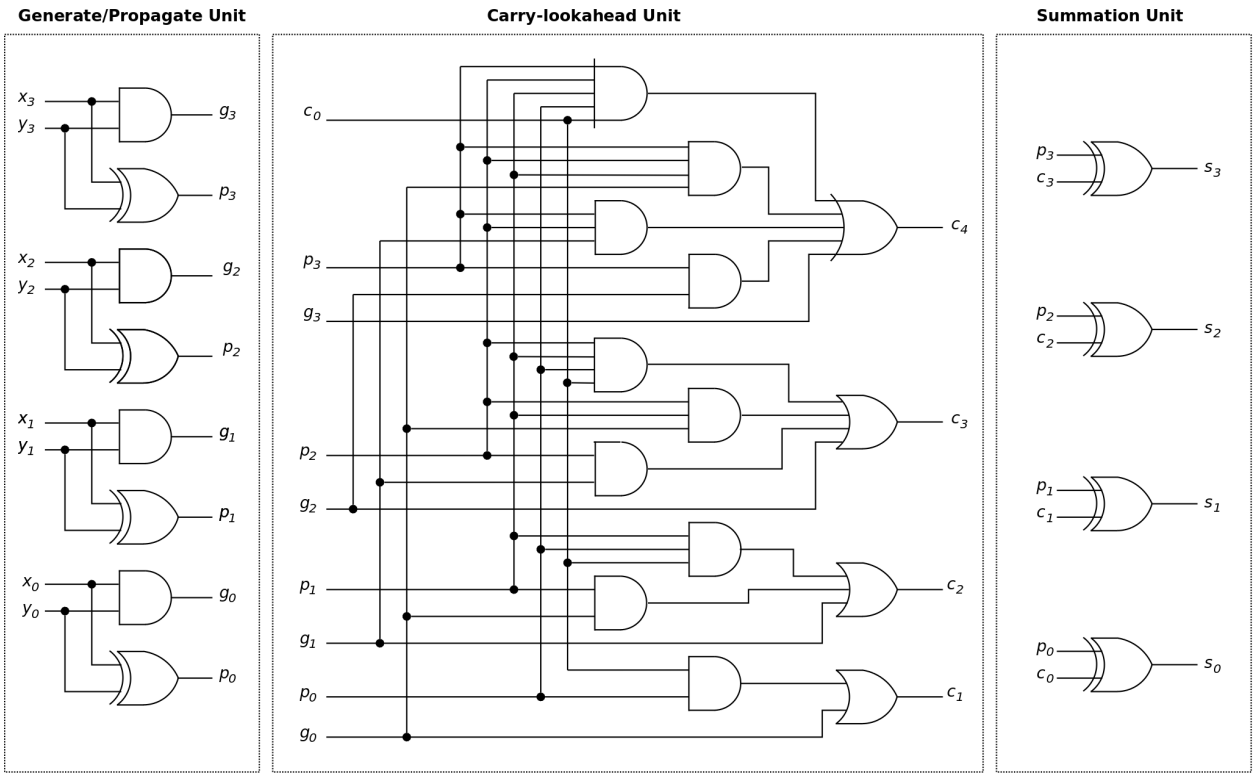

Figure 1: Carry-lookahead Adder

2. The g_i's and p_i's are forwarded to the Carry-Lookahead Unit, which generates all of the carries in two gate-delays, $\Delta_g = 2$.

3. The carries are then fed into the Summation Unit, which computes the sum bits, the s_i's, in one gate-delay $\Delta_g = 1$.

For simplicity, we are assuming that all gates have the same delay time. This assumption may or may not be true depending on the target technology that is being used to implement your logic. However, for the sake of comparison with other addition techniques, this model works well. Summarizing the above steps, we can see that the propagation delay for a 4-bit adder is no longer determined by a carry chain and is only four gate-delays, ($\Delta_g = 4$). The pre-lab assignment will include an exercise which asks you to look at the gate count of a 4-bit Carry-Lookahead Adder.

2.2 Two-Level Carry-lookahead Addition

For practical implementations, fan-in becomes a major concern as $n > 4$. Observe that the Generate/Propagate and Summation units scale nicely with n (i.e. the gate area has a linear dependency on the width of the

adder). However, notice that this is certainly not the case with the Carry-Lookahead Unit. Not only does the gate count rapidly increase, but also the fan-in of each gate increases. Both of these properties contribute to a larger gate area. For $n = 16$, *two-level* carry-lookahead addition can be employed such as Figure 2), where carry generation is broken up into blocks. Each block is m-bits wide, and the number of carry blocks required is given by $\frac{n}{m}$. For the example below, $n = 16$ and $m = 4$.

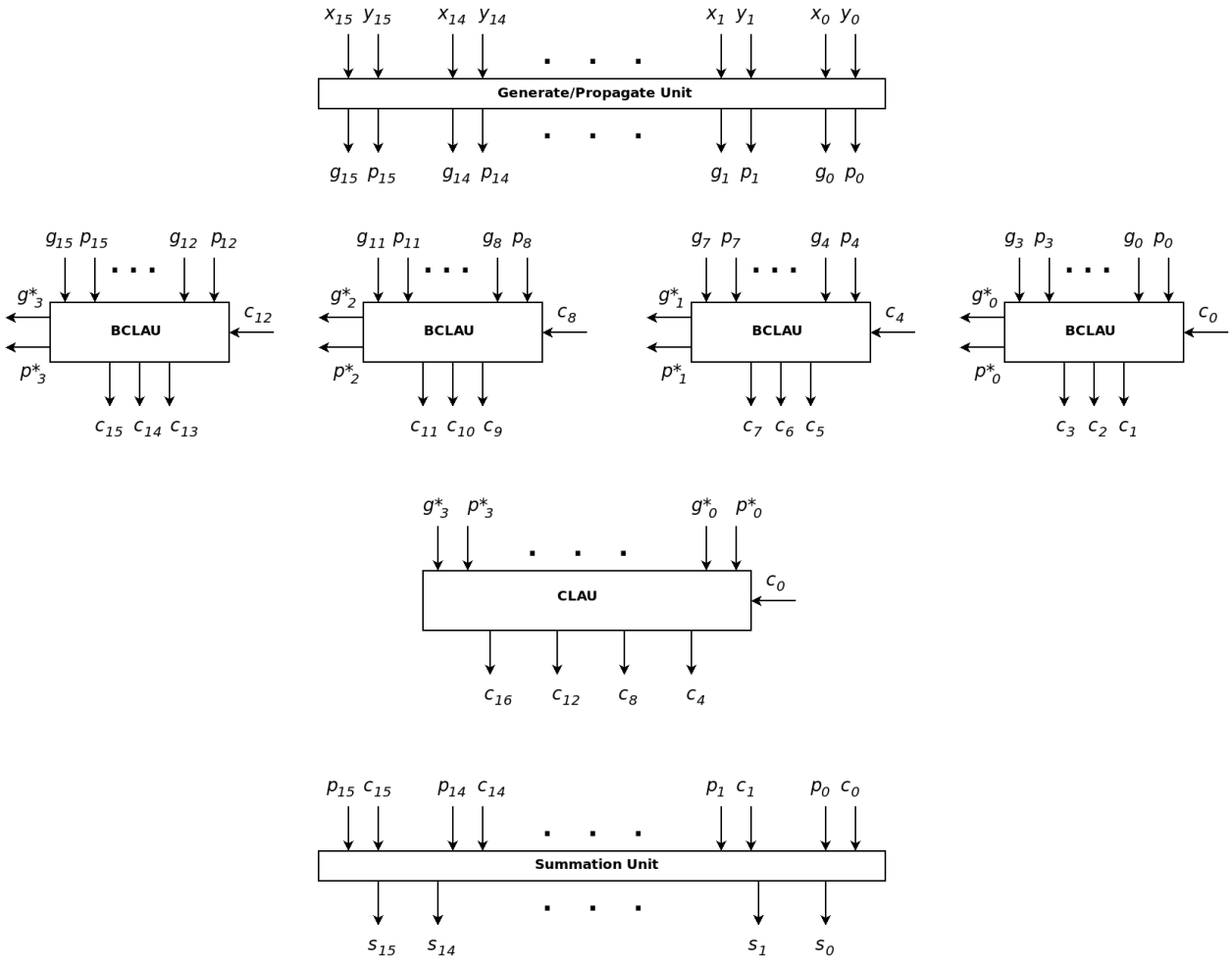

Figure 2: Two-Level Carry-Lookahead Adder

To accomplish such a scheme, some slight modifications can be made to our Carry-Lookahead Unit to create what is commonly referred to as a Block Carry-Lookahead Unit. See Figure 3. Notice the similarity between the Carry-Lookahead Unit (CLAU) in Figure 1 and the Block Carry-Lookahead Unit (BCLAU)

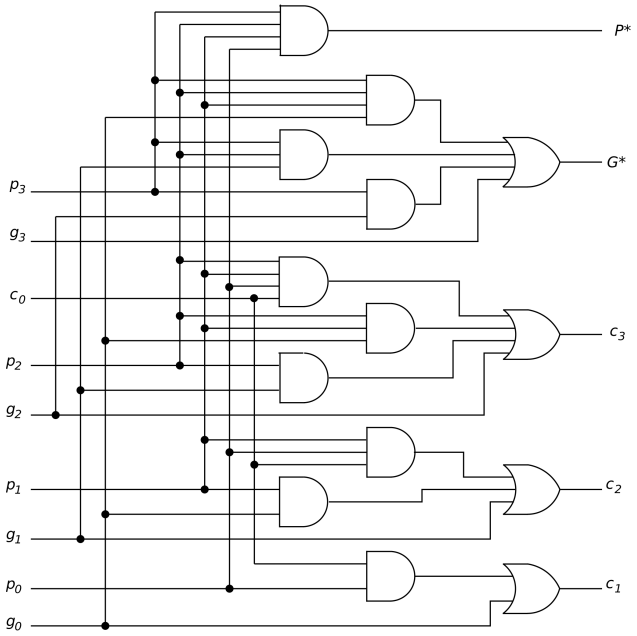

Figure 3: Block Carry-lookahead Unit

in Figure 3. Rather than producing c_4, the BCLAU creates a block-level generate, *G**, and a block-level propagate, *P**, each of which do not depend on the carry input of that particular block. These nets signal whether or not the corresponding block will generate or propagate a carry as a whole. Their expressions are as follows:

$$G* = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \tag{12}$$

$$P* = p_3p_2p_1p_0 \tag{13}$$

*g** and *p** can be fed into the CLAU discussed earlier in order to generate the carry input for each of the BCLAUs, which will then in parallel generate the remaining carries. As with the single-level *carry-lookahead* scheme, once all the carries have been computed, the sum vector can easily be computed in 1 gate-delay. As before, we will summarize the computation steps below:

1. The g_i's and p_i's are computed within the Generate/Propagate in $\Delta_g = 1$.
2. The g_i's and p_i's are forwarded to the BCLAUs, which calculate the block generates, $g*_i$'s, and the block propagates, $p*_i$'s, in $\Delta_g = 2$.
3. The $g*_i$'s and the $p*_i$'s are then used to compute the carry inputs (c_4, c_8, and c_{12}) to the BCLAUs in $\Delta_g = 2$.

4. With the carry inputs, the BCLAUs are able to generate the remaining carries in $\Delta_g = 2$.

5. Finally, with all of the arithmetic carries, the Summation Unit computes the s_i's in $\Delta_g = 1$.

As n increases, we can add more levels to the *carry-lookahead* adder, which creates a $log_4(n)$ delay function. The pre-lab assignment will include an exercise which asks you to look empirically at the gate-delay and gate-count of the 16-bit Carry-Lookahead Adder.

3 Pre-lab

The pre-lab assignment below will ask that you describe some of the components discussed in the background section in Verilog. You do not have to test these modules because that is what we will do in lab! Additionally, some questions are provided to quiz you on your knowledge of the material discussed above. Please answer the questions thoroughly.

1. With dataflow Verilog, describe the Generate/Propagate Unit, the Carry-Lookahead Unit, and the Summation Unit in Figure 1 as separate modules. Do not include delays in your models. We will add them later in the lab experiments. Use the module interfaces below as a guide. Gate-level schematics can be hard to read so you may find expressions (3) through (11) easier to follow.
   Note: If you are unsure of what dataflow Verilog looks like, consult lab 5!

```
/*This module describes the Carry Generate/Propagate*
 *Unit for 4-bit carry-lookahead addition            */
module generate_propagate_unit(G, P, X, Y);

    /*ports are wires as we will use dataflow*/
    output wire [3:0] G, P;
    input wire [3:0] X, Y;


/*This module describes the 4-bit carry-lookahead unit*
 *for a carry-lookahead adder                         */
module carry_lookahead_unit(C, G, P, C0);

    /*ports are wires because we will use dataflow*/
    output wire [4:1] C; //C4, C3, C2, C1
    input wire [3:0] G, P;//generates and propagates
    input wire C0; //input carry


/*This module describes the 4-bit summation unit*
 *for a carry-looahead adder                    */
module summation_unit(S, P, C);

    /*ports are wires because we will use dataflow*/
    output wire [3:0] S; //sum vector
```

```
    input wire [3:0] P, C;//propagate and carry vectors
```

2. Now, use structural Verilog along with the modules you have just created to wire up a 4-bit Carry-Lookahead adder. The module interface you should use is provided below.
   Note: If you are unsure of what structural Verilog looks like, consult lab 5!

```
/*This is the top-level module for a 4-bit*
 *carry-lookahead adder                    */
module carry_lookahead_4bit(Cout, S, X, Y, Cin);

    /*ports are wires as we will use structural*/
    output wire Cout; //C_4 for a 4-bit adder
    output wire [3:0] S; //final 4-bit sum vector
    input wire [3:0] X, Y;//the 4-bit addends
    input wire Cin; //input carry!
```

3. What is the gate-count of your 4-bit *carry-lookahead* adder?

4. The previous problems were concerned with a single-level 4-bit *carry-lookahead* adder. In one of the lab experiments, we will construct a 16-bit, 2-level *carry-lookahead* adder. The following questions will prepare you for this exercise. What is the propagation delay of the 16-bit, *2-level* carry-lookahead adder in Figure 2? Likewise, what is the gate-count?

4 Lab Procedure

The following lab exercises will guide you through the simulation of the circuits discussed in the background section above.

4.1 Experiment Part 1

The goal of the first experiment is to implement and test the 4-bit *carry-lookahead* adder introduced in the background section of the lab. To do so, we will load the Verilog modules you created in the pre-lab into the Vivado environment and simulate their operation.

1. By testing the 4-bit *carry-lookahead* adder, we can ensure that the sub-modules required to implement the 16-bit *carry-lookahead* adder are working properly. The following steps will guide you through the process.

   (a) Create a new project in Vivado called lab9.

   (b) Copy the test bench file, 'cla_4bit_tb.v', from the course directory into your lab9 directory, and add it to your Vivado project.

   (c) Add the 4-bit Carry-Lookahead adder and all of the required sub-modules created in the prelab to your Vivado project and simulate the test bench.

   (d) Ensure all of the tests pass. You do not have to gather a screenshot for this simulation; however, if your design does not work, use the waveform viewer for debugging. X's and Z's in your output waves usually indicate disconnected or misplaced wires.

2. With our 4-bit *carry-lookahead* adder working properly, we can examine its propagation delay. The following steps will guide you through the process.

   (a) Modify your sub-modules to include 2 ns gate-delays. For **assign** statements which describe more than one type of gate, make the delay that of the boolean expression being described.

   (b) Re-simulate the test bench and measure the propagation delay. To do this:

      i. Open the test bench file. Scroll down until you find the for loop.

      ii. Find the two lines in this loop that just read "#10;".

      iii. Change the delay to a low value such as 6 in both lines.

      iv. Re-simulate the test bench. If every test still passes, ensure your delays are added correctly.

      v. Increment these two delay values until all tests pass again. The final delay is your measured delay.

   (c) Once you have found your propagation delay, take a screenshot of the waveform and console output for your lab report.

4.2 Experiment Part 2

For experiment part 2, we will create a *two-level* carry-lookahead unit in order to add 16-bit numbers together.

1. Design a *two-level* carry-lookahead adder from the submodules of the 4-bit *carry-lookahead* adder.

   (a) Using your Carry-Lookahead Unit as a starting point, describe the Block Carry-Lookahead Unit in Figure 3. You may find that the expressions (12) and (13) are easier to read. Do not include gate-delays at the moment. The module interface is as follows:

```
/*This module describes the block carry-lookahead unit*
 *for a 2-level carry-lookahead adder                  */
module block_carry_lookahead_unit(G_star, P_star, C, G, P, C0);

    /*ports are wires because we will use dataflow*/
    output wire G_star, P_star;//block generate and propagate
    output wire [3:1] C; //C3, C2, C1
    input wire [3:0] G, P;//generates and propagates
    input wire C0; //input carry
```

   (b) Expand the Generate/Propagate and Summation Units to 16-bits wide and remove the gate-delays.

   (c) Remove the gate delays from the Carry-Lookahead Unit so that we do not confuse ourselves while debugging. Do not modify anything else in this unit because we will use it as-is.

   (d) Use the template code below to describe the 16-bit, two-level *carry-lookahead* adder depicted in Figure 2.

```
1  `timescale 1ns / 1ps
   `default_nettype none
3
   /*This is the top-level module for a 16-bit*
5   *2-level carry-lookahead adder             */
   module carry_lookahead_16bit(Cout, S, X, Y, Cin);
7
       /*ports are wires as we will use structural*/
9      output wire Cout; //C_16 for a 16-bit adder
       output wire [15:0] S; //final 16-bit sum vector
11     input wire [15:0] X, Y;//the 16-bit addends
       input wire Cin; //input carry!
13
       /*intermediate nets*/
15     wire [16:0] C; //17-bit carry vector
       wire [15:0] P, G; //generate and propagate vectors
17     wire [3:0] P_star, G_star;//block gens and props

19     /*hook up input and output carry*/
       //do something here
21
       /*sub-modules*/
23     //instantiate the generate/propagate unit here

25     /*for the more complicated module instantiations    *
        *you will probably find this form easier to read */
27     block_carry_lookahead_unit BCLAU0(
           /*since we are being explicit with our ports,  *
29          *the order does not matter!                    */
           .G_star (G_star[0]),
31         .P_star (P_star[0]),
           .C (C[3:1]),
33         .G (G[3:0]),
           .P (P[3:0]),
35         .C0 (C[0])
       );//a little more verbose, but much easier to follow!
37
       //instantiate BCLAUs 1-3 here
39
       /*we will use the same form for this one too*/
41     carry_lookahead_unit CLAU(
           /*okay so we will need to use some fancy  *
43          *concatenation syntax to create the carry*
            *vector connected to this module...      */
45         .C ({C[16], C[12], C[8], C[4]}),
           .G (G_star),
47         //two lines are missing here...

49     );
```

```
51      //instantiate the summation unit here

53 endmodule // yee yee
```

2. With the 16-bit *carry-lookahead* adder described in Verilog, we are ready to simulate its operation.
   (a) Copy the test bench file, 'cla_16bit_tb.v', into your lab9 directory and add it to your Vivado project.
   (b) Add the 16-bit Carry-Lookahead adder and all of the required sub-modules to your Vivado project and simulate the test bench.
   (c) Ensure all of the tests pass. You do not have to gather screenshots for this simulation; however, if your design does not work, use the waveform viewer for debugging.
3. With our *two-level* carry-lookahead adder simulating correctly without gate delays, the next step is to add gate delays to simulate the propagation delay of our 16-bit fast adder.
   (a) Modify the 16-bit *carry-lookahead* adder sub-modules to include 2 ns gate-delays. Set the TEST_DELAY **define** within the test bench to the delay you computed for the 2-level *carry-lookahead* adder. Re-simulate the test bench. Now, measure the propagation delay of your adder by decrementing TEST_DELAY and re-simulating until some tests fail.
   (b) If your calculations in the prelab are correct and you correctly added delays to your sub-modules, you should find that the computed delay matches the measure delay. Is this the case?
   (c) Once you have found your propagation delay, take a screenshot of the waveform and console output for your lab report.

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 in lab. You do **not** have to include test bench code. Code without comments will not be accepted!
2. Include the simulation screenshots requested in the above experiments in addition to the corresponding test bench console output.
3. Answer all questions throughout the lab manual.
   - Experiment Part 1, 2.b
   - Experiment Part 2, 3.a

- Experiment Part 2, 3.b

4. How does the gate-count of the 16-bit *carry-lookahead* adder compare to that of a ripple-carry adder of the same size? Give gate counts for both.

5. How does the propagation delay of the 4-bit *carry-lookahead* adder compare to that of a ripple-carry adder of the same size? Give delay values for both.

6. Similarly, how does the propagation delay of the 16-bit *carry-lookahead* adder compare to that of a ripple-carry adder of the same size? Give delay values for both.

7. Compare the delay growth of a ripple-carry adder versus a carry-lookahead adder. (ie. how does the delay increase as we increase the size of each adder?)

6 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?