ECE 385

EXPERIMENT #4

Introduction to SystemVerilog, FPGA, CAD, 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 Quartus Prime, a CAD tool for FPGA synthesis and simulation. Quartus Prime'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

Please read the INTRODUCTION TO SYSTEMVERILOG AND TUTORIAL (IST. 1-26) and the INTRODUCTION TO QUARTUS PRIME (IQT. 1-40).

In addition to the standard synthesis and simulation capability, Quartus Prime 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, Quartus Prime collects a variety of analysis data and display 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.

Quartus Prime offers a variety of optimization tools, such as TimeQuest Timing Analyzer for the timing constraint, PowerPlay Power Analyzer for the power constraint, 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 the Quartus Prime 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 in order 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 Quartus Prime Analyzers, where the analyzers will then be able to analyze and optimize the circuit based on the provided constraints.

To read more about the optimization process in Quartus Prime, please refer to Section III in Volume 2 of the Quartus Prime Standard Edition Handbook (currently v18.0, accessible on the Intel FPGA website) Chapter 10 gives a design optimization overview, Chapter 12-14 discuss timing, power and area optimization, respectively.

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.

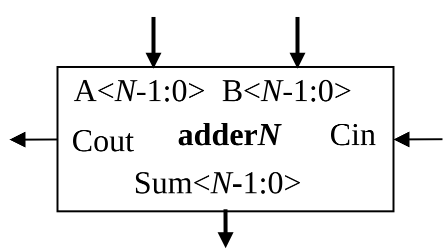


Figure 1: *N*-bit Binary Adder Block Diagram

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}).

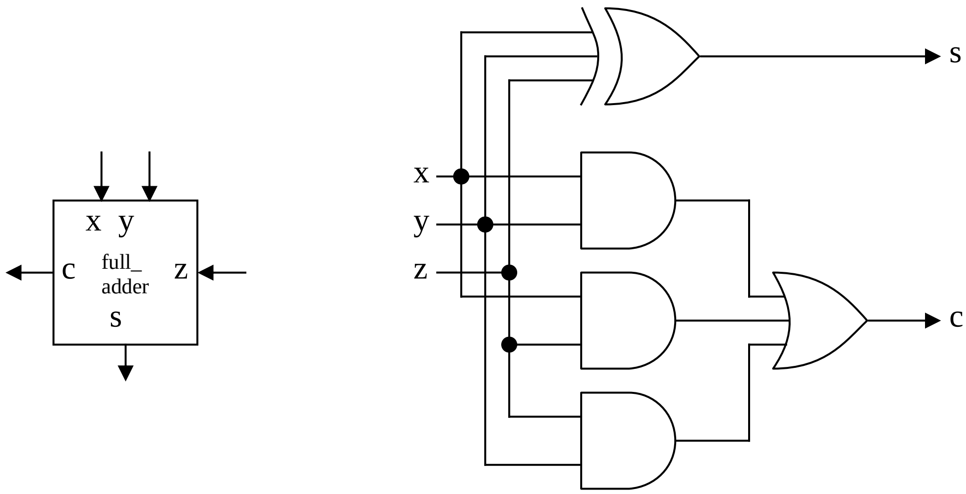


Figure 2: Full-Adder Block Diagram

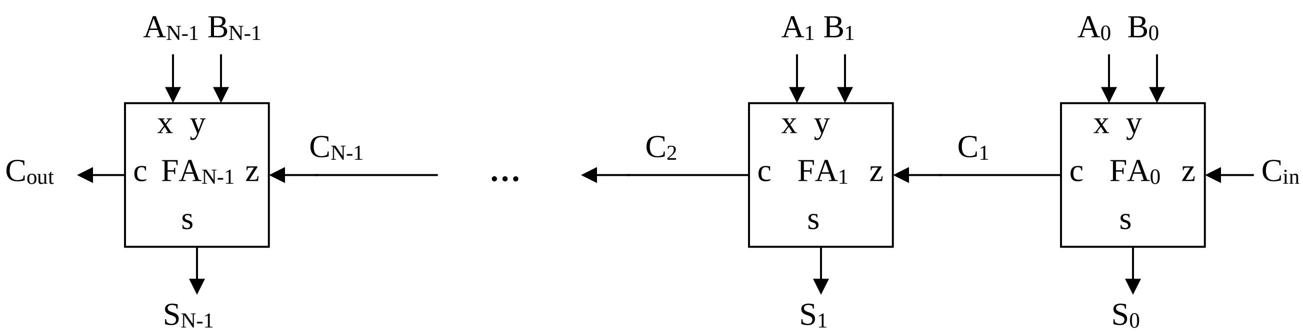


Figure 3: *N*-bit Carry-Ripple Adder Block Diagram

The CRA is simple in the design and straightforward to implement, but the long computation time is its drawback. Every full-adder has to 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 has to 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 still depends on C_{i-1}, 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 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.

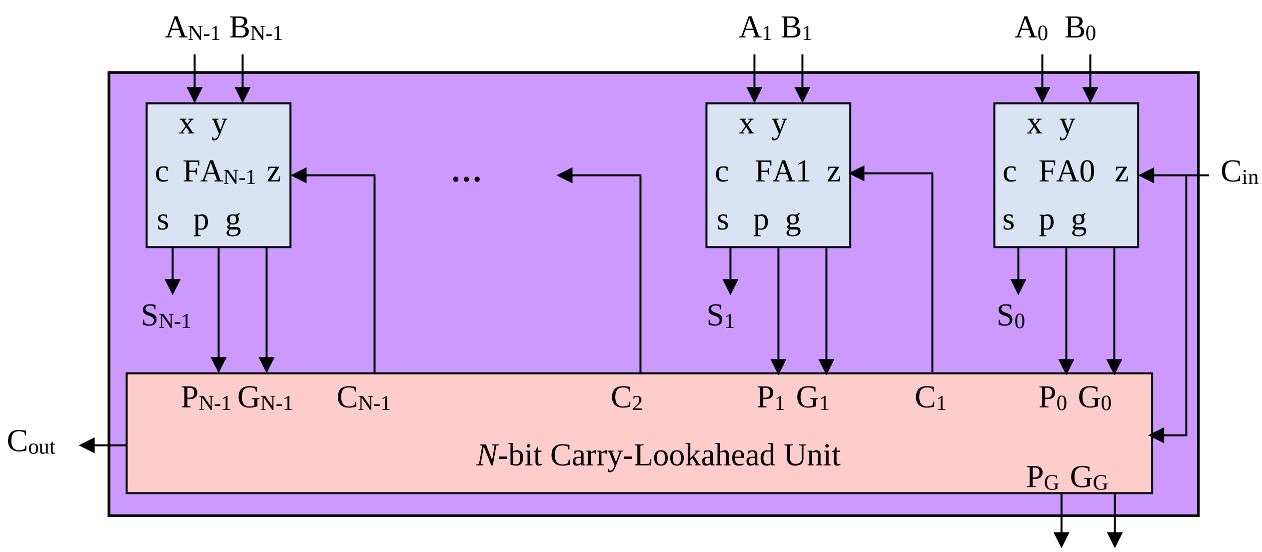


Figure 4: *N*-bit Carry-Lookahead Adder Block Diagram

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!

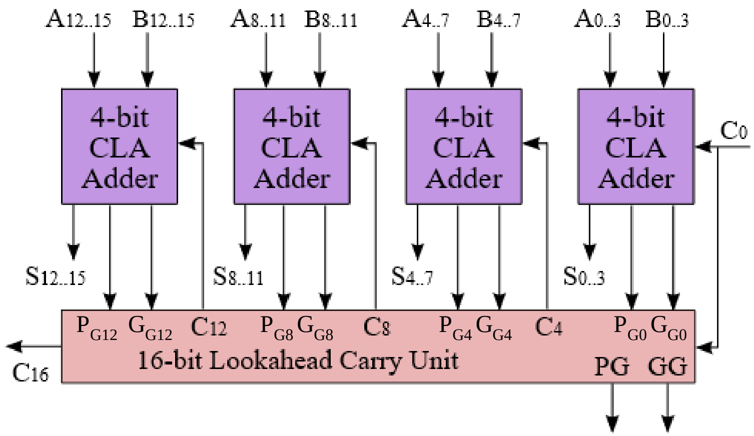


Figure 5: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

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.

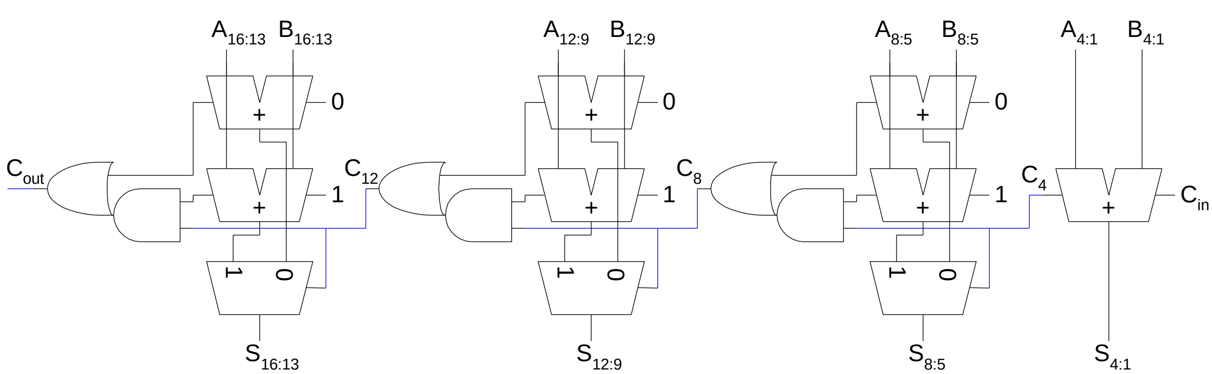


Figure 5: 16-bit Carry-Select Adder Block Diagram

<u>Your circuits should have the following inputs and outputs:</u>

Inputs

Clk, Reset, Load_B, Run – logic

SW – logic [15:0]

Outputs

CO – logic

Sum – logic[15:0]

Ahex0, Ahex1, Ahex2, Ahex3, Bhex0, Bhex1, Bhex2, Bhex3 – logic [6:0]

Internal Registers

A – logic [15:0]

B – logic [15:0]

SW[15:0] should come from on-board switches and its value should be displayed on Ahex0, Ahex1, Ahex2, and Ahex3 as a four-digit hex number. When Load_B is pressed, the registers B[15:0] should load the values of SW[15:0] to serve as *B*, one of the numbers to be added, and B[15:0] should be displayed on Bhex0, Bhex1, Bhex2, and Bhex3. At other times, the registers A[15:0] constantly load the values of SW[15:0] which serve as *A*, the other number to be added. The value of Sum[15:0] should be displayed on red LEDs (LEDR[15:0]), and CO should be displayed on LEDG[8] to indicate overflow. When Run is pressed, Sum[15:0] and CO should be updated with the result of adding SW[15:0] (*A*) and the old B[15:0] (*B*). Reset should clear all the registers. To achieve optimal speed, the CSA will also need to be built in a hierarchical fashion. In this lab, the CSA should be implemented in 4x4-bit instead of 16-bit.

A test platform is required to demo your adders as there are not enough switches on the DE2-115 board. This platform is provided in the included Lab 4 files on the website, and it should be clear where to place your code for the three types of adders you will design. Registers A and B store the operands to be added, depending on whether Load_B is pressed (register A is continuously loaded from the switches on every cycle). Upon pressing the 'Run' button, the state machine will load the resulting sum (A+B) into a 16-bit output register to display. The load and run operation will be executed only once when the Load_B or run button is pressed each time, respectively. The circuit should be able to run multiple times without resetting the circuit before each operation.

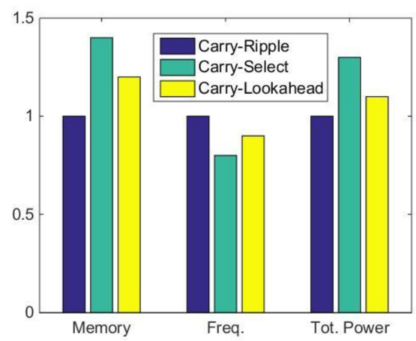
III. <u>PRE-LAB</u>

A. Complete the bit-serial logic processor exercise from the Introduction to SystemVerilog and Tutorial (IQT. 1-40). Include a copy of the generated diagram from Quartus of the 8-bit logic processor and the simulation waveform (with annotations) in your Lab 4 lab report.

B. 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.

C. 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. Say, you got 20 from carry-ripple, 21 from carry-select, and 23 from carry-lookahead, the numbers 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).

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)			
Frequency			
Total Power			



<u>You will need to bring the following to the lab:</u>

1. Your code for the 8-bit processor in a Quartus Prime project, ready to simulate in ModelSim. You can bring the code to the lab using a USB storage device, FTP, or any other method.

2. Your code for the 3, 16-bit adders with a project ready to synthesize and test on the FPGA board, be prepared to show your TA each adder's code to verify they are indeed performing according to design.

3. A block diagram for the 8-bit processor (or a project file which will generate the block diagram) to verify that you have completed the tutorial.

Demo Points Breakdown:

1.0 point: Functional simulation completed successfully for the 8-bit serial processor (annotations necessary)

1.0 point: RTL block diagram of the 8-bit logic processor extended from 4-bits. This can be automatically generated using Quartus.

1.0 point: Correct operation of the Carry-Ripple Adder on the DE2 board

1.0 point: Correct operation of the Carry-Lookahead Adder on the DE2 board using a 4x4 **hierarchical design** (TA's will look at code)

1.0 point: Correct operation of the Carry-Select Adder on the DE2 board using a **4x4 hierarchical design** (TA's will look at code)

IV. LAB

Follow the Lab 4 demo information on the course website.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Run	PIN_R24	On-Board Push Button (KEY3)
LoadB	PIN_M21	On-Board Push Button (KEY1)
Reset	PIN_M23	On-Board Push Button (KEY0)
SW[0]	PIN_AB28	On-board slider switch (SW0)
SW[1]	PIN_AC28	On-board slider switch (SW1)
SW[2]	PIN_AC27	On-board slider switch (SW2)
SW[3]	PIN_AD27	On-board slider switch (SW3)
SW[4]	PIN_AB27	On-board slider switch (SW4)
SW[5]	PIN_AC26	On-board slider switch (SW5)
SW[6]	PIN_AD26	On-board slider switch (SW6)
SW[7]	PIN_AB26	On-board slider switch (SW7)
SW[8]	PIN_AC25	On-board slider switch (SW8)
SW[9]	PIN_AB25	On-board slider switch (SW9)
SW[10]	PIN_AC24	On-board slider switch (SW10)
SW[11]	PIN_AB24	On-board slider switch (SW11)
SW[12]	PIN_AB23	On-board slider switch (SW12)
SW[13]	PIN_AA24	On-board slider switch (SW13)
SW[14]	PIN_AA23	On-board slider switch (SW14)
SW[15]	PIN_AA22	On-board slider switch (SW15)
Sum[0]	PIN_G19	On-Board LED (LEDR0)
Sum[1]	PIN_F19	On-Board LED (LEDR1)
Sum[2]	PIN_E19	On-Board LED (LEDR2)
Sum[3]	PIN_F21	On-Board LED (LEDR3)

Sum[4]	PIN_F18	On-Board LED (LEDR4)
Sum[5]	PIN_E18	On-Board LED (LEDR5)
Sum[6]	PIN_J19	On-Board LED (LEDR6)
Sum[7]	PIN_H19	On-Board LED (LEDR7)
Sum[8]	PIN_J17	On-Board LED (LEDR8)
Sum[9]	PIN_G17	On-Board LED (LEDR9)
Sum[10]	PIN_J15	On-Board LED (LEDR10)
Sum[11]	PIN_H16	On-Board LED (LEDR11)
Sum[12]	PIN_J16	On-Board LED (LEDR12)
Sum[13]	PIN_H17	On-Board LED (LEDR13)
Sum[14]	PIN_F15	On-Board LED (LEDR14)
Sum[15]	PIN_G15	On-Board LED (LEDR15)
Ahex0[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
Ahex0[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
Ahex0[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
Ahex0[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
Ahex0[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
Ahex0[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
Ahex0[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
Ahex1[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
Ahex1[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
Ahex1[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])
Ahex1[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
Ahex1[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
Ahex1[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
Ahex1[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])
Ahex2[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
Ahex2[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
Ahex2[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
Ahex2[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
Ahex2[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
Ahex2[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
Ahex2[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
Ahex3[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
Ahex3[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
Ahex3[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
Ahex3[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
Ahex3[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
Ahex3[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
Ahex3[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
Bhex0[0]	PIN_AB19	On-Board seven-segment display segment (HEX4[0])
Bhex0[1]	PIN_AA19	On-Board seven-segment display segment (HEX4[1])
Bhex0[2]	PIN_AG21	On-Board seven-segment display segment (HEX4[2])
Bhex0[3]	PIN_AH21	On-Board seven-segment display segment (HEX4[3])
Bhex0[4]	PIN_AE19	On-Board seven-segment display segment (HEX4[4])
Bhex0[5]	PIN_AF19	On-Board seven-segment display segment (HEX4[5])
Bhex0[6]	PIN_AE18	On-Board seven-segment display segment (HEX4[6])
Bhex1[0]	PIN_AD18	On-Board seven-segment display segment (HEX5[0])
Bhex1[1]	PIN_AC18	On-Board seven-segment display segment (HEX5[1])
Bhex1[2]	PIN_AB18	On-Board seven-segment display segment (HEX5[2])
Bhex1[3]	PIN_AH19	On-Board seven-segment display segment (HEX5[3])
Bhex1[4]	PIN_AG19	On-Board seven-segment display segment (HEX5[4])
Bhex1[5]	PIN_AF18	On-Board seven-segment display segment (HEX5[5])
Bhex1[6]	PIN_AH18	On-Board seven-segment display segment (HEX5[6])

Bhex2[0]	PIN_AA17	On-Board seven-segment display segment (HEX6[0])
Bhex2[1]	PIN_AB16	On-Board seven-segment display segment (HEX6[1])
Bhex2[2]	PIN_AA16	On-Board seven-segment display segment (HEX6[2])
Bhex2[3]	PIN_AB17	On-Board seven-segment display segment (HEX6[3])
Bhex2[4]	PIN_AB15	On-Board seven-segment display segment (HEX6[4])
Bhex2[5]	PIN_AA15	On-Board seven-segment display segment (HEX6[5])
Bhex2[6]	PIN_AC17	On-Board seven-segment display segment (HEX6[6])
Bhex3[0]	PIN_AD17	On-Board seven-segment display segment (HEX7[0])
Bhex3[1]	PIN_AE17	On-Board seven-segment display segment (HEX7[1])
Bhex3[2]	PIN_AG17	On-Board seven-segment display segment (HEX7[2])
Bhex3[3]	PIN_AH17	On-Board seven-segment display segment (HEX7[3])
Bhex3[4]	PIN_AF17	On-Board seven-segment display segment (HEX7[4])
Bhex3[5]	PIN_AG18	On-Board seven-segment display segment (HEX7[5])
Bhex3[6]	PIN_AA14	On-Board seven-segment display segment (HEX7[6])
CO	PIN_F17	On-Board LED (LEDG8)

V. POST-LAB

1.) Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?

2.) For the adders, refer to the **Design Resources and Statistics** in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every SystemVerilog circuit.

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

Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes 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?

VI. REPORT

In your lab report, you should hand in the following:

- An introduction;
- Schematic block diagram of the bit-serial logic processor;
- Annotated design simulations of the bit-serial logic processor;
- Written description of the operation of your adder circuit;
- Written purpose and operation of each module, including the inputs/outputs of the modules;
- Schematic block diagrams with components, ports, and interconnections labeled for all adders;
- Design analysis comparison results from pre-lab;
- Answers to post-lab questions;
- A conclusion regarding what worked and what didn’t, with explanations of any possible causes and the potential remedies.