ECE 385

EXPERIMENT #3

Introduction to SystemVerilog, FPGA, EDA, and 16-bit Adders

I. OBJECTIVE

In this experiment you will transition from breadboard TTL (transistor-transistor logic) elements to RTL (register-transfer level) design on an FPGA using SystemVerilog. You will come to understand the basic syntax and constructs of SystemVerilog, as well as acquire the basic skill required to operate Xilinx Vivado (2022.2), an EDA (Electronic Design Automation) tool for FPGA synthesis and simulation. Vivado's performance analysis and optimization tools will be explored in the process of implementing three types of adders: a carry-ripple adder, a carry-lookahead adder, and a carry-select adder. This performance analysis and optimization will look at the various adders' area, power, and maximum operating frequencies.

II. INTRODUCTION

If you have not done so already, please read the INTRODUCTION TO SYSTEMVERILOG AND TUTORIAL and the INTRODUCTION TO XILINX VIVADO

In addition to the standard synthesis and simulation capability, Xilinx Vivado provides a variety of compiler settings for the designer to tweak for the synthesis and compilation process. Depending on the settings the designer can gear the generated circuit to comply with some predefined constraints or performance criteria, such as the maximum operating frequency of the circuit, the maximum area of the circuit layout, or the maximum static or dynamic power consumed by the circuit.

During the synthesis and compilation process, Vivado collects a variety of analysis data and displays them in the generated Compilation Report. These data are important to the designer in the sense that the designer relies on these data to determine if his or her circuit has met the performance constraints. If the analysis result is far off from the performance criteria, the designer will most likely have to modify the circuit from the designing aspect of the circuit. On the other hand, if the analysis result is just slightly below the performance criteria, then the designer can use many of the built-in tools to optimize the circuit during the compilation process to meet the performance criteria.

Vivado offers a variety of optimization tools, such as built-in timing constraint tools, built-in power constraint tools, and a built-in placement fitter for the area constraint. Many of the optimization steps can be done by simply changing the various synthesis and compilation settings, as suggested by Xilinx Vivado Optimization Advisors, some of the in-depth optimization and analysis can only be done by providing specific constraints to the analyzers.

In most industry practices, circuit implementation on FPGA is usually only a small portion of the entire design, where the circuit on FPGA will interface with external circuits through its inputs and outputs. These external circuits will have their own performance constraints which the FPGA circuit has to follow to be integrated. To incorporate these external constraints into the FPGA design, they are written into constraint files such as the Synopsys Design Constraint (SDC) format as input to the Vivado Analyzers, where the analyzers will then be able to analyze and optimize the circuit based on the provided constraints.

For this lab, we will consider the design of a binary adder. Binary adders are a key component of logic circuits. They are used not only in the arithmetic logic units (ALU) for data processing but are also used in other parts of a logic processor to calculate addresses and signal evaluations. An *N*-bit binary adder takes two binary numbers (*A* and *B*) of size *N* and a carry-in (C_{in}) as inputs, sum up the three values, and produces a sum (*S*) and a carry-out (C_{out}), as shown in Figure 1.

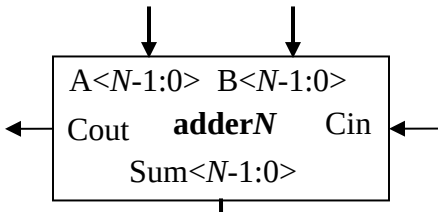

Figure 1 - N-bit Binary Adder Block Diagram

Among the many different binary adder designs, the most straightforward one is the Carry-Ripple Adder (CRA). It is constructed using *N* full-adders. A full-adder is a single-bit version of the binary adder, where three binary bits (*A*, *B* and C_{in}) are inputted through a set of logic gates to produce a single-bit sum (*S*) and a single-bit carry-out (C_{out}), as shown in Figure 2. The *N* full-adders are then linked together in series through the carry bits, forming an *N*-bit binary adder. When the binary inputs are provided, the full-adder of the least significant bit (LSB) will produce a sum (S_0) and a carry-out (C_1). The carry-out is fed to the carry-in of the second full-adder, which then produces a second sum (S_1) and a second carry-out (C_2). The process ripples through all *N* bits of the adder as shown in Figure 3, and settles when the full-adder of the most significant bit (MSB) outputs its sum (S_{N-1}) and carry-out (C_{out}).

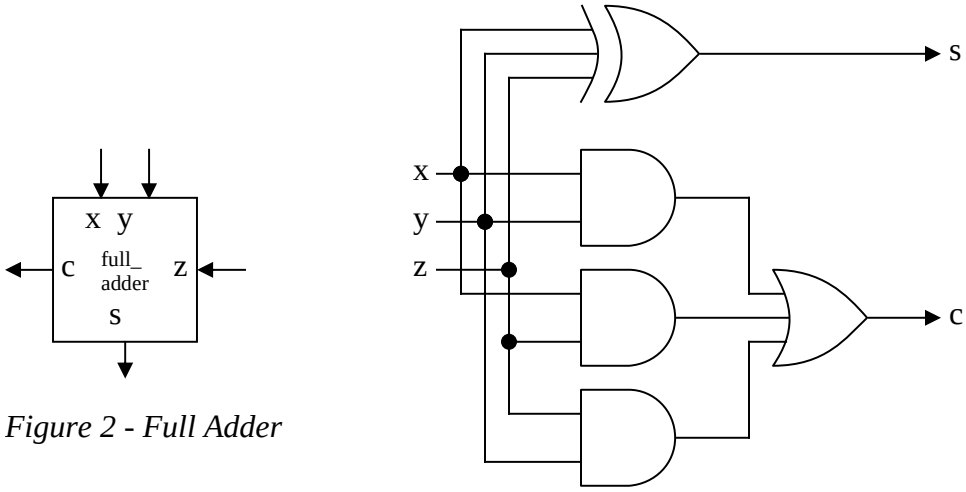

Figure 2 - Full Adder

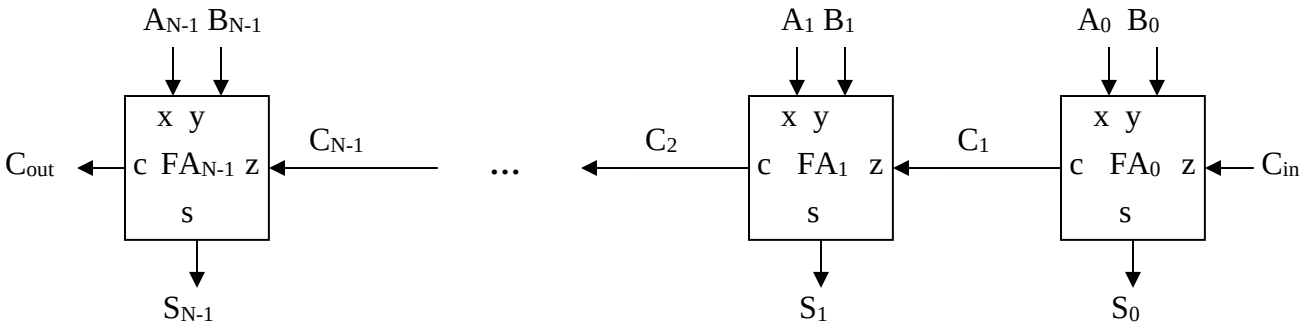

Figure 3 - n-bit Carry Ripple Adder Chain

The CRA is simple in design and straightforward to implement, but the long computation time is its drawback. Every full-adder must wait for their lower-bit neighbor to produce a carry-out before it can correctly compute its sum and carry-out. This means that the propagation delay of the CRA increases with *N*. If one wishes to reduce the computation time, it is apparent that the computation of the carry-out bits must be somehow parallelized. And this is precisely how a carry-lookahead adder operates.

Instead of waiting on the actual carry-in values, Carry-Lookahead Adder (CLA) uses the concept of *generating* (G) and *propagating* (P) logic. The concept is that every bit of the CLA makes predictions using its immediate available inputs (*A* and *B*), and predicts what its carry-out would be for any value of its carry-in. A carry-out is *generated* (G) if and only if both available inputs (*A* and *B*) are 1, regardless of the carry-in. The equation is $G(A,B) = A \cdot B$. On the other hand, a carry-out has the possibility of being *propagated* (P) if either *A* or *B* is 1, which is written as $P(A,B) = A \oplus B$. With P and G defined, the Boolean expression for the carry-out C_{i+1}

giving a potential C_i is then $C_{i+1} = G_i + (P_i \cdot C_i)$. Notice that C_{i+1} can be expressed in terms of C_i which in turn can be expressed in terms of C_{i-1}. However, if C_{i+1} still depends on C_i, it will behave like a ripple adder without giving any gain in speed. Therefore, to avoid the slow rippling of the carry bits, the expression of C_{i+1} should be expanded and computed directly from P_i s, G_i s. For example,

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$
$$...$$

In this way, the computation time of the CLA is much faster than that of the CRA, resulting in a higher operating frequency. The downside of the CLA is its additional logic gates, which increases both the area and power consumption of the adder.

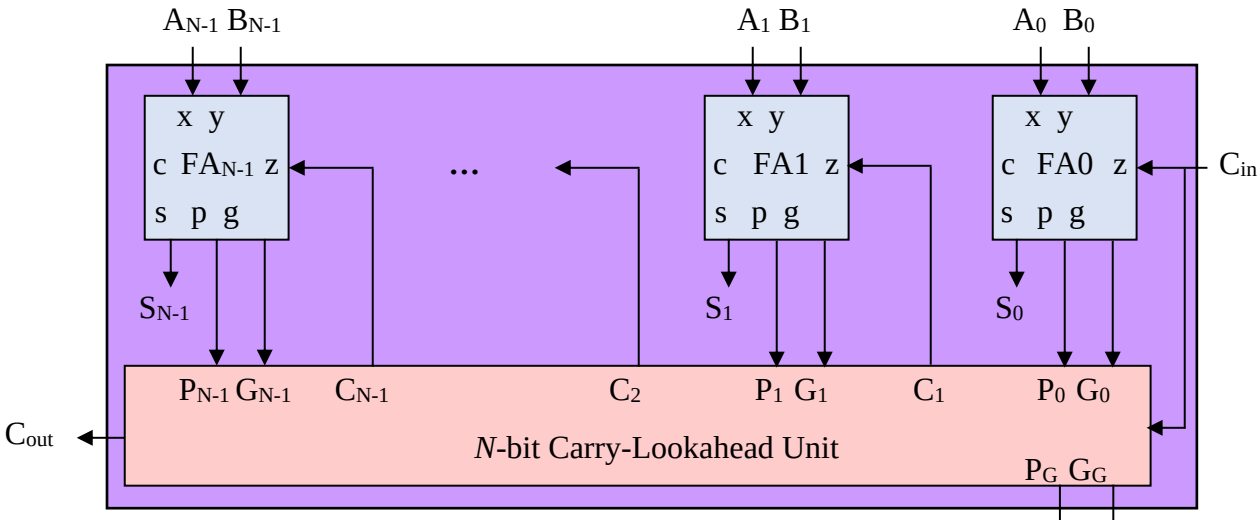

Figure 4 - n-bit Carry Lookahead Adder

To build an arbitrarily long *N*-bit CLA, one might be tempted to directly follow the above 'flat' approach. However, from the explicit expansion of C_i, you can find that the number of gates involved for an increasing *N* will soon grow too large for the CLA to be practical. And thus, it is a common practice to first construct 4-bit CLAs, then use them to create a larger CLA in a hierarchical fashion. In this lab, the CLA should be implemented in 4x4-bit instead of 16-bit.

In the 4x4-bit hierarchical CLA design, the 16-bit inputs *A* and *B* are divided into groups of 4 bits. First, each group of 4 bits go through a 4-bit CLA, which is illustrated by Figure 4 with *N*=4. Note that the 4-bit CLA generates two additional output signals, the group propagate (P_G) and the group generate (G_G), with their logics being:

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

We will denote the P_Gs and G_Gs from these four 4-bit CLAs as P_{G0}, P_{G4}, P_{G8}, P_{G12}, and G_{G0}, G_{G4}, G_{G8}, G_{G12} from this point on.

Next, a tempting design is to cascade the four 4-bit CLAs by connecting the C_{out} from the previous 4-bit CLA to the C_{in} of the next 4-bit CLA, but in this way we will be trapped by the slow rippling of these carry bits again. Therefore, instead of using the C_{out} from the previous 4-bit CLA, we should generate the C_{in}s of the 4-bit CLAs using the P_Gs and G_Gs, as shown by the formulas below,

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$
...

Does this look familiar to you? Observe that this is the same as how we generated the carry bits within a 4-bit CLA. Therefore, we can directly take a copy of the 4-bit Carry-Lookahead Unit (CLU, red block in Figure 4) in the 4-bit CLA, but instead of the inputs coming from full adders, this time the inputs are the P_Gs and G_Gs from the 4-bit CLAs at the upper level. Figure 5 illustrates the resulting 4x4-bit hierarchical CLA.

This explains why this design is called *hierarchical*. If we add another layer to the hierarchy and use four 4x4-bit hierarchical CLAs and another 4-bit CLU, we can make a 4x4x4-bit hierarchical CLA, namely a 64-bit adder, without any issue of the slow rippling of the carry bits!

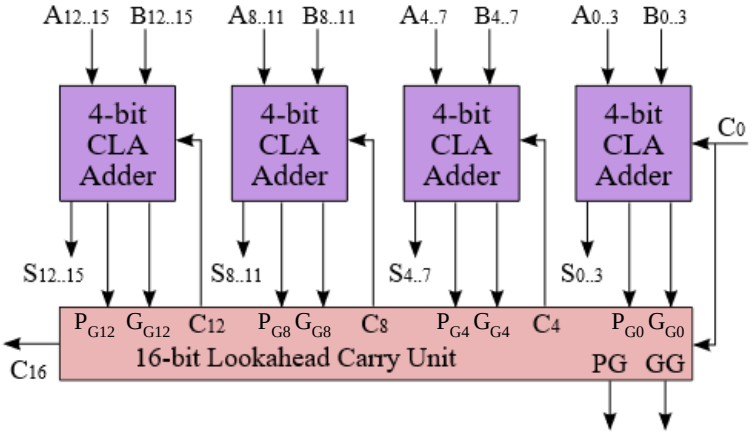

Figure 5 - Hierarchical Carry-Lookahead Adder

Carry-Select Adder (CSA) features another way to speed up the carry computation. It consists of two full adders (or CRAs if multiple bits are grouped) and a multiplexor. One adder computes the sum and carry-out based on the assumption that the carry-in is 0, and the other assumes that the carry-in is 1. In this way, both possible outcomes are pre-computed. Once the real carry-in arrives, the corresponding sum and carry-out is selected to be delivered to the next stage. By paying the price of almost twice the numbers of adders, we gain some speedup *(how exactly do we gain this speedup – we will discuss this in lecture, but you should make sure you understand and explain in your own words for your lab report!)*

In this lab, you are going to design a 16-bit CSA with 4x4-bit hierarchical structure as illustrated by Figure 5. For each group of 4-bit inputs, we use two CRAs to calculate two versions of the results, one with carry-in bit assumed to be 0 and the other to be 1. Note that the lowest significant group requires only one CRA, since its carry-in bit is directly available. Therefore, eventually the 16-bit CSA will contain seven 4-bit CRAs.

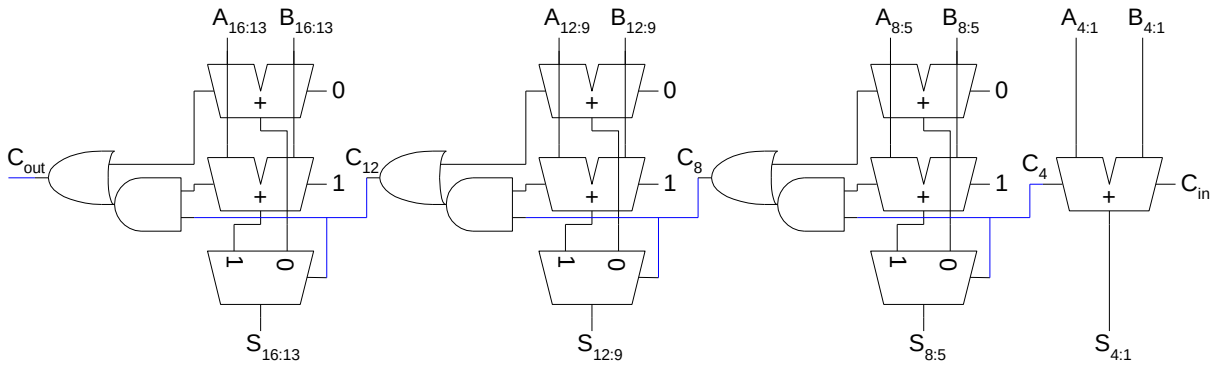

Figure 5: 16-bit Carry-Select Adder Block Diagram

Your circuits should have the following inputs and outputs:

Inputs

clk, reset, run_i – logic
sw_in – logic [15:0]

Outputs

sign_LED – logic
hex_gridA – logic [3:0]
hex_segA – logic [7:0]
hex_gridB – logic [3:0]
hex_segB – logic [7:0]

Internal Registers

In – logic [16:0] holds either switch or the sum
Out – logic [16:0] holds output of one operation
S– logic [16:0] holds the adder sum result

Sw_in[16:0] should come from on-board switches and its value should be displayed on hex grid A. There are only two buttons, reset and run_i. The sign_LED should be displayed on LED[0] to indicate overflow. When run_i is pressed, S[15:0] and sign_LED should be updated with the result of adding sw_in[15:0] (*A*) and the updated out[15:0] (*B*). The new sum result will be written into out[15:0]. As a result, each press on Run_i accumulates the switch value to the previous sum result. Reset should clear all the registers. To achieve optimal speed and resource usage balance, the CLA and CSA will also need to be built in a hierarchical fashion. In this lab, they should be implemented in 4x4-bit instead of 16-bit. Also, for this lab, Cin (carry-in) may be assumed to be 0.

Upon pressing the 'run_i' button, the run_once circuit will load the resulting sum (A+B) into the 16-bit out register to display. The load and run operation will be executed only once when the run_i button is pressed each time, respectively. The circuit should be able to run multiple times without resetting the circuit before each operation.

III. PRE-LAB

Design, document, and implement a 16-bit carry-ripple adder, a 16-bit carry-lookahead adder, and a 16-bit carry-select adder in SystemVerilog. Use the provided code (from the website) as a testing framework.

Make sure you are ready to demo with your code for the 3, 16-bit adders with a project ready to synthesize and test on the FPGA board and be prepared to show your TA each adder's code to verify they are indeed performing according to design. Note that you will have to comment out and uncomment each adder's code to demo each adder independently, as we do not want the results from one adder to affect the overall FPGA results. In addition, make sure you **remove** any debug cores you may have used to speed up the synthesis process and ensure that your timing results are valid.

For submitting your code online, make sure to put **only the .SV** files into a .zip and upload the file in the form of ece385-lab3-net_id1-netid_2.zip. Note that you will need to zip and include the entire <project_name>.srcs directory from your Vivado project, as it will include both imported and newly created files.

Demo Points Breakdown:

Consult the course webpage for the exact breakdown of the demo points.

IV. LAB

Follow the Lab 3 demo information on the course website. The pin assignment table is included below, although a .xdc file is also provided which you may use. Note that you may export/import the pin assignments for use in future labs.

Pin Assignment Table

Port Name	IO Standard	Location	Comments
sw_i[0]	LVCMOS25	G1	On-board slider switch (SW0)
sw_i [1]	LVCMOS25	F2	On-board slider switch (SW1)
sw_i [2]	LVCMOS25	F1	On-board slider switch (SW2)
sw_i [3]	LVCMOS25	E2	On-board slider switch (SW3)
sw_i [4]	LVCMOS25	E1	On-board slider switch (SW4)
sw_i [5]	LVCMOS25	D2	On-board slider switch (SW5)
sw_i [6]	LVCMOS25	D1	On-board slider switch (SW6)
sw_i [7]	LVCMOS25	C2	On-board slider switch (SW7)
sw_i [8]	LVCMOS25	B2	On-board slider switch (SW8)
sw_i [9]	LVCMOS25	A4	On-board slider switch (SW9)
sw_i [10]	LVCMOS25	A5	On-board slider switch (SW10)
sw_i [11]	LVCMOS25	A6	On-board slider switch (SW11)
sw_i [12]	LVCMOS25	C7	On-board slider switch (SW12)
sw_i [13]	LVCMOS25	A7	On-board slider switch (SW13)
sw_i [14]	LVCMOS25	B7	On-board slider switch (SW14)
sw_i [15]	LVCMOS25	A8	On-board slider switch (SW15)
sign_LED	LVCMOS33	C13	On-board LED (LED0)
hex_gridA[0]	LVCMOS25	G6	On-Board eight-segment display grid
hex_gridA[1]	LVCMOS25	H6	On-Board eight-segment display grid
hex_gridA[2]	LVCMOS25	C3	On-Board eight-segment display grid
hex_gridA[3]	LVCMOS25	B3	On-Board eight-segment display grid
hex_segA[0]	LVCMOS25	E6	On-Board eight-segment display segment
hex_segA[1]	LVCMOS25	B4	On-Board eight-segment display segment
hex_segA[2]	LVCMOS25	D5	On-Board eight-segment display segment
hex_segA[3]	LVCMOS25	C5	On-Board eight-segment display segment
hex_segA[4]	LVCMOS25	D7	On-Board eight-segment display segment
hex_segA[5]	LVCMOS25	D6	On-Board eight-segment display segment
hex_segA[6]	LVCMOS25	C4	On-Board eight-segment display segment
hex_segA[7]	LVCMOS25	B5	On-Board eight-segment display segment
hex_gridB[0]	LVCMOS25	E4	On-Board eight-segment display grid
hex_gridB[1]	LVCMOS25	E3	On-Board eight-segment display grid
hex_gridB[2]	LVCMOS25	F5	On-Board eight-segment display grid
hex_gridB[3]	LVCMOS25	H5	On-Board eight-segment display grid
hex_segB[0]	LVCMOS25	F3	On-Board eight-segment display segment

hex_segB[1]	LVCMOS25	G5	On-Board eight-segment display segment
hex_segB[2]	LVCMOS25	J3	On-Board eight-segment display segment
hex_segB[3]	LVCMOS25	H4	On-Board eight-segment display segment
hex_segB[4]	LVCMOS25	F4	On-Board eight-segment display segment
hex_segB[5]	LVCMOS25	H3	On-Board eight-segment display segment
hex_segB[6]	LVCMOS25	E5	On-Board eight-segment display segment
hex_segB[7]	LVCMOS25	J4	On-Board eight-segment display segment
clk	LVCMOS33	N15	50 MHz Clock from the on-board oscillators
reset	LVCMOS25	J1	On-Board Push Button (BTN1)
run_i	LVCMOS25	J2	On-Board Push Button (BTN0)

V. POST-LAB

1. Document design analysis for the three adders in the table below. Plot out the data from the table for comparison studies. Normalize the data across the three adders with the carry-ripple adder. When normalizing, choose data from one the carry-ripple adder as the baseline, and then divide the other two with the baseline number. Suppose you got 20 from carry-ripple, 21 from carry-select, and 23 from carry-lookahead, the number after normalization becomes 20/20=1.0, 21/20=1.05, 23/20=1.15, respectively. The resulting plot should resemble the one below (the plot below does not use real data). Note that to compute the frequency, Vivado reports something called “Worst Negative Slack (WNS)”. Since the provided clock on the Urbana board operates at 100 MHz by default, the clock period is 10 ns (specified in the XDC file). Slack is defined as how much margin is left in the design before the propagation delay of the critical path no longer satisfies the clock period (minus some inherent delays within the flip flops and some clock skew, which we will discuss in lecture). Therefore, the more slack (or margin) there is in the design, the faster it can run. Use the WNS value you got from the timing report (refer to IVT) to compute the maximum frequency. Explain how you computed the frequency from the WNS value in your report.

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT			
Frequency			
Total Power			

3.10

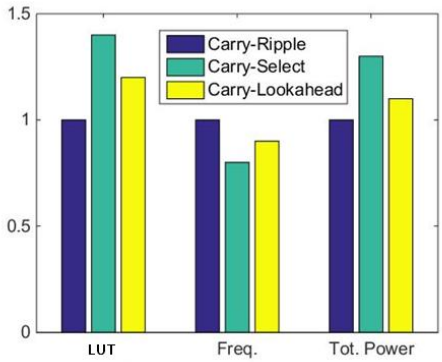

Figure 6 - Example Comparison Graph

Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot make sense? Are they complying with the theoretical design expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder? Which design consumes more power than the other as you expected, why?

2. In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)

3. In addition to the design statistics shown in graph form above, you will need to fill in the remaining columns for each adder. This is required for every SystemVerilog circuit in future (whereas the graph from 1 is only required for this lab). Note that some of the information is repeated in the graph. In addition, some of these values will be 0 if you did not use those resources for this lab.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

4. Do a critical path analysis using the Vivado tools. Specifically, you should show the critical path for each adder. You should watch the Xilinx provided **Vivado Timing Closure**

Techniques video (on Canvas) that explains how to get this information (around 8-minutes in) or Prof. Cheng's explanation in the Friday Q/A. Compare the critical path as given by Vivado to the theoretical critical path, which we discussed in class. For example, did the critical path for the Vivado carry-ripple adder consist of the carry chain through each full adder module? Why or why not? Include the screenshots of the critical path for each adder alongside your theoretical analysis and annotations. Hint: some of the critical paths may look a little strange, but keep in mind the FPGA does not use discrete gates (AND/OR) to implement combinational logic.

VI. REPORT

Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
    a. Summarize the high-level function performed by the three adders.
2. Adders
    a. Ripple Carry Adder
        i. Written description of the architecture of the adder
        ii. Block diagram.
    b. Carry Lookahead Adder
        i. Written description of the architecture of the adder
        ii. Describe how the P and G logic are used.
        iii. Describe how you created the hierarchical 4x4 adder.
        iv. Block diagram
            1. Block diagram inside a single CLA (4-bits)
            2. Block diagram of how each CLA was chained together.
    c. Carry Select Adder
        i. Written description of the architecture of the adder
        ii. Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later. Make sure you understand this!
        iii. Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic.
    d. Written description of all .SV modules (including the provided modules), see Appendix I of the previous lab for the specific formatting. Note that if modules

are repeated from previous labs, you may also use the same module descriptions from your previous reports.

e. Describe at a high level the area, complexity, and performance tradeoffs between the adders.
f. Document the performance of each adder by creating the graph as specified in post- lab part 1.
g. Table for all the remaining performance metrics (post-lab part 3).
h. Annotated simulation trace. You may include just a single annotated simulation trace as all the RTL simulation does not include any gate delays. Note that this will require writing a simple test bench.
i. Perform a critical path analysis and compare this to the theoretical understanding of each adder, this is e.g., post-lab part 4. Include a screen shot of each critical path alongside your explanation.

3. Answers to the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.
4. Conclusion
    a. Describe any bugs and countermeasures taken during this lab.
    b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.
    c. Any additional summary you want to include.